(July 9, 2009 1:47 p.m.)

A MATLAB Tutorial
Version 7

Ed Overman
Department of Mathematics
The Ohio State University

Introduction

MATLAB is an interactive software package which was developed to perform numerical calculations on vectors and matrices. Initially, it was simply a MATrix LABoratory. However, today it is much more powerful:

- It can do quite sophisticated graphics in two and three dimensions.
- It contains a high-level programming language (a "baby C") which makes it quite easy to code complicated algorithms involving vectors and matrices.
- It can numerically solve nonlinear initial-value ordinary differential equations.
- It can numerically solve nonlinear boundary-value ordinary differential equations.
- It contains a wide variety of toolboxes which allow it to perform a wide range of applications from science and engineering. Since users can write their own toolboxes, the breadth of applications is quite amazing.

Mathematics is the basic building block of science and engineering, and MATLAB makes it easy to handle many of the computations involved. You should not think of MATLAB as another complication programming language, but as a powerful calculator that gives you fingertip access to exploring interesting problems in science, engineering, and mathematics. And this access is available by using only a small number of commands and function[†] because MATLAB's basic data element is a matrix (or an array).

This is a crucial feature of MATLAB — it was designed to group large amounts of data in arrays and to perform mathematical operations on this data as individual arrays rather than as groups of data. This makes it very easy to apply complicated operations to the data, and it make it very difficult to do it wrong. In high-level computer languages you would usually have to work on each piece of data separately and use loops to cycle over all the pieces. In MATLAB this can frequently do complicated "things" in one, or a few, statements (and no loops). In addition, in a high-level language many mathematical operations require the use of sophisticated software packages, which you have to find and, much worse, to *understand* since the interfaces to these packages are frequently quite complicated and the documentation must be read and mastered. In MATLAB, on the other hand, these operations have simple and consistent interfaces which are quite easy to master. For an overview of the capabilities of MATLAB, type

```
≫ demo
```

in the `Help Navigator` and click on `MATLAB`.

This tutorial is designed to be a concise introduction to many of the capabilities of MATLAB. It makes no attempt to cover either the range of topics or the depth of detail that you can find in a reference manual, such as *Mastering MATLAB 7* by Duane Hanselman and Bruce Littlefield (which is over 850 pages long) or *MATLAB Guide, 2nd edition* by Desmond and Nicholas Higham (which is almost 400 pages long). This tutorial was initially written to provide students with a *free* "basic" overview of commands which are useful in an undergraduate course on linear algebra. Over the years it has grown to include courses in ordinary differential equations, mathematical modelling, and numerical analysis. It also includes an introduction to two- and three-dimensional graphics because graphics is often the preferred way to present the results of calculations.

In this tutorial MATLAB is first introduced as a calculator and then as a plotting package. Only afterwards are more technical topics discussed. We take this approach because most people are quite familiar with calculators, and it is only a small step to understand how to apply these same techniques to matrices rather than individual numbers or varibles. In addition, by viewing MATLAB as a simple but powerful calculater, rather than as a complicated software package or computer language, you will be in the correct frame of mind to use MATLAB.

You should view MATLAB as a tool that you are "playing with" — trying ideas out and seeing how

[†]There is a technical distinction between a *command* and a *function* in MATLAB: input arguments to commands are not enclosed in parentheses (they are separated by spaces) and there are no output arguments (i.e., a command cannot be on the right-hand side of an equal sign). In reality, this is a very fine distinction since many commands can be written as functions by putting the arguments between parentheses and separating them with commas. We will generally use the term *functions* unless there is a reason to make a distinction.

they work. If an idea works, fine; if it doesn't, investigate further and figure out why. Maybe you misunderstood some MATLAB command, or maybe your idea needs some refinement. "Play around" interactively and figure it out. There are no hard and fast rules for figuring it out — try things and see what happens. Don't be afraid to make mistakes; MATLAB won't call you an idiot for making a mistake. When you first learned to ride a bicycle, you fell down a lot — and you looked pretty silly. But you kept at it until you didn't fall down. You didn't study Newton's laws of motion and try to analyze the motion of a bicycle; you didn't take classes in how to ride a bicycle; you didn't get videos from the library on how to ride a bicycle. You just kept at it, possibly with the assistance of someone who steadied the bicycle and gave you a little push to get you started. This is how you should learn MATLAB.

However, this tutorial is not designed for "playing around". It is very ordered, because it has been designed as a brief introduction to all the basic topics that I consider important and then as a reference manual. It would be very useful for students to have a document which uses this "play around" approach so you would learn topics by using them in exploring some exercise. This is how workbooks should be written: present some exercise for students to investigate, and let them investigate it themselves. And these exercises should be interesting, having some connection to physical or mathematical models that the students — or at least a reasonable fraction thereof — have some knowledge of and some interest in. This tutorial is designed to be a reference manual that could be used alongside such a workbook — if only someone would write it.

Summary of Contents

We have tried to make this tutorial as linear as possible so that the building blocks necessary for a section are contained in preceding sections. This is not the best way to learn MATLAB, but it is a good way to document it. In addition, we try to separate these building blocks and put them in short subsections so that they are are easy to find and to understand. Next, we collect all the commands discussed in a subsection and put them in a box at the end along with a very brief discussion to make it easy to remember these commands. Finally, we collect all these commands and put them in the appendix again boxed up by topic. MATLAB has a **HUGE** number of commands and functions and this is one way to collect them for easy reference.

Warning: Usually we do not discuss the complete behavior of these commands, but only their most "useful" behavior. Typing

```
>> help <command>
```

or

```
>> doc <command>
```

gives you complete information about the command.

Notation: `help <command>` means to enter whatever command you desire (without the braces).
`help command` means to type these two words as written.

Section 1 of this tutorial discusses how to use MATLAB as a "scalar" calculator, and section 2 how to use it as a "matrix" calculator. Following this, you will be able to set up and solve the matrix equation $\mathtt{Ax} = \mathtt{b}$ where $\mathtt{A}$ is a square nonsingular matrix.

Section 4 discusses how to plot curves in two and three dimensions and how to plot surfaces in three dimensions. These three sections provide a "basic" introduction to MATLAB. At the end of each of these three sections there is a subsection entitled "Be Able To Do" which contains sample exercises to make sure you understand the basic commands discussed. (Solutions are included.)

You have hopefully noticed that we skipped section 3. It discusses a number of minor topics. Since they are useful in generating two- and three-dimensional plots, we have included it here.

The following sections delve more deeply into particular topics. Section 5 discusses how to find any and all solutions of $\mathtt{Ax} = \mathtt{b}$ where $\mathtt{A} \in \mathbb{C}^{m \times n}$ need not be a square matrix; there might be no solutions, one solution, or an infinite number to this linear system. When no solution exists, it discusses how to calculate a least-squares solution (i.e., the "best" approximation to a solution). In addition, it discusses how round-off errors can corrupt the solution, and how to determine if this is likely to occur.

Section 6 is quite brief and discusses advanced commands to input data into MATLAB and output it to a file. (The basic commands are discussed in subsection 4.1.) This is useful if the data is being shared

between various computer programs and/or software packages.

Section 7 discusses a number of commands which are useful in linear algebra and numerical linear algebra. Probably the most useful of these is calculating some or all of the eigenvalues of a square matrix.

Section 8 discusses MATLAB as a programming language — a very "baby C". Since the basic data element of MATLAB is a matrix, this programming language is *very* simple to learn and to use. Most of this discussion focuses on writing your own MATLAB commands, called function m-files (which are similar to functions in C and to functions, more generally subprograms, in Fortran). Using these functions, you can code a complicated sequence of statements such that all these statements as well as all the the variables used by these commands are hidden and will not affect the remainder of your MATLAB session. The only way to pass data into and out of these functions is through the argument list.

Section 9 discusses how to generate sparse matrices (i.e., matrices where most of the elements are zero). These matrices could have been discussed in section 2, but we felt that it added too much complexity at too early a point in this tutorial. Unless the matrix is **very large** it is usually not worthwhile to generate sparse matrices — however, when it is worthwhile the time and storage saved can be boundless.

Section 10 discusses how to use MATLAB to numerically solve initial-value ordinary differential equations. This section is divided up into a "basic" part and an "advanced" part. It often requires very little effort to solve even complicated odes; when it does we discuss in detail what to do and provide a number of examples. Section 11 discusses how to use MATLAB to numerically solve boundary-value ordinary differential equations.

Section 12 discusses how to numerically handle standard polynomial calculations such as evaluating polynomials, differentiating polynomials, and finding their zeroes. Polynomials and piecewise polynomials can also be used to interpolate data.

Section 13 discusses how to numerically calculate zeroes, extrema, and integrals of functions.

Section 14 discusses the discrete Fourier transform and shows how it arises from the continuous Fourier transform. We also provide an example which shows how to recover a simple signal which has been severely corrupted by noise.

Finally, section 15 discusses how to apply mathematical functions to matrices.

There is one appendix which collects all the commands discussed in this tutorial and boxes them up by topic. If a command has more than one use, it might occur in two or more boxes.

This tutorial closes with an index. It is designed to help in finding things that are "just on the tip of your tongue". All the MATLAB commands discussed here are listed at the beginning of the index, as well as alphabetically throughout the index.

1. Scalar Calculations

1.1. Simple Arithmetical Operations

MATLAB can be used as a scientific calculator. To begin a MATLAB session, type `matlab` or click on a MATLAB icon and wait for the prompt, i.e., "`≫`", to appear. (To exit MATLAB, type `exit` or `quit`.) You are now in the MATLAB *workspace*.

You can calculate $3.17 \cdot 5.7 + 17/3$ by entering

```
≫  3.17*5.7 + 17/3
```

and 2^{20} by entering

```
≫  2^20
```

And $\sum_{j=1}^{12} 1/j$ can be entered as

```
≫  1 + 1/2 + 1/3 + 1/4 + 1/5 + 1/6 + 1/7 + 1/8 + 1/9 + 1/10 + 1/11 + 1/12
```

You can enter a number in scientific notation using the "^" operator. For example, you can enter $2{\times}10^{-20}$ by

```
≫  2*10^-20
```

MATLAB, however, uses "e" to represent "10^" so that MATLAB displays

```
2.0000e-20
```

The "standard" way to input $2{\times}10^{-20}$ is as `2e-20` or `2E-20` or `2.e-20` or `2.E-20` (even `2.0000000e-00020` is acceptable).

Warning: 10^{-20} cannot be input as `e-20`, but must be input as `1e-20` or `1E-20` or `1.e-20` or `1.E-20` or ...

MATLAB can also handle complex numbers, where `i` or `j` represents $\sqrt{-1}$. For example, $5i$ can be input as `5i` or as `5*i`, while $5{\times}10^{30}i$ can be input as `5e30i` or as `5e30*i` or as `5*10^30*i`, **but not as** `5*10^30i` (which MATLAB considers to be $5{\times}10^{30i}$). To calculate $(2+2i)^4$, enter

```
≫  (2 + 2i)^4
```

and MATLAB returns -64.

You can also save all of your input to MATLAB and most of the output (plots are not saved) by using the `diary` command. This archive of your work can be invaluable when you are solving homework problems. You can later use an editor to extract the part you want to turn in, while "burying" all the false starts and typing mistakes that occur. Conversely, if you are involved in a continuing project, this archive can be invaluable in keeping a record of your progress.

If you do not specify a file, this archive is saved to the file `diary` (no extension) in the present directory. If the file already exists, this is appended to the end of the file (i.e., the file is not overwritten). Because of this feature you can use the `diary` command without fear that crucial work will be overwritten.

If you are entering a line and make a mistake, there are a number of ways you can correct your error:

- you can use the backspace or delete key to erase all the text back to your mistake,
- you can use the left-arrow key, i.e., "←", and the right-arrow key, i.e., "→", to move back and forth in the line, or
- you can use the mouse to move back and forth in the line.

Frequently, you will want to reexecute the previous line, or another previous line. For example, you might have made a mistake in the previous line and so it did not execute, or did not execute correctly. Of course, you can just retype the line — but, if it is very long, this can get very time-consuming. Instead, you can use the up-arrow key, i.e., "↑", to move backward, one statement at a time (or the down-arrow key, i.e., ↓" to move forward). Then hit the enter (or the return) key to execute the line.

Arithmetical Operations

`a + b`	Addition.	`a/b`	Division.
`a - b`	Subtraction.	`a\b`	Left division, (this is exactly the same as `b/a`).
`a*b`	Multiplication.	`a^b`	Exponentiation (i.e., a^b).

`diary` Saves your input to MATLAB and most of the output to disk. This command toggles `diary` on and off. (If no file is given, it is saved to the file `diary` in the current directory.)
`diary on` turns the diary on.
`diary off` turns the diary off.
`diary '<file name>'` saves to the named file.

↑ The up-arrow key moves backward in the MATLAB workspace, one line at a time.

1.2. Variables

Notation: We always use lowercase letters to denote scalar variables.

Variables can be used to store numerical values. For example, you can store the value $2^{1/3}$ in the variable `x` by entering

```
≫  x = 2^(1/3)
```

This variable can then be used on the right-hand side of an equation such as

```
≫  fx = 3*x^6 - 17*x^3 + 79
```

There can also be more than one command on a line. For example, if you type

```
≫  x = 2^(1/3); fx = 3*x^6 - 17*x^3 + 79; g = 3/fx;
```

then all three commands will be executed. Nothing will be printed out because semicolons follow each command. If you want everything printed out then type

```
≫  x = 2^(1/3), fx = 3*x^6 - 17*x^3 + 79, g = 3/fx
```

Thus, you can separate statements on a line by commas or semicolons. If semicolons are used, the results of the statement are not displayed, but if commas are used, the results appear on the computer screen.

Warning: A variable can be overwritten at will. For example, at present `x` $= 2^{1/3}$. If you now type

```
≫  x = x + 5
```

then `x` becomes $2^{1/3} + 5$. No warning messages are printed if a variable is overwritten, just as in a programming language.

Although we do not discuss vectors and matrices until the next section, it is important to understand that MATLAB considers scalar variables to be vectors of length one or matrices of size 1×1. For example, if you type

```
≫  fx
```

the number 57 is returned. But you can also type

```
≫  fx(1)
```

or

```
≫  fx(1,1)
```

and obtain the same result.

Character strings can also be stored in variables. For example, to store the string "And now for something completely different" in a variable, enter

```
≫  str = 'And now for something completely different'
```

(We discuss text variables in more detail in section 3.)

Note: To put a single quote mark into the string, use two single quote marks.

You can change a variable from a scalar to a vector or a matrix whenever you desire — or whenever you forget that the variable has already been defined. Unlike C, for example, variables do not needed to

be declared (or typed). A variable springs into existence the first time it is assigned a value, and its type depends on its context.

At start-up time, MATLAB also contains some predefined variables. Many of these are contained in the table below. Probably the most useful of these is `pi`.

Warning: Be careful since you can redefine these predefined variables. For example, if you type

```
≫  pi = 2
```

then you have redefined π — and no error messages will be printed out!

Another very useful predefined variable is `ans`, which contains the last calculated value which was not stored in a variable. For example, it sometimes happens that you forget to put a value into a variable. Then MATLAB sets the expression equal to the variable `ans`. For example, if you type

```
≫  (3.2*17.5 - 5/3.1)^2
```

but then realize that you wanted to save this value, simply enter

```
≫  x = ans
```

and `x` now contains $(3.2 \cdot 17.5 - 5/3.1)^2$.

In MATLAB it is trivial to display a variable: simply type it. For example, if `x` has the value -23.6 then

```
≫  x
```

returns

```
x =

  -23.6000
```

It is sometimes useful to display the value of a variable or an expression or a character string without displaying the name of the variable or `ans`. This is done by using `disp`. For example,

```
≫  disp(x)
≫  disp(pi^3)
≫  disp('And now for something completely different')
≫  disp('----------------------------------------')
```

displays

```
 -23.6000

  31.0063

And now for something completely different
----------------------------------------
```

(The command `fprintf`, which will be discussed in section 6, allows much finer formatting of variables.)

Note: When `disp` displays a variable or an array or an expression, it follows with a blank line. However, when it displays a string or a string variable, it does not.

Variables can also be deleted by using `clear`. For example, to delete x type

```
≫  clear x
```

Warning: **This is a very dangerous command because it is so easy to lose a great deal of work. If you mean to type**

```
≫  clear x
```

but instead you type

```
≫  clear
```

you will delete *all* the variables you have created in the workspace!

Predefined Variables

`ans`	The default variable name when one has not been specified.
`pi`	π.
`eps`	Approximately the smallest positive real number on the computer such that `1 + eps` $\neq$ `1`.
`Inf`	∞ (as in $1/0$). You can also type `inf`.
`NaN`	Not-a-Number (as in $0/0$). You can also type `nan`.
`i`	$\sqrt{-1}$.
`j`	$\sqrt{-1}$ (the same as `i` because engineers often use these interchangeably).
`realmin`	The smallest "usable" positive real number on the computer. This is "approximately" the smallest positive real number that can be represented on the computer (on some computer `realmin/2` returns 0).
`realmax`	The largest "usable" positive real number on the computer. This is "approximately" the largest positive real number that can be represented on the computer (on most computer `2*realmax` returns `Inf`).

About Variables

`Variables:` are case sensitive (so `xa` is not the same as `Xa`).
can contain up to 31 characters (but this is certainly "overkill").
must start with a letter, and can then be followed by any number of letters, numbers, and/or underscores (so `z_0__` is allowed).
do not need to be declared or typed.

`To display a variable, type it alone on a line.`
`To delete a variable, type clear <variable>.`
This is a very dangerous command — use it at your own risk.

`disp`	Displays a variable or an expression without printing the variable name or `ans`.
`,`	Separates multiple statements on the same line. The results appear on the screen.
`;`	When this ends a MATLAB command, the result is not printed on the screen. This can also separate multiple statements on the same line.

1.3. Round-off Errors

The most important principle for you to understand about computers is the following.

Principle 1.1. Computers cannot add, subtract, multiply, or divide correctly!

Computers do integer arithmetic correctly (as long as the numbers are not too large to be stored in the computer). However, computers cannot store most floating-point numbers (i.e., real numbers) correctly. For example, the fraction $\frac{1}{3}$ is equal to the real number $0.3333\ldots$ Since a computer cannot store this infinite sequence of threes, the number has to be truncated.

`eps` is "close to" the difference between the exact number $\frac{1}{3}$ and the approximation to $\frac{1}{3}$ used in MATLAB. It is defined to be the smallest positive real number such that `1 + eps > 1` (although it is not actually calculated quite this accurately). For example, in MATLAB `1 + 0.1` is clearly greater than 1; however, on our computer `1 + 1e-40` is not. To see this, when we enter

```
≫  (1 + .1) - 1
```

we obtain `0.1000` as expected.

Note: MATLAB guarantees that the expression in parentheses is evaluated first, and then `1` is subtracted from the result.

However, when we enter

```
≫  (1 + 1.e-40) - 1
```

MATLAB returns `0` rather than `1.e-40`. The smallest positive integer `n` for which

```
≫  (1 + 10^(-n)) - 1
```

returns `0` is computer dependent. (On our computer it is 16.) What is not computer dependent is that this leads to errors in numerical calculations. For example, when we enter

```
≫  n = 5; ( n^(1/3) )^3 - n
```

MATLAB returns `-1.7764e-15` rather than the correct result of 0. If you obtain 0, try some different values of `n`. You should be able to rerun the last statement executed without having to retype it by using the up-arrow key. Alternatively, on a Mac or a PC use the `copy` command in the menu; in Unix enter `^p`.

Note: It might not seem important that MATLAB does not do arithmetical operations *precisely*. However, you will see in subsection 5.2 that there are simple examples where this can lead to VERY incorrect results.

One command which is occasionally useful when you are just "playing around" is the `input` command, which displays a prompt on the screen and waits for you to enter some input from the keyboard. For example, if you want to try some different values of n in experimenting with the expression $(n^{1/3})^3 - n$, enter

```
≫  n = input('n = '); ( n^(1/3) )^3 - n
```

The argument to the command `input` is the string which prompts you for input, and the input is stored in the variable `n`; the semicolon keeps the result of this command from being printed out. You can easily rerun this line for different values of `n` (as we described above) and explore how round-off errors can affect simple expressions.

Warning: `eps` and `realmin` are very different numbers. `realmin` is approximately the smallest positive number that can be represented on the computer, whereas `eps` is approximately the smallest positive number on the computer such that $1 + \texttt{eps} \neq 1$. (`eps`/`realmin` is larger than the total number of atoms in the known universe.)

Request Input

`input('<prompt>')`	Displays the prompt on the screen and waits for you to enter whatever is desired.

1.4. Formatting Printing

The reason that $(n^{1/3})^3 - n$ can be nonzero numerically is that MATLAB only stores real numbers to a certain number of digits of accuracy. **Type**

```
≫  log10(1/eps)
```

and remember the integer part of this number. This is approximately the maximum number of digits of accuracy of any calculation performed in MATLAB. For example, if you type `1/3` in MATLAB the result is only accurate to approximately this number of digits. You do not see the decimal representation of `1/3` to this number of digits because on start-up MATLAB only prints the result to four decimal digits — or five significant digits if scientific notation is used (e.g., the calculation `1/30000` is displayed in scientific notation). To change how the results are printed out, use the `format` command in MATLAB. Use each of these four `format` functions and then type in `1/3` to see how the result is printed out.

Format Options

`format short`	The default setting.
`format long`	Results are printed to approximately the maximum number of digits of accuracy in MATLAB.
`format short e`	Results are printed in scientific notation using five significant digits.
`format long e`	Results are printed in scientific notation to approximately the maximum number of digits of accuracy in MATLAB.
`format short g`	Results are printed in the best of either `format short` or `format short e`.
`format long g`	Results are printed in the best of either `format long` or `format long e`.

1.5. Common Mathematical Functions

MATLAB contains a large number of mathematical functions. Most are entered exactly as you would write them mathematically. For example,

```
≫  sin(3)
≫  exp(2)
≫  log(10)
```

return exactly what you would expect. As is common in programming languages, the trig functions are evaluated in radians.†

Almost all the functions shown here are *built-in functions*. That is, they are coded in C so they execute very quickly. The one exception is the factorial function, i.e., $n! = 1 \cdot 2 \cdot 3 \cdots n$, which is calculated by

```
≫  factorial(n)
```

Note: This function is actually calculated by generating the vector $(1, 2, \ldots, n)$ and them multiplying all its elements together by `prod([1:n])`. (We discuss the colon operator in section 2.2.)

There is an important principle to remember about computer arithmetic in MATLAB.

Principle 1.2. If all the numbers you enter into MATLAB to do some calculation are "reasonably large" and the result of this calculation is one or more numbers which are "close to" `eps`, it is very likely that the number or numbers should be zero.

As an example, enter

```
≫  deg = pi/180; th = 40; 1 - ( cos(th*deg)^2 + sin(th*deg)^2 )
```

The result is `1.1102e-16`. Clearly, all the numbers entered into this calculation are "reasonable" and the result is approximately `eps`. Obviously, the result is supposed to be zero since, from the Pythagorean theorem

$$\cos^2\theta + \sin^2\theta = 1$$

for all angles θ. MATLAB tries to calculate the correct result, but it cannot quite. It is up to you to interpret what MATLAB is trying to tell you.

Note: If you obtained zero for the above calculation, try

```
≫  th = input('angle = '); 1 - ( cos(th*deg)^2 + sin(th*deg)^2 )
```

for various angles.‡ Some of these calculations should be nonzero.

There are a number of occasions in this overview where we reiterate that MATLAB cannot usually calculate results *exactly*. Sometimes these errors are small and unimportant — other times they are very important.

Warning: There is one technical detail about functions that will trip you up occasionally: how does MATLAB determine whether a word you enter is a variable or a function? The answer is that MATLAB first checks if the word is a variable and only if it fails does it check if the word is a function. For example, suppose you enter

```
≫  sin = 20
```

†A simple way to calculate $\sin 40^\circ$ is to type

```
≫  deg = pi/180; sin(40*deg)
```

‡Be sure to define `deg = pi/180` beforehand.

by mistake (possibly you meant `bin = 20` but were thinking about something else). If you now type

```
≫ sin(3)
```

MATLAB will reply

```
??? Index exceeds matrix dimensions.
```

because it recognizes that `sin` is a variable. Since MATLAB considers a variable to be a vector of length one, its complaint is that you are asking for the value of the third element of the vector `sin` (which only has one element). Similarly, if you enter

```
≫ sin(.25*pi)
```

MATLAB will reply

```
Warning: Subscript indices must be integer values.
```

because it thinks you are asking for the $.25\pi$-th element of the vector `sin`. The way to undo your mistake is by typing

```
≫ clear sin
```

Some Common Real Mathematical Functions

`abs(x)`	The absolute value of `x`.	`factorial(n)`	$n!$ for n a non-negative integer.
`acos(x)`	$\arccos$ `x`.	`fix(x)`	If `x` ≥ 0 this is the largest integer which is $\leq$ `x`. If `x` < 0 this is the smallest integer which is $\geq$ `x`.
`acosd(x)`	$\arccos$ `x` where the result is in degrees.		
`acosh(x)`	$\operatorname{arccosh}$ `x`.		
`acot(x)`	arccot `x`.	`floor(x)`	This is the largest integer which is $\leq$ `x`.
`acotd(x)`	arccot `x` where the result is in degrees.	`log(x)`	The natural log of `x`, i.e., $\log_e$`x`.
`acsc(x)`	arccsc `x`.	`log10(x)`	The common log of `x`, i.e., $\log_{10}$`x`.
`acscd(x)`	arccsc `x` where the result is in degrees.	`mod(x, y)`	The modulus after division. That is, `x` $-$ `n` $*$ `y` where `n` = `floor`(`y`/`x`).
`asin(x)`	$\arcsin$ `x`.		
`asind(x)`	$\arcsin$ `x` where the result is in degrees.		
`asinh(x)`	$\operatorname{arcsinh}$ `x`.	`rem(x, y)`	The remainder of `x`/`y`. This is *almost* the same as `mod(x, y)`. Warning: be careful if `x` < 0.
`atan(x)`	$\arctan$ `x`.		
`atand(x)`	$\arctan$ `x` where the result is in degrees.	`round(x)`	The integer which is closest to `x`.
`atan2(x, y)`	$\arctan$ `y`/`x` where the angle is in $(-\pi, +\pi]$.	`sec(x)`	$\sec$ `x`.
		`secd(x)`	$\sec$ `x` where `x` is in degrees.
`atanh(x)`	$\operatorname{arctanh}$ `x`.	`sign(x)`	If `x` > 0 this returns $+1$, if `x` < 0 this returns -1, and if `x` $= 0$ this returns 0.
`ceil(x)`	The smallest integer which is $\geq$ `x`.		
`cos(x)`	$\cos$ `x`.	`sin(x)`	$\sin$ `x`.
`cosd(x)`	$\cos$ `x` where `x` is in degrees.	`sind(x)`	$\sin$ `x` where `x` is in degrees.
`cosh(x)`	$\cosh$ `x`.	`sinh(x)`	$\sinh$ `x`.
`cot(x)`	$\cot$ `x`.	`sqrt(x)`	$\sqrt{\texttt{x}}$.
`cotd(x)`	$\cot$ `x` where `x` is in degrees.	`tan(x)`	$\tan$ `x`.
`csc(x)`	$\csc$ `x`.	`tand(x)`	$\tan$ `x` where `x` is in degrees.
`cscd(x)`	$\csc$ `x` where `x` is in degrees.	`tanh(x)`	$\tanh$ `x`.
`exp(x)`	$e^{\texttt{x}}$.		

1.6. Complex Numbers

MATLAB can work with complex numbers as easily as with real numbers. For example, to find the roots of the quadratic polynomial $x^2 + 2x + 5$ enter

```
≫  a = 1; b = 2; c = 5;
≫  x1 = ( -b + sqrt( b^2 - 4*a*c ) ) / (2*a)
≫  x2 = ( -b - sqrt( b^2 - 4*a*c ) ) / (2*a)
```

The output is

```
-1.0000 + 2.0000i
```

and

```
-1.0000 - 2.0000i
```

As another example, to calculate $e^{i\pi/2}$ enter

```
≫  exp(i*pi/2)
```

and obtain

```
0.0000 + 1.0000i
```

There are standard commands for obtaining the real part, the imaginary part, and the complex conjugate[†] of a complex number or variable. For example,

```
≫  x = 3 - 5i
≫  real(x)
≫  imag(x)
≫  conj(x)
```

returns `3`, `-5`, and `3.0000 + 5.0000i` respectively.

Note that many of the common mathematical functions can take complex arguments. Above, MATLAB has calculated $e^{i\pi/2}$, which is evaluated using the formula

$$e^z = e^{x+iy} = e^x(\cos y + i \sin y)\,.$$

Similarly,

$$\cos z = \frac{e^{iz} + e^{-iz}}{2} \quad \text{and} \quad \sin z = \frac{e^{iz} - e^{-iz}}{2i}\,.$$

Some Common Complex Mathematical Functions

`abs(z)`	The absolute value of `z` $= $ `x` $+ i$`y`.	`conj(z)`	`z`* $=$ `x` $- i$`y`.
`angle(z)`	The angle of `z`. This is calculated by `atan2(y, x)`.	`imag(z)`	The imaginary part of `z`, i.e., `y`.
		`real(z)`	The real part of `z`, i.e., `x`.

1.7. Script M-files

So far we have always entered MATLAB statements directly into the text window so that they are executed immediately. However, if we want to repeatedly execute a number of statements we have to put them all on one line and reexecute the whole line. This line can get very l o o o n n n g! The solution is to type the sequence of statements in a separate file named `<file name>.m`. It is easy to edit this file to remove any errors, and the sequence can be executed whenever desired by typing

```
≫  <file name>
```

The MATLAB statements themselves are not printed out, but the result of each statement is, unless a semicolon ends it. This type of file is called a *script m-file*: when MATLAB executes the command

[†]If a is a complex number, then its complex conjugate, denoted by a^* is obtained by changing the sign of i whenever it appears in the expression for a. For example, if $a = 3 + 17i$, then $a^* = 3 - 17i$; if $a = e^{i\pi/4}$, then $a^* = e^{-i\pi/4}$; if $a = (2 + 3i)\sin(1 + 3i)/(3 - \sqrt{5}\,i)$, then $a^* = (2 - 3i)\sin(1 - 3i)/(3 + \sqrt{5}\,i)$.

`<file name>` the contents of the file "`<file name>.m`" are executed just as if you had typed them into into the text window. We will not emphasize script m-files further, but you will find many occasions where they are very helpful.

However, there is one point we need to emphasize. **Make sure your file name is not the same as one of MATLAB's commands or functions.** If it is, your file will not execute — MATLAB's will! To check this, you can enter

```
≫  type <file name>
```

which types out the entire file `file name.m` if it is written in MATLAB and types out

```
<file name> is a built-in function.
```

(A built-in function is written in C or Fortran and so cannot be viewed directly.) It your file is typed out, fine; if not, change the name of your file.

A long expression can be continued to a new line by typing three periods followed by the "enter (or "return") key. For example, $\sum_{j=1}^{20} 1/j$ can be entered as

```
≫  1 + 1/2 + 1/3 + 1/4 + 1/5 + 1/6 + 1/7 + 1/8 + 1/9 + 1/10 + 1/11 + 1/12 + ...
1/13 + 1/14 + 1/15 + 1/16 + 1/17 + 1/18 + 1/19 + 1/20
```

although there are much better ways to obtain this same expression with many fewer keystrokes (as you will see in subsection 2.8). Lines can also be continued in the MATLAB workspace by using three periods, but it is much more common to use continuation in an m-file.

If your m-file is very long, it is often valuable to include comments to explain what you are doing. Each line of comments must begin with the percent character, i.e., "%". Comments can appear alone on a line or they can follow a statement that you have entered.

Note: By the way, comment lines are particularly useful if you write a number of script m-files. It is very easy to forget what this file is supposed to be doing.

Odds and Ends

`type`	Displays the actual MATLAB code for a command or function or m-file
`...`	Continue an expression onto the next line.
`%`	Begin a comment

1.8. Help!

Before discussing how to obtain help in MATLAB, here is a good place to discuss a very frustrating situation where you desperately need help: how do you abort a MATLAB statement which is presently executing. The answer is simply to type `^C` (that is, hold down the control key and type "c").

The on-line help facility in MATLAB is quite extensive. If you type

```
≫  help
```

you will get a list of all the topics that you can peruse further by typing `help` followed by the name of the topic. If you want help on a specific command, simply type `help` followed by the name of the command, i.e.,

```
help <command>
```

For example, if you forget the exact form of the `format` command, just type

```
≫  help format
```

and you will see all the various ways that the output can be formatted.

Note: Typing

```
≫  help ?
```

gives you lots of information about arithmetical and relational and logical operators and special characters.

There is a more general command that can help you determine which commands might be of use. The command `lookfor` searches through the first line of *all* MATLAB help entries for a particular keyword. It is case insensitive so capital letters need not be used. For example,

```
≫ lookfor plot
```

returns all the MATLAB commands that have something to do with plots. (There are over one hundred.) This command may be useful — or it may not be. However, it is worth a try if you cannot remember the name of the command you want to use.

Warning: *All* of the thousands of MATLAB commands have to be checked, so this command might run slowly.

Note: The keyword need not be a complete word. For example, the keyword `compl` is contained in the words "complement", "complex", "complete", "completion", and "incomplete" — and in the capitals of all these words.

If you want to find out more about a specific command, enter

```
≫ type <command>
```

If the command is written in MATLAB's programming language (as discussed in section 8), the entire function will be typed out. (The `type` command does not work on internal MATLAB commands, called *built-in function*, which are coded in C.)

MATLAB also has an entire reference manual on-line which can be accessed by entering

```
≫ doc
```

or

```
≫ helpdesk
```

This HTML documentation is displayed using your Web browser. It generally gives much more information than the `help` command, and in a more easily understood format.

After working for a while, you may well forget what variables you have defined in the workspace. Simply type `who` or `whos` to get a list of all your variables (but not their values). `who` simply returns the names of the variables you have defined, while `whos` also returns the size and type of each variable. To see what a variable contains, simply type the name of the variable on a line.

By the way, the demonstrations available by running `demo` show many of the capabilities of MATLAB and include the actual code used. This is always a good place to look if you are not sure how do do something.

Two commands that don't quite fit in any category are `save` and `load`. However, since these commands are occasionally very *helpful*, this is a good place to discuss them. Occasionally, you might need to save one or more MATLAB variables: it might have taken you some time to generate these variables and you might have to quit your MATLAB session without finishing your work — or you just might be afraid that you will overwrite some of them by mistake. The `save` command saves the contents of *all* your variables to the file "`matlab.mat`". Use `help` or `doc` to learn how to save all the variables to a file of your own choice and how to save just some of the variables. The `load` command loads all the saved variables back into your MATLAB session.† (As we discuss in subsection 4.1, the `load` command can also be used to input our own data into MATLAB.)

†These variables are saved in binary format; when loaded back in using `load` the variables will be *exactly* the same as before. The contents of this file can be viewed by the user with an editor — but the contents will appear to be gibberish. The contents can only be interpreted by the `load` command.

Getting Help

`help`	On-line help. `help` lists all the primary help topics. `help <command>` displays information about the command.
`doc`	On-line help reference manual in HTML format. `doc` accesses the manual. `doc <command>` displays information about the command.
`helpdesk`	Accesses the main page of the on-line reference manual.
`type <command>`	Displays the actual MATLAB code for this command.
`lookfor <keyword>`	Searches all MATLAB commands for this keyword.
`who`	Lists all the current variables.
`whos`	Lists all the current variables in more detail than `who`.
`demo`	Runs demonstrations of many of the capabilities of MATLAB.
`save`	Saves all of your variables.
`load`	Loads back all of the variables which have been saved previously.
`^C`	Abort the command which is currently executing (i.e., hold down the control key and type "c").

1.9. Be Able To Do

After reading this section you should be able to do the following exercises. The MATLAB statements are given on page 135.

1. Consider a triangle with sides a, b, and c and corresponding angles $\angle ab$, $\angle ac$, and $\angle bc$.
   (a) Use the law of cosines, i.e.,
   $$c^2 = a^2 + b^2 - 2ab\cos\angle ab\,,$$
   to calculate c if $a = 3.7$, $b = 5.7$, and $\angle ab = 79°$.
   (b) Then show c to its full accuracy.
   (c) Use the law of sines, i.e.,
   $$\frac{\sin\angle ab}{c} = \frac{\sin\angle ac}{b}\,,$$
   to calculate $\angle ac$ *in degrees* and show it in scientific notation.
   (d) What MATLAB command should you have used first if you wanted to save these results to the file `triangle.ans`?
2. Calculate $\sqrt[3]{1.2\times10^{20} - 12^{20}i}$.
3. Analytically, $\cos 2\theta = 2\cos^2\theta - 1$. Check whether this is also true numerically when using MATLAB by using a number of different values of θ. Use MATLAB statements which make it as easy as possible to do this.
4. How would you find out information about the `fix` command?

2. Arrays: Vector and Matrix Calculations

In the previous section we discussed operations using single numbers, i.e., scalars. In this section we discuss operations on sets of numbers called *arrays*. Until the advanced subsection at the end, we restrict our attention to one-dimensional arrays, which are called vectors, and two-dimensional arrays, which are called matrices. In this section we will generally refer to these sets of numbers specifically as vectors or matrices rather than use the more inclusive term "arrays". MATLAB was originally developed specifically to work with vectors and matrices and that is still one of its primary uses. For example, the multiplication of a matrix and a vector has a very specific meaning whereas the multiplication of two arrays does not.
Notation: **We will always write matrices using capital letters and vectors using lower case letters.**

This makes it much easier to understand MATLAB operations. **This is also a good practice for you to use**.

In addition, when we write "vector" we mean a column vector and so it is immediately obvious that `A*x` is a legitimate operation of a matrix times a vector as long as the number of rows of the matrix `A` equals the number of columns of the column vector `x`. Also, `x*A` is illegitimate because the column vector `x` has only one column while the matrix `A` is expected to have more than one row. On the other hand, `x'A` is legitimate (`x'` denotes the conjugate transpose of the vector `x`) as long as the row vector `x'` has the same number of columns as the number of rows of the matrix `A`.

In addition, we have very specific notation for denoting vectors and matrices and the elements of each. We collect all this notation here.

Notation: $\mathbb{R}^m$ denotes all real column vectors with m elements and $\mathbb{C}^m$ denotes all complex column vectors with m elements.

$\mathbb{R}^{m\times n}$ denotes all real $m\times n$ matrices (i.e., having m rows and n columns) and $\mathbb{C}^{m\times n}$ denotes all complex $m\times n$ matrices.

Notation: In this overview the word "vector" means a *column* vector so that $\mathbb{C}^m = \mathbb{C}^{m\times 1}$. Vectors are denoted by boldface letters, such as `x`; we will write a *row* vector as, for example, $\mathtt{x}^{\mathrm{T}}$, where $^{\mathrm{T}}$ denotes the transpose of a matrix or vector (that is, the rows and columns are reversed.)

Notation: $\mathtt{A} = (a_{ij})$ means that the (i,j)-th element of `A` (i.e., the element in the i-th row and the j-th column) is a_{ij}.

$\mathtt{x} = (x_i)$ means that the i-th element of `x` is x_i.

By the way MATLAB works with complex matrices as well as it does real matrices. To remind you of this fact, we will use $\mathbb{C}$ rather than $\mathbb{R}$ unless there is a specific reason not to. If there is a distinction between the real and complex case, we will first describe the real case and then follow with the complex case in parentheses.

2.1. Generating Matrices

To generate the matrix

$$\mathtt{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

in MATLAB type

```
» A = [1␣2␣3; 4␣5␣6; 7␣8␣9]
```

(where "␣" denotes one or more spaces) or

```
» A = [␣1␣2␣3␣; ␣4␣5␣6␣;␣7␣8␣9]
```

or

```
» A = [1,2,3; 4,5,6; 7,8,9]
```

or

```
» A = [␣1␣,␣2␣,␣3␣;␣4␣,␣5␣,␣6␣;␣7␣,␣8␣,␣9␣]
```

In other words, either spaces or commas can be used to delineate the elements of each row of a matrix; semicolons are required to separate rows. (Any number of spaces can be put around commas or semicolons to improve the readability of the expression.)

Notation: Since we prefer spaces, we will generally use them rather than commas to separate elements in a row.

Rows can also be separated by beginning each on a separate line. For example, the matrix `A` can also be entered by

```
» A = [1,2,3
4,5,6
7,8,9]
```

However, we consider this to be more work than simply using semicolons and will not use it again. The more complicated matrix

$$\mathtt{C} = \begin{pmatrix} 1 & 2+\sqrt{3} & 3\sin 1 \\ e^2 & 17/3 & \pi+3 \\ 1/3 & 2-\sqrt{3} & -7\cos\pi/7 \end{pmatrix}$$

can be entered by typing

```
≫  C = [ 1  2+sqrt(3)  3*sin(1); exp(2)  17/3  pi+3; 1/3  2-sqrt(3)  -7*cos(pi/7) ]
```

or

```
≫  C = [ 1, 2+sqrt(3), 3*sin(1); exp(2), 17/3, pi+3; 1/3, 2-sqrt(3), -7*cos(pi/7) ]
```

Warning: When an element of a matrix consists of more than one term, it is important to enter all the terms without spaces — unless everything is enclosed in parentheses. For example,

```
≫  x1 = [1  pi+3]
```

is the same as

```
≫  x2 = [1  pi+  3]
```

and is the same as

```
≫  x3 = [1  (pi  +3)]
```

but is not the same as

```
≫  x4 = [1  pi  +3]     % not the same as the previous three statements
```

(Try it!) In other words, MATLAB tries to understand what you mean, but it does not always succeed.

Definition

The *transpose* of a matrix $\mathtt{A} \in \mathbb{C}^{m\times n}$, denoted by $\mathtt{A}^{\mathrm{T}}$, is obtained by reversing the rows and columns of $\mathtt{A}$. That is, if $\mathtt{A} = (a_{ij})$ then $\mathtt{A}^{\mathrm{T}} = (a_{ji})$. (For example, the $(2,4)$ element of $\mathtt{A}^{\mathrm{T}}$, i.e., $i = 2$ and $j = 4$, is a_{42}.)

A square matrix $\mathtt{A}$ is *symmetric* if $\mathtt{A}^{\mathrm{T}} = \mathtt{A}$.

The *conjugate transpose* of a matrix $\mathtt{A} \in \mathbb{C}^{m\times n}$, denoted by $\mathtt{A}^{\mathrm{H}}$, is obtained by reversing the rows and columns of $\mathtt{A}$ and then taking the complex conjugates of all the elements. That is, if $\mathtt{A} = (a_{ij})$ then $\mathtt{A}^{\mathrm{H}} = (a^*_{ji})$, where * denotes the complex conjugate of a number.

A square matrix $\mathtt{A}$ is *Hermitian* if $\mathtt{A}^{\mathrm{H}} = \mathtt{A}$.

Note: In MATLAB $\mathtt{A}^{\mathrm{T}}$ is calculated by `A.'` (i.e., a period followed by a single quote mark).
In MATLAB $\mathtt{A}^{\mathrm{H}}$ is calculated by `A'` (i.e., just a single quote mark.)

A vector can be entered in the same way as a matrix. For example, the vector

$$\mathtt{x} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{pmatrix} = (1,2,3,4,5,6)^{\mathrm{T}}$$

can be entered as

```
≫  x = [1; 2; 3; 4; 5; 6]
```

However, this requires many semicolons; instead, take the transpose of a row vector by entering

```
≫  x = [1 2 3 4 5 6].'
```

where the MATLAB command for the transpose, i.e., "$^{\mathrm{T}}$", is "`.'`" (i.e., a period followed by a single quote mark). There is one further simplification that is usually observed when entering a vector. The MATLAB command for the conjugate transpose, i.e., "$^{\mathrm{H}}$", of a matrix is "`'`" (i.e., just a single quote mark), which requires one less character than the command for the transpose. Thus, `x` is usually entered as

```
≫  x = [1 2 3 4 5 6]'
```

Aside: In fact, `x` should be entered as

```
≫  x = [1:6]'
```

since this requires much less typing. (We will discuss the colon operator shortly.)

Warning: $\mathtt{x}^{\mathrm{T}} \rightarrow$ `x.'` while $\mathtt{x}^{\mathrm{H}} \rightarrow$ `x'` **so that you can only calculate** $\mathtt{x}^{\mathrm{T}}$ **by** `x'` **if** `x` **is real**. This has bitten us occasionally!

Sometimes the elements of a matrix are complicated enough that you will want to simplify the process of generating the matrix. For example, the vector $\mathtt{r} = \left(\sqrt{2/3}, \sqrt{2}, \sqrt{3}, \sqrt{6}, \sqrt{2/3}\right)^{\mathrm{T}}$ can be entered by typing

```
≫  s2 = sqrt(2); s3 = sqrt(3); r = [ s2/s3  s2  s3  s2*s3  s2/s3 ]'
```

We have now discussed how to enter matrices into MATLAB by using square parentheses, i.e., `[...]`. You work with individual elements of a matrix by using round parentheses, i.e., `(...)`. For example, the element a_{ij} of the matrix `A` is `A(i,j)` in MATLAB. Suppose you want to create the matrix

$$\mathtt{B} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{pmatrix}$$

without having to enter all nine elements. If `A` (see the beginning of this section) has already been generated, the simplest way is to type

```
≫  B = A; B(3,3) = 10
```

Also, the element x_i of the vector `x` is `x(i)` in MATLAB. For example, to create the column vector

$$\mathtt{x} = (1, 2, 3, \ldots, 47, 48, 49, 51)^{\mathrm{T}} \in \mathbb{R}^{50}$$

enter

```
≫  x = [1:50]'; x(50) = 51
```

or

```
≫  x = [1:50]'; x(50) = x(50) + 1
```

or

```
≫  x = [1:50]'; x(length(x)) = x(length(x)) + 1
```

where `length` returns the number of elements in a vector.

MATLAB also has a number of commands that can generate matrices. For example,

```
≫  C = zeros(5)
```

or

```
≫  C = zeros(5, 5)
```

generates a 5×5 zero matrix. Also,

```
≫  C = zeros(5, 8)
```

generates a 5×8 zero matrix. Finally, you can generate a zero matrix `C` with the same size as an already existing matrix, such as `A`, by

```
≫  C = zeros(size(A))
```

where `size(A)` is a row vector consisting of the number of rows and columns of `A`. This command is frequently used to *preallocate* a matrix of a given size so that MATLAB does not need to "guess" how large to make it.

Similarly, you can generate a matrix with all ones by `ones(n)` or `ones(m, n)` or `ones(size(D))`. You can also generate the *identity matrix*, i.e., the matrix with ones on the main diagonal and zeroes off of it, by using the command `eye` with the same arguments as above.

Another useful matrix is a random matrix, that is, a matrix whose elements are all random numbers. This is generated by the `rand` command, which takes the same arguments as above. Specifically, the elements are uniformly distributed random numbers in the interval $(0, 1)$. To be precise, these are *pseudo*random numbers because they are calculated by a deterministic formula which begins with an initial "seed" — which is called the *state* , not the seed. Every time that a new MATLAB session is started, the default seed is set, and so the same sequence of random numbers will be generated. However, every time that this command is executed during a session, a different sequence of random numbers is generated. If desired, a different seed can be set at any time by entering

```
≫  rand('state', <non-negative integer state number>)
```

Note: You can use the word "seed", rather than "state", as the first argument to `rand` but this causes MATLAB to use an older random number generator which has a period of $2^{31} - 2$. The period of the current default random number generator is 2^{1492}.

Random matrices are often useful in just "playing around" or "trying out" some idea or checking out some algorithm. The command `randn` generates a random matrix where the elements are normally distributed (i.e., Gaussian distributed) random numbers with mean 0 and standard deviation 1.

There is another "random" function which is useful if you want to reorder a sequence. The command `randperm(n)` generates a random permutation of the integers $1, 2, \ldots, n$.

Note: `randperm` changes the state of `rand`.

MATLAB also makes it convenient to assemble matrices in "pieces", that is, to put matrices together to make a larger matrix. That is, the original matrices are submatrices of the final matrix. For specificity, let us continue with `A` (see the beginning of this section). Suppose you want a 5×3 matrix whose first three rows are the rows of `A` and whose last two rows are all ones. This is easily generated by

```
≫  [ A ; ones(2, 3) ]
```

(The semicolon indicates that a row has been completed and so the next rows consist of all ones. The fact that `A` is a matrix in its own right is immaterial. All that is necessary is that the number of columns of `A` be the same as the number of columns of `ones(2, 3)`.) This matrix could also be generated by

```
≫  [ A ; ones(1, 3) ; ones(1, 3) ]
```

or by

```
≫  [ A ; [1 1 1] ; [1 1 1] ]
```

or even by

```
≫  [ A ; [1 1 1 ; 1 1 1] ]
```

Similarly, to generate a 3×4 matrix whose first three columns are the columns of `A` and whose last column is $(1,5,9)^{\mathrm{T}}$ type

```
≫  [A  [1 5 9]']
```

(The space following the `A` indicates that the next column is to follow. The fact that the next entry is a column vector is immaterial. All that is necessary is that the number of rows of `A` be the same as the number of rows in the new last column.)

Elementary Matrices

`zeros(n)`	Generates an $n\times n$ matrix with all elements being 0.
`zeros(m, n)`	Generates an $m\times n$ matrix.
`zeros(size(A))`	Generates a zero matrix with the same size as `A`.
`ones`	Generates a matrix with all elements being 1. The arguments are the same as for `zeros`.
`eye`	Generates the identity matrix, i.e., the diagonal elements are 1 and the off-diagonal elements are 0. The arguments are the same as for `zeros`.
`rand`	Generates a matrix whose elements are uniformly distributed random numbers in the interval $(0,1)$. Each time that this function is called during a session it returns different random numbers. The arguments are the same as for `zeros`. The initial seed is changed by `rand('seed', <seed number>)`.
`randn`	Generates a matrix whose elements are normally (i.e., Gaussian) distributed random numbers with mean 0 and standard deviation 1. Each time that this function is called during a session it returns different random numbers. The arguments are the same as for `zeros`.
`randperm(n)`	Generates a random permutation of the integers $1, 2, \ldots, n$.

`size(A)`	The size of a matrix as the row vector (m,n). Also, `size(A,1)` returns the number of rows (the first element of `A`) and `size(A,2)` returns the number of columns (the second element of `A`).
`length(x)`	The number of elements in a vector.

`A.'`	Transpose, i.e., $\mathtt{A}^{\mathrm{T}}$.	`A'`	Conjugate transpose, i.e., $\mathtt{A}^{\mathrm{H}}$.

2.2. The Colon Operator

For real numbers `a` and `b` the MATLAB command

```
≫  [a:b]
```

or, more simply,

```
≫  a:b
```

generates the row vector $(\mathtt{a}, \mathtt{a}+1, \mathtt{a}+2, \ldots, \mathtt{a}+k)$ where the integer k satisfies $\mathtt{a}+k \le \mathtt{b}$ and $\mathtt{a}+(k+1) > \mathtt{b}$. Thus, the vector $\mathtt{x} = (1,2,3,4,5,6)^{\mathrm{T}}$ should be entered into MATLAB as

```
≫  x = [1:6]'
```

or even as

```
≫  x = [1:6.9]'
```

(although we can't imagine why you would want to do it this way). If `c` is also a real number the MATLAB command

```
≫  [a:c:b]
```

or

```
≫  a:c:b
```

generates a row vector where the difference between successive elements is `c`. Thus, we can generate numbers in any arithmetic progression using the colon operator. For example, typing

```
≫  [18:-3:2]
```

generates the row vector $(18, 15, 12, 9, 6, 3)$. while typing

```
≫  [ pi : -.2*pi : 0 ]
```

generates the row vector $(\pi, .8\pi, .6\pi, .4\pi, .2\pi, 0)$.

Warning: There is a slight danger if `c` is not an integer. As an oversimplified example, entering

```
≫  x = [.02 : .001 : .98]'
```

should generate the column vector $(0.02, 0.021, 0.022, \ldots, 0.979, 0.98)^{\mathrm{T}}$. However, because of round-off errors in storing floating-point numbers, there is a possibility that the last element in `x` will be 0.979. The MATLAB package was written specifically to minimize such a possibility, but it still remains.[†] We will discuss the command `linspace` which avoids this difficulty in section 4. An easy "fix" to avoid this possibility is to calculate `x` by

```
≫  x = [20:980]'/1000
```

2.3. Manipulating Matrices

For specificity in this subsection we will mainly work with the 5×6 matrix

$$\mathtt{E} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 & 17 & 18 \\ 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 & 29 & 30 \end{pmatrix},$$

which can be generated by

```
≫  E = [ 1:6 ; 7:12 ; 13:18 ; 19:24 ; 25:30 ]
```

Note: Spaces will frequently be used in MATLAB commands in this subsection for readability.

You can use the colon notation to extract submatrices from `E`. For example,

```
≫  F = E( [1 3 5] , [2 3 4 5] )
```

extracts the elements in the first, third, and fifth rows and the second, third, fourth, and fifth columns of `E`; thus,

$$\mathtt{F} = \begin{pmatrix} 2 & 3 & 4 & 5 \\ 14 & 15 & 16 & 17 \\ 26 & 27 & 28 & 29 \end{pmatrix}.$$

[†]This possiblity is much more real in the programming language C. For example, the statement

```
for ( i = 0.02; i <= 0.98; i = i + .001 )
```

generates successive values of `i` by adding 0.001 to the preceding value. It is possible that when `i` *should* have the value 0.98, due to round-off errors the value will be slightly larger; the condition `i <= 0.98` will be false and the loop will not be evaluated when `i` should be 0.98.

You can generate this submatrix more easily by typing

```
≫  F = E( 1:2:5 , 2:5 )
```

There is an additional shortcut you can use: in a matrix a colon by itself represents an entire row or column. For example, the second column of `F` is `F(:,2)` and the second row is `F(2,:)`. To replace the second column of `F` by two times the present second column minus four times the fourth column enter

```
≫  F(:,2) = 2*F(:,2) - 4*F(:,4)
```

And suppose you now want to double all the elements in the last two columns of `F`. Simply type

```
≫  F(:,3:4) = 2*F(:,3:4)
```

Entering `E(:,:)` prints out exactly the same matrix as entering `E`. This is not a very useful way of entering `E`, but it shows how the colon operator can work. On the other hand, entering

```
≫  G = E( : , 6:-1:1 )
```

generates a matrix with the same size as `E` but with the columns reversed, i.e.,

$$\mathtt{G} = \begin{pmatrix} 6 & 5 & 4 & 3 & 2 & 1 \\ 12 & 11 & 10 & 9 & 8 & 7 \\ 18 & 17 & 16 & 15 & 14 & 13 \\ 24 & 23 & 22 & 21 & 20 & 19 \\ 30 & 29 & 28 & 27 & 26 & 25 \end{pmatrix} .$$

Finally, there is one more use of a colon. Entering

```
≫  f = E(:)
```

generates a column vector consisting of the columns of `E` (i.e., the first five elements of `f` are the first column of `E`, the next five elements of `f` are the second column of `E`, etc.).

Note: On the right side of an equation, `E(:)` is a column vector with the elements being the columns of `E` in order. On the left side of an equation, `E(:)` reshapes a matrix. However, we will not discuss this reshaping further because the `reshape` command described below is easier to understand.

The colon operator works on rows and/or columns of a matrix. A different command is needed to work on the diagonals of a matrix. For example, you extract the main diagonal of `E` by typing

```
≫  d = diag(E)
```

(so `d` is the column vector $(1, 8, 15, 22, 29)^{\mathrm{T}}$), one above the main diagonal by typing

```
≫  d1 = diag(E, 1)
```

(so `d1` is the column vector $(2, 9, 16, 23, 30)^{\mathrm{T}}$), and two below the main diagonal by typing

```
≫  d2 = diag(E, -2)
```

(so `d2` is the column vector $(13, 20, 27)^{\mathrm{T}}$).

The MATLAB command `diag` transforms a matrix (i.e., a non-vector) into a column vector. The converse also holds: when `diag` is applied to a vector, it generates a symmetric matrix. The command

```
≫  F = diag(d)
```

generates a $\underline{5{\times}5}$ matrix whose main diagonal elements are the elements of `d`, i.e., $1, 8, 15, 22, 29$, and whose off-diagonal elements are zero. Similarly, entering

```
≫  F1 = diag(d1, 1)
```

generates a $\underline{6{\times}6}$ matrix whose first diagonal elements (i.e., one above the main diagonal) are the elements of `d1`, i.e., $2, 9, 16, 23, 30$, and whose other elements are zero, that is,

$$\mathtt{F1} = \begin{pmatrix} 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 16 & 0 & 0 \\ 0 & 0 & 0 & 0 & 23 & 0 \\ 0 & 0 & 0 & 0 & 0 & 30 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} .$$

Finally, typing

```
≫  F2 = diag(d2, -2)
```

generates a 5×5 matrix whose -2-nd diagonal elements (i.e., two below the main diagonal) are the elements of `d2`, i.e., $13, 20, 27$, and whose other elements are zero, i.e.,

$$\texttt{F2} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 13 & 0 & 0 & 0 & 0 \\ 0 & 20 & 0 & 0 & 0 \\ 0 & 0 & 27 & 0 & 0 \end{pmatrix} .$$

You can also extract the upper triangular or the lower triangular part of a matrix. For example,

```
≫  G1 = triu(E)
```

constructs a matrix which is the same size as `E` and which contains the same elements as `E` on and above the main diagonal; the other elements of `G1` are zero. This command can also be applied to any of the diagonals of a matrix. For example,

```
≫  G2 = triu(E, 1)
```

constructs a matrix which is the same size as `E` and which contains the same elements as `E` on and above the first diagonal, i.e.,

$$\texttt{G2} = \begin{pmatrix} 0 & 2 & 3 & 4 & 5 & 6 \\ 0 & 0 & 9 & 10 & 11 & 12 \\ 0 & 0 & 0 & 16 & 17 & 18 \\ 0 & 0 & 0 & 0 & 23 & 24 \\ 0 & 0 & 0 & 0 & 0 & 30 \end{pmatrix} .$$

The similar command `tril` extracts the lower triangular part of a matrix.

As an example of the relationship between these three commands, consider the square random matrix `F` generated by

```
≫  F = rand(6)
```

All the following MATLAB commands calculate `F` anew:

```
≫  triu(F) + tril(F) - diag(diag(F))
≫  triu(F, 1) + diag(diag(F)) + tril(F, -1)
≫  triu(F) + tril(F, -1)
≫  triu(F, 2) + diag(diag(F, 1), 1) + tril(F)
```

Note: Numerically the first command might not generate *exactly* the same matrix as the following three because of round-off errors.

By the way, `diag`, `triu` and `tril` cannot appear on the left-hand side of an equation. Instead, to zero out all the diagonals above the main diagonal of `F` enter

```
≫  F = F - triu(F, 1)
```

and to zero out just the first diagonal above the main diagonal enter

```
≫  F = F - tril(triu(F, 1), 1)
```

What if you want to insert numbers from the upper right-hand corner of a matrix to the lower left-hand corner? There is no explicit function which does this but there are a number of indirect functions:

`fliplr(A)` flips the matrix from left to right, i.e., reverses the columns of the matrix;

`flipud(A)` flips the matrix up and down, i.e., reverses the rows of the matrix;

`rot90(A)` `rot90` rotates the matrix $90°$; and

`rot90(A,k)` rotates the matrix $k\times90°$.

MATLAB has a command which is useful in changing the shape of a matrix while keeping the same numerical values. The statement

```
≫  K = reshape(H, m, n)
```

reshapes the matrix $\texttt{H} \in \mathbb{C}^{p\times q}$ into $\texttt{K} \in \mathbb{C}^{m\times n}$ where m and n must satisfy $mn = pq$ (or an error message will be generated). A column vector is generated from `H`, as in `H(:)`, and the elements of `K` are taken columnwise from this vector. That is, the first m elements of this column vector go in the first column of `K`, the second m elements go in the second column, etc. For example, the matrix `E` which has been used throughout this subsection can be easily (and quickly) generated by

```
≫  E = reshape([1:30], 6, 5)'
```

Occasionally, there is a need to delete elements of a vector or rows or columns of a matrix. This is easily done by using the null matrix `[]`. For example, entering

```
≫  x = [1 2 3 4]'
≫  x(2) = []
```

results in $\mathtt{x} = (1, 3, 4)^{\mathrm{T}}$. As another example, you can delete the even columns of `G` by

```
≫  G( : , 2:2:6 ) = []
```

The result is

$$\mathtt{G} = \begin{pmatrix} 6 & 4 & 2 \\ 12 & 10 & 8 \\ 18 & 16 & 14 \\ 24 & 22 & 20 \\ 30 & 28 & 26 \end{pmatrix}.$$

Also, occasionally, there is a need to replicate or tile a matrix. That is, the command

```
≫  B = repmat(A, m, n)
```

generates a matrix `B` which contains m rows and n columns of copies of `A`. (If $n = m$ then `repmat(A, m)` is sufficient.) If `A` is a p by q matrix, then $\mathtt{B} \in \mathbb{R}^{mp \times nq}$. This even works if `A` is a scalar, in which case this is the same as

```
≫  B = A*ones(m, n)
```

(but it is much faster if m and n are large since no multiplication is involved).

Manipulating Matrices

`A(i,j)`	$a_{i,j}$.
`A(:,j)`	the j-th column of `A`.
`A(i,:)`	the i-th row of `A`.
`A(:,:)`	`A` itself.
`A(?1,?2)`	There are many more choices than we care to describe: `?1` can be `i` or `i1:i2` or `i1:i3:i2` or `:` or `[i1 i2 ... ir]` and `?2` can be `j` or `j1:j2` or `j1:j3:j2` or `:` or `[j1 j2 ... jr]`.
`A(:)`	On the right-hand side of an equation, this is a column vector containing the columns of `A` one after the other.
`diag(A)` `diag(A, k)`	A column vector of the k-th diagonal of the matrix (i.e., non-vector) `A`. If k is not given, then $k = 0$.
`diag(d)` `diag(d, k)`	A square matrix with the k-th diagonal being the vector `d`. If k is not given, then $k = 0$.
`triu(A)` `triu(A, k)`	A matrix which is the same size as `A` and consists of the elements on and above the k-th diagonal of `A`. If k is not given, then $k = 0$.
`tril(A)` `tril(A, k)`	The same as the command `triu` except it uses the elements on and *below* the k-th diagonalof `A`. If k is not given, then $k = 0$.
`fliplr(A)`	Flips a matrix left to right.
`flipud(A)`	Flips a matrix up and down.
`rot90(A)` `rot90(A, k)`	Rotates a matrix $k{\times}90°$. If k is not given, then $k = 1$.
`repmat(A, m, n)`	Generates a matrix with m rows and n columns of copies of `A`. (If $n = m$ the third argument is not needed.)
`reshape(A, m, n)`	Generates an $m{\times}n$ matrix whose elements are taken columnwise from `A`. Note: The number of elements in `A` must be mn.
`[]`	The null matrix. This is also useful for deleting elements of a vector and rows or columns of a matrix.

2.4. Simple Arithmetical Operations

Matrix Addition:

If $\mathtt{A}, \mathtt{B} \in \mathbb{C}^{m\times n}$ then the MATLAB operation

```
>> A + B
```

means $\mathtt{A}+\mathtt{B} = (a_{ij}) + (b_{ij}) = (a_{ij} + b_{ij})$. That is, the (i,j)-th element of $\mathtt{A}+\mathtt{B}$ is $a_{ij} + b_{ij}$.

Matrix Subtraction:

If $\mathtt{A}, \mathtt{B} \in \mathbb{C}^{m\times n}$ then the MATLAB operation

```
>> A - B
```

means $\mathtt{A}-\mathtt{B} = (a_{ij}) - (b_{ij}) = (a_{ij} - b_{ij})$.

Matrix Multiplication by a scalar:

If $\mathtt{A} \in \mathbb{C}^{m\times n}$ then for any scalar `c` the MATLAB operation

```
>> c*A
```

means $\mathtt{cA} = \mathtt{c}(a_{ij}) = (\mathtt{c}\,a_{ij})$. For example, the matrix $\mathtt{q} = (0, .1\pi, .2\pi, .3\pi, .4\pi, .5\pi)^{\mathrm{T}}$ can be generated by

```
>> q = [ 0 : .1*pi : .5*pi ]'
```

but more easily by

```
>> q = [ 0 : .1 : .5 ]'*pi
```

or

```
>> q = [0:5]'*.1*pi
```

Matrix Multiplication:

If $\mathtt{A} \in \mathbb{C}^{m\times \ell}$ and $\mathtt{B} \in \mathbb{C}^{\ell\times n}$ then the MATLAB operation

```
>> A*B
```

means $\mathtt{AB} = (a_{ij})(b_{ij}) = \left(\sum_{k=1}^{\ell} a_{ik}b_{kj}\right)$. That is, the (i,j)-th element of `AB` is $a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{i\ell}b_{\ell j}$.

Matrix Exponentiation:

If $\mathtt{A} \in \mathbb{C}^{n\times n}$ and p is a positive integer, then the MATLAB operation

```
>> A^p
```

means $\mathtt{A}^p = \underbrace{\mathtt{A}\mathtt{A}\cdots\mathtt{A}}_{p \text{ times}}$.

Matrix Exponentiation is also defined when p is not an integer. For example,

```
>> A = [1 2; 3 4]; B = A^(1/2)
```

calculates a complex matrix `B` whose square is `A`. (Analytically, $\mathtt{B}^2 = \mathtt{A}$, but numerically

```
>> B^2 - A
```

returns a non-zero matrix — however, all of its elements are less than $10 \cdot$ `eps` in magnitude.)

Note: For two values of p there are equivalent MATLAB commands:

$\mathtt{A}^{1/2}$ can also be calculated by `sqrtm(A)` and

$\mathtt{A}^{-1}$ can also be calculated by `inv(A)`.

Matrix Division:

The expression

$$\frac{\mathtt{A}}{\mathtt{B}}$$

makes no sense in linear algebra: if `B` is a square non-singular matrix it might mean $\mathtt{B}^{-1}\mathtt{A}$ or it might mean $\mathtt{A}\mathtt{B}^{-1}$. Instead, use the operation

```
>> A\b
```

to calculate the solution of the linear system $\mathtt{A}\mathtt{x} = \mathtt{b}$ (where `A` must be a square non-singular matrix) by Gaussian elimination. This is much faster computationally than calculating the solution of $\mathtt{x} = \mathtt{A}^{-1}\mathtt{b}$ by

```
>> x = inv(A)*b
```

Similarly,

```
A\B
```

solves $\mathtt{A}\mathtt{X} = \mathtt{B}$ by repeatedly solving $\mathtt{A}\mathtt{x} = \mathtt{b}$ where `b` is each column of `B` in turn and `x` is the corresponding column of `X`.

It is also possible to solve $\mathtt{x}^{\mathrm{T}}\mathtt{A} = \mathtt{b}^{\mathrm{T}}$ for $\mathtt{x}^{\mathrm{T}}$ by

```
b.'/A
```

and to solve `XA = B` by

```
B/A
```

(This is the same as solving $\mathtt{A}^{\mathrm{T}}\mathtt{X}^{\mathrm{T}} = \mathtt{B}^{\mathrm{T}}$.)

Elementwise Multiplication:

If $\mathtt{A}, \mathtt{B} \in \mathbb{C}^{m\times n}$, then the MATLAB operation

```
≫  A.*B
```

means $(a_{ij}b_{ij})$. That is, the (i,j)-th element of `A.*B` is $a_{ij}b_{ij}$. Note that this is not a *matrix* operation, but it is sometimes a useful operation. For example, suppose $\mathtt{y} \in \mathbb{R}^n$ has been defined previously and you want to generate the vector $\mathtt{z} = (1y_1, 2y_2, 3y_3, \ldots, ny_n)^{\mathrm{T}}$. You merely type

```
≫  z = [1:n]' .* y
```

(where the spaces are for readability). Recall that if $\mathtt{y} \in \mathbb{C}^n$ you will have to enter

```
≫  z = [1:n].' .* y
```

because you do not want to take the complex conjugate of the complex elements of `y`.

Elementwise Division:

If $\mathtt{A}, \mathtt{B} \in \mathbb{C}^{m\times n}$, then the MATLAB operation

```
≫  A./B
```

means (a_{ij}/b_{ij}).

Elementwise Left Division:

If $\mathtt{A}, \mathtt{B} \in \mathbb{C}^{m\times n}$, then the MATLAB operation

```
≫  B.\A
```

means the same as `A./B`

Elementwise Exponentiation:

If $\mathtt{A} \in \mathbb{C}^{m\times n}$, then

```
≫  A.^p
```

means (a_{ij}^{p}) and

```
≫  p.^A
```

means $(p^{a_{ij}})$. Also, if $\mathtt{A}, \mathtt{B} \in \mathbb{C}^{m\times n}$, then

```
A.^B
```

means $\left(a_{ij}^{b_{ij}}\right)$.

Where needed in these arithmetic operations, MATLAB checks that the matrices have the correct size. For example,

```
≫  A + B
```

will return an error message if `A` and `B` have different sizes, and

```
≫  A*B
```

will return an error message if the number of columns of `A` is not the same as the number of rows of `B`.

Note: There is one exception to this rule. When a scalar is added to a matrix, as in $\mathtt{A} + \mathtt{c}$, the scalar is promoted to the matrix `cJ` where `J` has the same size as `A` and all its elements are 1. That is,

```
≫  A + c
```

is evaluated as

```
≫  A + c*ones(size(A))
```

This is not a legitimate expression in linear algebra, but it is a very useful expression in MATLAB. For example, you can represent the function

$$y = 2\sin(3x + 4) - 5 \quad \text{for } x \in [2, 3]$$

by 101 data points using

```
≫  x = [2:.01:3]';
≫  y = 2*sin(3*x + 4) - 5
```

This is much more intelligible than calculating `y` using

```
>>  y = 2*sin(3*x + 4*ones(101, 1)) - 5*ones(101, 1)
```

In some courses that use vectors, such as statics courses, the *dot product* of the real vectors $\vec{a}$ and $\vec{b}$ is defined by

$$\vec{a} \bullet \vec{b} = \sum_{i=1}^{n} a_i b_i \, .$$

In linear algebra this is called the *inner product* and is defined for vectors $\mathtt{a}, \mathtt{b} \in \mathbb{R}^n$ by $\mathtt{a}^{\mathrm{T}}\mathtt{b}$. It is calculated by

```
>>  a'*b
```

(If $\mathtt{a}, \mathtt{b} \in \mathbb{C}^n$ the inner product is $\mathtt{a}^{\mathrm{H}}\mathtt{b}$ and is calculated by `a'*b`.) The *outer product* of these two vectors is defined to be $\mathtt{a}\mathtt{b}^{\mathrm{T}}$ and is calculated by

```
>>  a*b'
```

(If `a, b` are complex the outer product is $\mathtt{a}\mathtt{b}^{\mathrm{H}}$ and is calculated by `a*b'`.) It is important to keep these two products separate: the inner product is a scalar, i.e., $\mathtt{a}^{\mathrm{T}}\mathtt{b} \in \mathbb{R}$ (if complex, $\mathtt{a}^{\mathrm{H}}\mathtt{b} \in \mathbb{C}$), while the outer product is an $n \times n$ matrix, i.e., $\mathtt{a}\mathtt{b}^{\mathrm{T}} \in \mathbb{R}^{n \times n}$ (if complex, $\mathtt{a}\mathtt{b}^{\mathrm{H}} \in \mathbb{C}^{n \times n}$).

In linear algebra we often work with "large" matrices and are interested in the amount of "work" required to perform some operation. Previously, MATLAB kept track of the number of flops, i.e., the number of *fl*oating-point *op*eration*s*, performed during the MATLAB session. Unfortunately, this disappeared in version 6. Instead, we can calculate the amount of CPU time[†] required to execute a command by using `cputime`. This command returns the CPU time in seconds that have been used *since you began your MATLAB session*. This time is frequently difficult to calculate, and is seldom more accurate than to $^1/_{100}$-th of a second. Here is a simple example to determine the CPU time required to invert a matrix.

```
>>  n = input('n = '); time = cputime; inv(rand(n)); cputime - time
```

Warning: Remember that you have to subtract the CPU time used *before* the operation from the CPU time used *after* the operation.

By the way, you can also calculate the wall clock time required for some sequence of commands by using `tic` and `toc`. For example,

```
>>  tic; <sequence of commands>; toc
```

returns the time in seconds for this sequence of commands to be performed.

Note: This is very different from using `cputime`. `tic` followed by `toc` is exactly the same as if you had used a stopwatch to determine the time. Since a timesharing computer can be running many different processes at the same time, the elapsed time might be much greater than the CPU time. Normally, the time you are interested in is the CPU time.

[†]The CPU, Central Processing Unit, is the "guts" of the computer, that is, the hardware that executes the instructions and operates on the data.

Arithmetical Matrix Operations

`A + B`	Matrix addition.	`A.*B`	Elementwise multiplication.
`A - B`	Matrix subtraction.	`A.^p`	Elementwise exponentiation.
`A*B`	Matrix multiplication.	`p.^A`	
`A^n`	Matrix exponentiation.	`A.^B`	
`A\b`	The solution to $\mathtt{Ax} = \mathtt{b}$ by Gaussian elimination when `A` is a square non-singular matrix.	`A./B`	Elementwise division.
		`B.\A`	Elementwise left division, i.e., `B.\A` is exactly the same as `A./B`.
`A\B`	The solution to $\mathtt{AX} = \mathtt{B}$ by Gaussian elimination.		
`b/A`	The solution to $\mathtt{xA} = \mathtt{b}$ **where** `x` **and** `b` **are row vectors**.		
`B/A`	The solution to $\mathtt{XA} = \mathtt{B}$.		

`cputime`	Approximately the amount of CPU time (in seconds) used during this session.
`tic, toc`	Returns the elapsed time between these two commands.

2.5. Operator Precedence

It is important to list the precedence for MATLAB operators. That is, if an expression uses two or more MATLAB operators, in which order does MATLAB do the calculations? For example, what is `1:n+1`? Is it `(1:n)+1` or is it `1:(n+1)`? And if we solve $\mathtt{ACx} = \mathtt{b}$ by `A*C\b`, does MATLAB do `(A*C)\b` or `A*(C\b)`? The former is $\mathtt{C}^{-1}\mathtt{A}^{-1}\mathtt{b}$ while the latter is $\mathtt{A}^{-1}\mathtt{C}^{-1}\mathtt{b}$ — and these are different unless `A` and `C` commute. The following table shows the precedence of all MATLAB operators, that is, the order in which it evaluates an expression. The precedence is from highest to lowest. Within each precedence level, operators are evaluated from left to right in the expression.

Operator Precedence (highest to lowest) (operators with same precedence are separated by spaces)	
1	( )
2	.′ .^ ′ ^
3	+ [unary plus] - [unary minus] ~
4	.* ./ .\ * / \
5	+ [addition] - [subtraction]
6	:
7	< <= > >= == ~=
8	&
9	|
10	&&
11	||

The unary plus and minus are the plus and minus signs in $x = +1$ and $x = -1$. The plus and minus signs for addition and subtraction are, for example, $x = 5 + 1$ and $x = 10 - 13$. Thus, `1:n+1` is `1:(n+1)` because "`+`" has higher precedence than "`:`".† Also, `A*C\b = (A*C)\b` because "`*`" and "`\`" have the same precedence and so the operations are evaluated from left to right.

†On the other hand, in the statistical computer languages R and S (which are somewhat similar to MATLAB), "`:`" has higher precedence than "`+`" and so `1:n+1` is `(1:n)+1 = 2:(n+1)`.

2.6. Be Careful!

Be very careful: occasionally you might misinterpret how MATLAB displays the elements of a vector or matrix. For example, the MATLAB command `eig` calculates the eigenvalues of a square matrix. (We discuss eigenvalues in section 7.) To calculate the eigenvalues of the Hilbert matrix of order 5, i.e.,

$$\begin{pmatrix} 1 & 1/2 & 1/3 & 1/4 & 1/5 \\ 1/2 & 1/3 & 1/4 & 1/5 & 1/6 \\ 1/3 & 1/4 & 1/5 & 1/6 & 1/7 \\ 1/4 & 1/5 & 1/6 & 1/7 & 1/8 \\ 1/5 & 1/6 & 1/7 & 1/8 & 1/9 \end{pmatrix},$$

(we discuss this matrix in detail in subsection 5.2) enter

```
≫  format short
≫  eig(hilb(5))
```

MATLAB displays the eigenvalues as the column vector

```
ans =

   0.0000
   0.0003
   0.0114
   0.2085
   1.5671
```

You might think the the first element of this vector is 0. However, if it was zero MATLAB would display `0` and not `0.0000`. Entering

```
≫  format short e
≫  ans
```

displays

```
ans =

   3.2879e-06
   3.0590e-04
   1.1407e-02
   2.0853e-01
   1.5671e+00
```

which makes it clear that the smallest eigenvalue is far from zero.

On the other hand, if you enter

```
≫  format short
≫  A = [1 2 3; 4 5 6; 7 8 9]
≫  eig(A)
```

MATLAB displays

```
ans =

   16.1168
   -1.1168
   -0.0000
```

It might appear from our previous discussion that the last eigenvalue is not zero, but is simply too small to appear in this format. However, entering

```
≫  format short e
≫  ans
```

displays

```
ans =

    1.6117e+01
   -1.1168e+00
   -8.0463e-16
```

Since the last eigenvalue is close to `eps`, but all the numbers in the matrix `A` are of "reasonable size", you can safely assume that this eigenvalue is zero analytically. It only appears to be nonzero when calculated by MATLAB because **computers cannot add, subtract, multiply, or divide correctly!**

As another example of how you might misinterpret the display of a matrix, consider the Hilbert matrix of order two

$$\mathtt{H} = \begin{pmatrix} 1 & 1/2 \\ 1/2 & 1/3 \end{pmatrix} .$$

We write $\mathtt{H}^{100}$ as

$$\mathtt{H}^{100} \approx 10^{10} \begin{pmatrix} 1.5437 & 0.8262 \\ 0.8262 & 0.4421 \end{pmatrix} ,$$

while in MATLAB entering

```
≫  format short
≫  H = hilb(2)
≫  H^100
```

displays

```
ans =

   1.0e+10 *

    1.5437    0.8262
    0.8262    0.4421
```

It is very easy to miss the term "`1.0e+10 *`" because it stands apart from the elements of the matrix.

Similarly, entering

```
≫  format short
≫  H = hilb(2)
≫  ( H^(1/2) )^2 - H
```

should result in the zero matrix, since $(\mathtt{H}^{1/2})^2 = \mathtt{H}$. However, MATLAB displays

```
ans =

   1.0e-15 *

    0.2220    0
         0    0
```

where, again, it is easy to miss the term "`1.e-15 *`" and not realize that this matrix is **very** small — in fact, it *should* be zero.

Be careful: MATLAB has finite memory. You should have no problem creating a matrix by

```
≫  A = zeros(1000)
```

but you might well have a problem if you enter

```
≫  A = zeros(10000)
```

The amount of memory available is dependent on the computer and the operating system and is very hard to determine. Frequently it is much larger than the amount of physical memory on your computer. But, even if you have sufficient memory, MATLAB may slow to a crawl and become unusable. The `whos` command will tell you how much memory you are using and show you the size of all your variables. If you have large matrices which are no longer needed, you can reduce their sizes by equating them to the null matrix, i.e., `[]`, or remove them entirely by using `clear`.

Warning: Recall that the `clear` command is very dangerous because `clear A` deletes the variable `A` but `clear` (without anything following) **deletes all variables**!

2.7. Common Mathematical Functions

In linear algebra mathematical functions cannot usually be applied to matrices. For example, $e^{\mathtt{A}}$ and $\sin \mathtt{A}$ have no meaning unless `A` is a square matrix. (We will discuss their mathematical definitions in section 15.)

Here we are interested in how MATLAB applies common mathematical functions to matrices and vectors. For example, you might want to take the sine of every element of the matrix $\mathtt{A} = (a_{ij}) \in \mathbb{C}^{m \times n}$, i.e., $\mathtt{B} = (\sin a_{ij})$. This is easily done in MATLAB by

```
≫  B = sin(A)
```

Similarly, if you want $\mathtt{C} = (e^{a_{ij}})$, enter

```
≫  C = exp(A)
```

Also, if you want $\mathtt{D} = \left(\sqrt{a_{ij}}\right)$ type

```
≫  C = sqrt(A)
```

or

```
≫  C = A.^(1/2)
```

All the common mathematical functions in the table entitled "Some Common Real Mathematical Functions" in subsection 1.5 can be used in this way.

As we will see in the section on graphics, this new interpretation of mathematical functions makes it easy in MATLAB to graph functions without having to use the MATLAB programming language.

2.8. Data Manipulation Commands

MATLAB has a number of "simple" commands which are used quite frequently. Since many of them are quite useful in analyzing data, we have grouped them around this common "theme".

To calculate the maximum value of the vector `x`, type

```
≫  m = max(x)
```

If you also want to know the element of the vector which contains this maximum value, type

```
≫  [m, i] = max(x)
```

If the elements of the vector are all real, the result of this command is the element which has the maximum value. However, if any of the elements of `x` are complex (i.e., non-real), this command has no mathematical meaning. MATLAB *defines* this command to determine the element of the vector which has the maximum *absolute value* of the elements of `x`.

Warning: Make sure you understand the description of `max` if you every apply it to non-real vectors. For example, if $\mathtt{x} = (-2, 1)^{\mathrm{T}}$ then `max(x)` returns 1 as expected. However, if $\mathtt{x} = (-2, i)^{\mathrm{T}}$ then `max(x)` returns -2. This is because the element which has the largest absolute value is -2. Thus, if `x` is a non-real vector, then `max(x)` is **not** the same as `max(abs(x))`.

Since the columns of the matrix `A` can be considered to be vectors in their own right, this command can also be applied to matrices. Thus,

```
≫  max(A)
```

returns a *row* vector of the maximum element in each of the columns of `A` if all the elements of `A` are real. If any of the elements of `A` are non-real, this command returns the element in each column which has the maximum *absolute value* of all the elements in that column.

To find the maximum value of an entire real matrix, type

```
≫  max(max(A))
```

or

```
≫  max(A(:))
```

and to find the maximum absolute value of an entire real or complex matrix, type

```
≫  max(max(abs(A)))
```

or

```
≫  max(abs(A(:)))
```

The command `min` acts similarly to `max` except that it finds the minimum value (or element with the minimum absolute value) of the elements of a vector or the columns of a matrix.

To calculate the sum of the elements of the vector `x`, type

```
≫ sum(x)
```

`sum` behaves similarly to `max` when applied to a matrix. That is, it returns the row vector of the sums of each column of the matrix. This command is sometimes useful in adding a deterministic series. For example,

```
≫ 1 + 1/2 + 1/3 + 1/4 + 1/5 + 1/6 + 1/7 + 1/8 + 1/9 + 1/10 + 1/11 + 1/12 + ...
1/13 + 1/14 + 1/15 + 1/16 + 1/17 + 1/18 + 1/19 + 1/20
```

is entered much more easily as

```
≫ sum(ones(1, 20)./[1:20])
```

or even as

```
≫ sum(1./[1:20])
```

The mean, or average, of these elements is calculated by

```
≫ mean(x)
```

where `mean(x) = sum(x)/length(x)`.

`std` calculates the standard deviation of the elements of a vector. The standard deviation is a measure of how much a set of numbers "vary" and is defined as

$$\texttt{std(x)} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\bigl(x_i - \langle\,\texttt{x}\,\rangle\bigr)^2}$$

where $\langle\,\texttt{x}\,\rangle$ is the mean of the elements.

MATLAB can also sort the elements of the vector `x` in increasing order by

```
≫ sort(x)
```

If the vector is non-real, the elements are sorted in increasing absolute value. (If two elements have the same absolute value, the one with the smaller absolute angle in polar coordinates is used.)

The MATLAB command `diff` calculates the difference between successive elements of a vector. For example, if $\texttt{x} \in \mathbb{R}^n$ then the command

```
≫ s = diff(x)
```

generates the vector $\texttt{s} \in \mathbb{R}^{n-1}$ which is defined by $s_i = x_{i+1} - x_i$. There are a number of uses for this command. For example,

- if `s` has been sorted, then `diff(s) == 0` can be used to test if any elements of `s` are repeated (or the number that are repeated.
- similarly, `all(diff(x)) > 0` tests if the elements of `s` are monotonically increasing.
- a numerical approximation to the derivative of $y = f(x)$ can be calculated by `diff(y)./diff(x)`.

The MATLAB function which is almost the inverse of `diff` is `cumsum` which calculates the cumulative sum of the elements of a vector or matrix. For example, if $\texttt{s} \in \mathbb{R}^{n-1}$ has been generated by `s = diff(x)`, then

```
≫ c = cumsum(s)
```

generates the vector $\texttt{c} \in \mathbb{R}^{n-1}$ where $c_i = \sum_{j=1}^{i} s_j$. We can recover `x` by

```
≫ xrecovered = zeros(length(x),1)
≫ xrecovered(1) = x(1)
≫ xrecovered(2:length(x)) = x(1) + c
```

There are also a number of MATLAB commands which are particularly designed to plot data. The commands we have just discussed, such as the average and standard deviation, give a coarse measure of the distribution of the data. To actually "see" what the data looks like, it has to be plotted. Two particularly useful types of plots are histograms (which show the distribution of the data) and plots of data which include error bars. These are both discussed in subsection 4.1.

Although it does not quite fit here, sometimes you want to know the length of a vector `x`, which is $\sqrt{x_1^2 + x_2^2 + \cdots x_n^2}$. (Note that this is **not** `length(x)` which returns the number of elements in `x`, i.e., n.) This length, which is often called the Euclidean length, can be calculated by entering

```
≫ sqrt( x'*x )
```

but it can be entered more easily by

```
≫  norm(x)
```

(As we discuss in section 7, the norm of a vector is a more general concept than simply the Euclidean length.)

Data Manipulation Commands

`max(x)`	The maximum element of a real vector. `[m, i] = max(x)` also returns the element which contains the maximum value in `i`.
`max(A)`	A row vector containing the maximum element in each column of a matrix. `[m, i] = max(A)` also returns the element in each column which contains the maximum value in `i`.
`min(x)` `min(A)`	The sum of the elements of a vector, or a row vector containing the sum of the elements in each column in a matrix.
`mean(x)` `mean(A)`	The mean, or average, of the elements of a vector, or a row vector containing the mean of the elements in each column in a matrix.
`norm(x)`	The Euclidean length of a vector.
`norm(A)`	The matrix norm of `A`. *Note*: the norm of a matrix is **not** the Euclidean length of each column in the matrix.
`prod(x)` `prod(A)`	The product of the elements of a vector, or a row vector containing the product of the elements in each column in a matrix.
`sort(x)` `sort(A)`	Sorts the elements in increasing order of a real vector, or in each column of a real matrix.
`std(x)` `std(A)`	The standard deviation of the elements of a vector, or a row vector containing the standard deviation of the elements in each column in a matrix.
`sum(x)` `sum(A)`	The sum of the elements of a vector, or a row vector containing the sums of the elements in each column in a matrix.
`diff(x)` `diff(A)`	The difference between successive elements of a vector, or between successive elements in each column of a matrix.
`cumsum(x)` `cumsum(A)`	The cumulative sum between successive elements of a vector, or between successive elements in each column of a matrix.

2.9. Advanced Topic: Multidimensional Arrays

We have already discussed 1-D arrays (i.e., vectors) and 2-D arrays (i.e., matrices). Since these are two of the most fundamental objects in linear algebra, there are many operations and functions which can be applied to them. In MATLAB you can also use *multidimensional* arrays (i.e., n-D arrays).

A common use for multidimensional arrays is simply to hold data. For example, suppose a company produces three products and we know the amount of each product produced each quarter; the data naturally fits in a 2-D array, i.e., (product, amount). Now suppose the company has five sales regions so we split the amount of each product into these regions; the data naturally fits in a 3-D array, i.e., (product, region, amount). Finally, suppose that each product comes in four colors; the data naturally fits in a 4-D array, i.e., (product, color, region, amount).

For another example, a 3-D array might be the time evolution of 2-D data. Suppose we record a grey scale digital image of an experiment every minute for an hour. Each image is stored as a matrix `M` with $m_{i,j}$ denoting the value of the pixel positioned at (x_i, y_j). The 3-D array `Mall` can contain all these images: `Mall(i,j,k)` denotes the value of the pixel positioned at (x_i, y_j) in the k-th image. The entire k-th image is `Mall(:,:,k)` and it is filled with the k-th image `M` by

```
≫  Mall(:,:,k) = M
```

If you want to multiply `M` by another matrix `A`, you can use `M*A` or `Mall(:,:,k)*A`; if you want to average the first two images you can use `.5*(Mall(:,:,1)+Mall(:,:,2))`.

Many MATLAB functions can be used in n-D, such as `ones`, `rand`, `sum`, and `size`. The `cat` function is particularly useful in generating higher-dimensional arrays. For example, suppose we have four matrices `A`, `B`, `C`, and `D` $\in \mathbb{R}^{2\times 7}$ which we want to put into a three-dimensional array. This is easily done by

```
≫  ABCD = cat(3, A, B, C, D)
```

which concatenates the four matrices using the third dimension of `ABCD`. (The "3" denotes the third dimension of `ABCD`.) And it is much easier than entering

```
≫  ABCD(:,:,1) = A;
≫  ABCD(:,:,2) = B;
≫  ABCD(:,:,3) = C;
≫  ABCD(:,:,4) = D;
```

If instead, we enter

```
≫  ABCD = cat(j, A, B, C, D)
```

then the four matrices are concatenated along the j-th dimension of `ABCD`. That is, `cat(1, A, B, C, D)` is the same as `[A, B, C, D]` and `cat(2, A, B, C, D)` is the same as `[A; B; C; D]`.

Another useful command is `squeeze` which squeezes out dimensions which only have one element. For example, if we enter

```
≫  E = ABCD(:,2,:)
```

(where the array `ABCD` was created above), then we might think that `E` is a matrix whose columns consist of the second columns of `A`, `B`, `C`, and `D`. However, `size(E) = 2 1 4` so that `E` is a *three*-dimensional array, not a two-dimensional array. We obtain a two-dimensional array by `squeeze(E)`.

Multidimensional Array Functions

`cat`	Concatenates arrays; this is useful for putting arrays into a higher-dimensional array.
`squeeze`	Removes (i.e., squeezes out) dimensions which only have one element.

2.10. Be Able To Do

After reading this section you should be able to do the following exercises. The answers are given on page 135.

1. Consider the matrix
$$\mathtt{A} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}.$$
(a) Enter it in the following three ways:
   (i) type in all 16 elements directly.
   (ii) since each row is in arithmetic progression, use the colon operator to enter each row.
   (iii) since each column is in arithmetic progression, use the colon operator (and the transpose operator) to enter each column.

   (b) Multiply the second row of `A` by $-9/5$, add it to the third row, and put the result back in the second row. Do this all using <u>one</u> MATLAB statement.
2. Generate the tridiagonal matrix
$$\mathtt{A} = \begin{pmatrix} 4 & -1 & & & & \\ -1 & 4 & -1 & & & 0 \\ & -1 & 4 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & -1 & 4 & -1 \\ & & & & -1 & 4 \end{pmatrix} \in \mathbb{R}^{n\times n}$$

where the value of n has already been entered into MATLAB.

3. Generate the tridiagonal matrix

$$\mathtt{A} = \begin{pmatrix} 1 & -1 & & & & & \\ e^1 & 4 & -1 & & & & 0 \\ & e^2 & 9 & -1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & e^{n-1} & (n-1)^2 & -1 & \\ 0 & & & & e^n & n^2 & \end{pmatrix} \in \mathbb{R}^{n\times n}$$

where the value of n has already been entered into MATLAB.

4. Consider the matrix

$$\mathtt{A} = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & -5 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

(a) Enter it using as few keystrokes as possible. (In other words, don't enter the elements individually.)
(b) Zero out all the elements of `A` below the diagonal.

5. Enter the column vector

$$x = (0, 1, 4, 9, 16, 25, \ldots, 841, 900)^{\mathrm{T}}$$

using as few keystrokes as possible. (In other words, don't enter the elements individually.)

6. (a) Generate a random 5×5 matrix `R`.
(b) Determine the largest value in each row of `R` and the element in which this value occurs.
(c) Determine the average value of all the elements of `R`.
(d) Generate the matrix `S` where every element of `S` is the sine of the corresponding element of `R`.
(e) Put the diagonal elements of R into the vector `r`.

7. Generate the matrix

$$\mathtt{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{pmatrix}.$$

(a) Calculate a matrix `B` which is the square root of `A`. That is, $\mathtt{B}^2 = \mathtt{A}$. Also, calculate a matrix `C` each of whose elements is the square root of the corresponding element of `A`.
(b) Show that the matrices you have obtained in (a) are correct by substituting the results back into the original formulas.

3. Anonymous Functions, Strings, and Other Data Types

Now that we have discussed scalar and matrix calculations, the next important topic is graphics. However, there are a number of minor topics which are useful in graphics and so we collect them here. First, anonymous functions allow us to easily define a function which we can then plot. Second, some graphics functions require that the name of a function be passed as an argument. Third, character strings are necessary in labelling plots. And, finally, cell arrays are occasionally helpful in labelling plots. Cell arrays are generally used to manage data and since structures are also used to manage data we also include them here. Another reason is that there are a number of *data types* in MATLAB including numeric, text variables, cell arrays, and structures. (A "minor" data type is logical, which we discuss in section 8.1.) We might as well get the last three out of the way at once.

3.1. Anonymous Functions

In MATLAB it is common to define a mathematical function in a separate file as we discuss in subsection 8.3. (This is similar to writing a function or subroutine or subprogram in a high-level computer language.) However, if the mathematical function is particularly simple, that is, it can be written as one simple expression, we can define it in MATLAB using an *anonymous function*. If our function is

$$f(< \texttt{arg1} >, < \texttt{arg2} >, \dots) = < \texttt{expression} >$$

the MATLAB statement is

```
» f = @(<arg1>, <arg2>, ...) <expression>
```

For example, we can define the function

$$f(t) = t^5 e^{-2t} \cos(3t)$$

by

```
» f = @(t) t.^5 .* exp(-2*t) .* cos(3*t)
```

and then evaluate it by

```
» x = [0:.01:1]'
» fx = f(x)
» A = rand(5)
» fA = f(A)
```

More generally, we can define

$$g(x, y, a, b, c) = x^a e^{-bx} \cos(cy)$$

by

```
» g = @(x, y, a, b, c) x.^a .* exp(-b.*x) .* cos(c.*y)
```

in which case any of the input arguments can be in $\mathbb{R}$ or in $\mathbb{R}^n$. It is also possible — although probably not very useful — to let g have *one* vector argument, say $\texttt{x} = (x, y, a, b, c)^{\mathrm{T}}$ by

```
» g_ = @(x) x(1)^x(3) * exp(-x(4)*x(1)) * cos(x(5)*x(2))
```

(In this example there is no purpose to use `.*` or `.^`.)

Warning: It is quite easy to forget to put dots (i.e., "`.`") before the mathematical operations of multiplication (i.e., `*`), division (i.e., `/`), and exponentiation (i.e., `^`). For example, if f is defined by

```
» f = @(t) t^5 * exp(-2*t) * cos(3*t)
```

then

```
» f(3)
```

is allowed, but not

```
» f([1:10])
```

Be careful!

The syntax for defining an anonymous function is

```
» @(<argument list>) <expression>
```

(Since there is no left-hand side to this expression, the name of this function is `ans`.) The symbol `@` is the MATLAB operator that constructs a *function handle*. This is similar to a pointer in C which gives the address of a variable or of a function. The name "handle" is used in MATLAB to denote a variable which refers to some "object" which has been created. Thus, we can think of an anonymous function as being created by

```
(<argument list>) <expression>
```

and the handle to the function (in C, the address of the function) being returned by using `@`. By the way, we can create a function handle to a MATLAB function by, for example,

```
» f = @cos
```

so that `f(3)` is the same as `cos(3)`. We give an example where this is very useful in section 3.4

It is important to understand that all user-defined variables which appear in `<expression>` must **either** appear in the argument list **or** be defined before the function is defined. *If the variable does not appear in the argument list, then its value is fixed when the function is defined.* For example, if a very simple function is defined by

```
» r = 10
» h = @(x) r*x
```

then the function is $h(x) = 10x$ even if `r` is modified later. Thus,

```
≫  h(5)
```

returns 50 and so does

```
≫  r = 0
≫  h(5)
```

Warning: Don't forget that if a variable does not appear in the argument list, then its value is *fixed* when the function is defined.

A function can also be defined — **but don't do it** —by the `inline` command. For example, the function f defined above can also be defined by

```
≫  f = inline('t.^5 .* exp(-2*t) .* cos(3*t)', 't')
```

In general, if our function is

$$f(< \texttt{arg1} >, < \texttt{arg2} >, \ldots) = < \texttt{expression} >$$

the MATLAB statement is

```
≫  f = inline('<expression>', '<arg1>', '<arg2>', ...)
```

Since it is quite easy to forget to put dots (i.e., `.`) before the mathematical operations of multiplication (i.e., `*`), division (i.e., `/`), and exponentiation (i.e., `^`), the MATLAB command `vectorize` does it for you. To continue the first example,

```
≫  f = vectorize( inline('t^5 * exp(-2*t) * cos(3*t)', 't') )
```

is equivalent to the `f` defined above but does not require you to remember all the dots.

Warning: The `inline` command is obsolete and should not be used because it is very, very, very, very, very inefficient. It is mentioned here only because you might find it in "old" codes.

3.2. Passing Functions as Arguments to Commands

It is sometimes necessary to pass the name of a function into a MATLAB function or a function m-file created by the user. For example, as we discuss in section 4.1, we can plot the function $y = f(x)$ in the interval $[-5, +5]$ by

```
fplot(<function "name">, [-5 +5])
```

But how do we pass this "`name`"?

We put the `name` in quotes because we do not pass the name of the function, but its handle. If `f` has been defined by an anonymous function, then we enter

```
fplot(f, [-5 +5])
```

because `f` is a variable which we have already defined. If `fnc` is a MATLAB function or a user-defined function m-file, then it is not known in the MATLAB workspace so

```
fplot(fnc, [-5 +5])      % WRONG
```

will not work. Instead, we use

```
fplot(@fnc, [-5 +5])      % CORRECT
```

Note: There are a number of "older" ways to pass function names. For example,

```
≫  fplot('fnc', [-5 +5])
```

will also work. We can even pass a "simple" function by, for example,

```
≫  fplot('(x*sin(x) + sqrt(1 + cos(x)))/(x^2 + 1)', [-5 +5])
```

but don't do it! Instead, use anonymous functions.

3.3. Strings

Character strings are a very minor part of MATLAB, which is mainly designed to perform numerical calculations. However, they perform some very useful tasks which are worth discussing now.

It is often important to combine text and numbers on a plot. Since we discuss graphics in the next section, now is a good time to discuss how characters are stored in MATLAB variables. A string variable, such as

```
≫  str = 'And now for something completely different'
```

is simply a row vector with each character (actually its ASCII representation as shown on page 137) being a single element. MATLAB knows that this is a *text* variable, not a "regular" row vector, and so converts the numerical value in each element into the corresponding character when it is printed out. For example, to see what is actually contained in the vector `str` enter

```
≫  str + 0
```

or

```
≫  1*str
```

Character variables are handled the same as vectors or matrices. For example, to generate a new text variable which adds "– by Monty Python" to `str`, i.e., to concatenate the two strings, enter

```
≫  str2 = [str  ' - by Monty Python']
```

or

```
≫  str2 = [str, ' - by Monty Python']
```

(which might be easier to read). To convert a scalar variable, or even a vector or a matrix, to a character variable use the function `num2str`. For example, suppose you enter

```
≫  x = linspace(0, 2*pi, 100)'
≫  c1 = 2
≫  c2 = -3
≫  y = c1*sin(x) + c2*cos(x)
```

and want to put a description of the function into a variable. This can be done by

```
≫  s = [ num2str(c1), '*sin(x) + ', num2str(c2), '*cos(x)']
```

without explicitly having to enter the values of `c1` and `c2`.

A text variable can also contain more than one line if it is created as a matrix. For example,

```
≫  Str = ['And       '
          'now       '
          'for       '
          'something '
          'completely'
          'different ']
```

is four lines long. Since `str` is a matrix, each row must have the same number of elements and so we have to pad all but the longest row. (Using cell arrays, we will shortly show how to avoid this requirement that each row must have the same number of characters.)

Note: We do not usually enter matrices this way, i.e., one column per line. Instead, we simply use ";" to separate columns. However here we need to make sure that each row has EXACTLY the same number of characters — or else a fatal error message will be generated.

If desired, you can have more control over how data is stored in strings by using the `sprintf` command which behaves very similarly to the C commands `sprintf`, `fprintf`, and `printf`. It is also very similar to the `fprintf` command in MATLAB which is discussed in detail in Section 6. Note that the data can be displayed directly on the screen by using `disp`. That is, `sprintf(...)` generates a character string and `disp(sprintf(...))` displays it on the screen.

There also is a `str2num` command to convert a text variable to a number and `sscanf` to do the same with more control over how the data is read. (This is also very similar to the C command, as discussed in section 6.)

3.4. Advanced Topic: Cell Arrays and Structures

It is occasionally useful in MATLAB to have a single variable contain all the data which is related to a specific task — and this data might well consist of scalars, vectors and/or matrices, and text variables. One simple reason for this is that it is easier to pass all the data into and out of functions. A cell array generalizes the "standard" arrays which were discussed in the previous section. The elements of a "standard" array are numbers, either real or complex, whereas the elements of a cell array can be any data type. The primary difference between a cell array and a structure is that in a structure the elements are named rather than numbered. We consider this an advanced topic not because it is complicated, but because it is so seldom necessary.

A simple example of a cell array is

```
≫  C = {2+3i, 'go cells'; [1 2 3]', hilb(5) }
```

and the output is

```
C =

    [2.0000 + 3.0000i]    'go cells'
          [3x1 double]    [5x5 double]
```

The only difference between this and a "standard" array is that here curly braces, i.e., `{ ... }`, enclose the elements of the array rather than brackets, i.e., `[ ... ]`. Note that only the scalar and the text variable are shown explicitly. The other elements are only described. A second way to generate the same cell array is by

```
≫  C(1,1) = {2+3i }
≫  C(1,2) = {'go cells'}
≫  C(2,1) = {[1 2 3]'}
≫  C(2,2) = {hilb(5) }
```

and a third way is by

```
≫  C{1,1} = 2+3i
≫  C{1,2} = 'go cells'
≫  C{2,1} = [1 2 3]'
≫  C{2,2} = hilb(5)
```

It is important to understand that there is a difference between `C(i,j)` and `C{i,j}`. The former is the *cell* containing element in the (i, j)-th location whereas the latter is the element itself. For example,

```
≫  C(1,1)^5      % WRONG
```

returns an error message because a cell cannot be raised to a power whereas

```
≫  C{1,1}^5      % CORRECT
```

returns `1.2200e+02 - 5.9700e+02i`. All the contents of a cell can be displayed by using the `celldisp` function. In addition, just as a "standard" array can be preallocated by using the `zeros` function, a cell array can be preallocated by using the `cell` function. We will not discuss cells further except to state that cell array manipulation is very similar to "standard" array manipulation.

Warning: In MATLAB you can change a variable from a number to a string to a matrix by simply putting it on the left-hand side of equal signs. For example,

```
≫  c = pi
≫  c = 'And now for something completely different'
≫  c(5,3) = 17
```

redefines `c` twice without any difficulty. However, this cannot be done with cells. If you now try

```
≫  c{3} = hilb(5)
```

MATLAB will return with the error message

```
??? Cell contents assignment to a non-cell array object.
```

In order to use `c` as a cell (if has been previously used as a non-cell), you have to either clear it using `clear`, empty it using `[]`, or explicitly redefine it by using the `cell` function.

One particularly useful feature of cell arrays is that a number of text variables can be stored in one cell array. We previously defined a "standard" array of strings in `Str` on page 38 where each string had to have the same length. Using a cell array we can simply enter

```
≫  Str_cell = {'And'
               'now'
               'for'
               'something'
               'completely'
               'different'}
```

or

```
≫  Str_cell = {'And'; 'now'; 'for'; something'; 'completely'; 'different'}
```

and obtain the `i`-th row by

```
≫ Str_cell{i,:}
```

Structures can store different types of data similarly to cell arrays, but the data is stored by name, called *fields* , rather than by number. Structures are very similar to structures in C and C++ . The cell array we have been using can be written as a structure by

```
≫ Cs.scalar = 2+3i
≫ Cs.text = 'go cells'
≫ Cs.vector = [1 2 3]'
≫ Cs.matrix = hilb(5)
```

Typing

```
≫ Cs
```

returns

```
Cs =

    scalar: 2.0000 + 3.0000i
      text: 'go cells'
    vector: [3x1 double]
    matrix: [5x5 double]
```

The structure can also be created using one command by

```
≫ Cs = struct('scalar', 2+3i, 'text', 'go cells', ...
              'vector', [1 2 3]', 'matrix', hilb(5))
```

By the way, structures can themselves be vectors or matrices. For example,

```
≫ Cs(2) = struct('scalar', pi, 'text', 'structures rule', ...
                 'vector', ones(10,1), 'matrix', hilb(5)^2)
```

Now

```
≫ Cs
```

returns

```
Cs =

1x2 struct array with fields:
    scalar
    text
    vector
    matrix
```

Warning: As with cells, you cannot change a nonstructure variable to a structure variable. Instead, you have to either clear it using `clear`, empty it using `[]`, or explicitly redefine it by using the `struct` function.

We can also use function handles in cell elements and structures. For example, suppose you want to work with all six basic trig functions. They can be stored in a cell array by

```
≫ T = {@sin, @cos, @tan, @cot, @sec, @csc}
```

so that `T{2}(0) = 1`. They can also be stored in a structure by

```
≫ Tr.a = @sin; Tr.b = @cos; Tr.c = @tan; Tr.d = @cot; Tr.e = @sec; Tr.f = @csc;
```

so that `Tr.b(0) = 1`. By the way, we can even store anonymous functions in cell arrays and structures. For example,

```
≫ C = {@sin, @(x) exp(sin(x)), @(x) exp(exp(sin(x)))}
```

is allowed — but probably not very interesting.

Note: We cannot store function handles in standard matrices — we can only store numbers.

Other Data Types

`num2str(x)`	Converts a variable to a string. The argument can also be a vector or a matrix.
`str2num(str)`	Converts a string to a variable. The argument can also be a vector or a matrix string.
`sscanf`	Behaves very similarly to the C command in reading data from a file using any desired format. (See `fscanf` for more details.)
`sprintf`	Behaves very similarly to the C command in writing data to a string using any desired format. (See `fprintf` for more details.)
`cell`	Preallocate a cell array of a specific size.
`celldisp`	Display all the contents of a cell array.
`struct`	Create a structure.

4. Graphics

A very useful feature of MATLAB is its ability to generate high quality two- and three-dimensional plots using simple and flexible commands. All graphical images are generated in a "graphics window", which is completely separate from the "text window" in which MATLAB commands are typed. Thus, non-graphical and graphical commands can be completely intermixed.

Graphical images can be generated both from data calculated in MATLAB and from data which has been generated outside of MATLAB. In addition, these images can be output from MATLAB and printed on a wide variety of output devices, including color ink-jet printers and black-and-white and color laser printers.

There are a number of demonstrations of the graphical capabilities in MATLAB which are invoked by

```
≫ demo
```

Since the MATLAB commands which generate the plots are also shown, this demo makes it quite easy to generate your own graphics. You also can have very fine control over the appearance of the plots. We begin by considering only the basic commands; more advanced graphics commands are discussed in the next section.

Note: Most MATLAB commands which take vectors as arguments will accept either row or column vectors.

4.1. Two-Dimensional Graphics

The MATLAB command `plot` is used to constructing basic two-dimensional plots. For example, suppose you want to plot the functions $\mathtt{y1} = \sin \mathtt{x}$ and $\mathtt{y2} = e^{\cos \mathtt{x}}$ for $\mathtt{x} \in [0, 2\pi]$; also, you want to plot $\mathtt{y3} = \sin\big(\cos(\mathtt{x}^2 - \mathtt{x})\big)$ for $\mathtt{x} \in [0, 8]$. First, generate n data points on the curve by

```
≫ n = 100;
≫ x = 2*pi*[0:n-1]'/(n-1);
≫ y1 = sin(x);
≫ y2 = exp(cos(x));
≫ xx = 8*[0:n-1]/(n-1);
≫ y3 = sin( cos( xx.^2 - xx ) );
```

We plot these data points by

```
≫ plot(x, y1)
≫ plot(x, y2)
≫ plot(xx, y3)
```

Note that the axes are changed for every plot so that the curve just fits inside the axes. We can generate the x coordinates of the data points more easily by

```
>>  x = linspace(0, 2*pi, n);
>>  xx = linspace(0, 8, n);
```

The `linspace` command has two advantages over the colon operator:

(1) the endpoints of the axis and the number of points are entered directly as

```
>>  x = linspace(<first point>, <last point>, <number of points>)
```

so it is much harder to make a mistake; and

(2) round-off errors are minimalized so you are guaranteed that `x` has exactly `n` elements, and its first and last elements are exactly the values entered into the command.†

To put all the curves on one plot, type

```
>>  plot(x, y1, x, y2, xx, y3)
```

Each curve will be a different color — but this will not be visible on a black-and-white output device. Instead, you can change the type of lines by

```
>>  plot(x, y1, x, y2, '--', xx, y3, ':')
```

where "`--`" means a dashed line and "`:`" means a dotted line. (We discuss these symbols in detail in subsection 4.3.) In addition, you can use small asterisks to show the locations of the data points for the `y3` curve by

```
>>  plot(x, y1, x, y2, '--', xx, y3, ':*')
```

These strings are used to modify the color of the line, to put markers at the nodes, and to modify the type of line as shown in the table below. (As we discuss later in this section, the colors are defined by giving the intensities of the red, green, and blue components.)

Customizing Lines and Markers

Symbol	Color (R G B)
r	red (1 0 0)
g	green (0 1 0)
b	blue (0 0 1)
y	yellow (1 1 0)
m	magenta (1 0 1) (a deep purplish red)
c	cyan (0 1 1) (greenish blue)
w	white (1 1 1)
k	black (0 0 0)

Symbol	Line Style
-	solid line (default)
--	dashed line
:	dotted line
-.	dash-dot line

Marker	Description
+	plus sign
o	circle
*	asterisk
.	point
x	cross
s	square
d	diamond
^	upward pointing triangle
v	downward pointing triangle
>	right pointing triangle
<	left pointing triangle
p	pentagram
h	hexagram
(none)	no marker

†As we discussed previously, it is very unlikely (but it is possible) that round-off errors might cause the statement

```
>>  x = [0: 2*pi/(n-1): 2*pi]';
```

to return `n` − 1 elements rather than `n`. This is why we used the statement

```
>>  x = 2*pi*[0:n-1]'/(n-1);
```

above, which does not suffer from round-off errors because the colon operator is only applied to integers.

For example,

```
≫ plot(x, y1, 'r', x, y2, 'g--o', x, y3, 'mp')
```

plots three curves: the first is a red, solid line; the second is a a green, dashed line with circles at the data points; the third has magenta pentagrams at the data points but no line connecting the points.

We can also plot the first curve, and then add the second, and then the third by

```
≫ plot(x, y1)
≫ hold on
≫ plot(x, y2)
≫ plot(xx, y3)
≫ hold off
```

Note that the axes can change for every new curve. However, all the curves appear on the same plot.

Instead of putting a number of curves on one plot, you might want to put a number of curves individually in the graphics window. You can display m plots *vertically* and n plots *horizontally* in one graphics window by

```
≫ subplot(m, n, p)
```

This divides the graphics window into $m \times n$ rectangles and selects the p-th rectangle for the current plot. All the graphics commands work as before, but now apply only to this particular rectangle in the graphics window. You can "bounce" between these different rectangles by calling `subplot` repeatedly for different values of p.

Warning: If you are comparing a number of plots, it is important to have the endpoints of the axes be the same in all the plots. Otherwise your brain has to try to do the rescaling "on the fly" — which is very difficult. Of course, you frequently do not know how large the axes need to be until you have filled up the entire graphics window. The `axis` command (discussed below) can then be used to rescale *all* the plots.

In addition, you can also change the endpoints of the axes by

```
≫ axis([-1 10 -4 4])
```

The general form of this command is `axis([xmin xmax ymin ymax])`. If you only want to set some of the axes, set the other or others to $\pm$`Inf` ($-$`Inf` if it is the minimum value and $+$`Inf` if it is the maximum). Also, you can force the two axes to have the same scale by

```
≫ axis equal
```

or

```
≫ axis image
```

and to have the same length by

```
≫ axis square
```

To learn about all the options for these commands, use the `help` or `doc` command.

Note: The command `axis` is generally only in effect for one plot. Every new plot turns it off, so it must be called for every plot (unless `hold` is `on`).

The `plot` command generates linear axes. To generate logarithmic axes use `semilogx` for a logarithmic axis in `x` and a linear axis in `y`, `semilogy` for a linear axis in `x` and a logarithmic axis in `y`, and `loglog` for logarithmic axes in both `x` and `y`.

MATLAB has two different commands to plot a function directly rather than plotting a set of points.

Warning: *These commands do not always generate the correct curve (or curves) because they know nothing of the actual behavior of the function.* They can have problems with sharp peaks and asymptotes and other "strange behavior". We will show some examples shortly.

The first command we discuss is `fplot`, which can be executing by simply entering

```
≫ fplot(<function handle>, <limits>)
```

where the function is usually generated as an anonymous function or a MATLAB function or a user generated function m-file (as described in section 8.3). The limits are either

```
[xmin  xmax]
```

in which case the y-axis just encloses the curve or

```
[xmin  xmax  ymin  ymax]
```

in which case you are also specifying the endpoints on the y-axis.

Note: Recall in section 3.2 we discussed how to pass a function as an argument.

This function uses adaptive step control to generate as many data points as it considers necessary to plot the function accurately. You can also store the data points calculated by

```
≫  [x, y] = fplot(<function handle>, <limits>)
```

rather than having the function plotted directly. You then have complete control over how to plot the curve using the `plot` function.

The other command which can plot a function is `ezplot`, which is more general than `fplot`. To plot a function on the interval $[-2\pi, +2\pi]$ enter

```
≫  ezplot(<function handle>)
```

To include limits (as with `fplot`) enter

```
≫  ezplot(<function handle>, <limits>)
```

In addition, a parametrically defined function can be plotted by

```
≫  ezplot(<fnc 1>, <fnc 2>, <limits>)
```

Finally, this command can also plot an implicitly defined function, i.e., $f(x, y) = 0$, by

```
≫  ezplot(<2D fnc>, <limits>)
```

For example,

```
≫  f = @(x, y) (x^2 + y^2)^2 - (x^2 - y^2);
≫  ezplot(f)
```

plots the lemniscate of Bernoulli (basically an "∞" symbol).

Warning: Be particularly careful when plotting implicit functions because they can be REALLY NASTY and occasionally `ezplot` may not get it right.

There is an important difference between

```
≫  fplot(f, [-5 5])
```

and

```
≫  ezplot(f, [-5 5])
```

In the former `f(x)` is only evaluated for scalar values of `x`, while in the latter `f(x)` is evaluated for vector values of `x`. Thus, when using `ezplot` care must be taken if `f` is evaluated in a function m-file. If `f(x)` cannot be evaluated for vector values, the error message

```
Warning: Function failed to evaluate on array inputs; vectorizing the function may
speed up its evaluation and avoid the need to loop over array elements.
```

will be generated

`fplot` and `ezplot` do not always generate exactly the same curves. For example, in

```
≫  f = @(x) log(x) + 1;
≫  fplot(f, [-2*pi 2*pi])
≫  ezplot(f)
```

`fplot` generates a spurious plot for $x \in [-2 * pi, 0)$ where it plots the real part of $\log x$ while `ezplot` only plots the function for $x \in (0, 2\pi]$. Also, in

```
≫  f = @(x) x ./ (x^2 + 0.01);
≫  fplot(f, [-2*pi +2*pi])
≫  ezplot(f)
```

the vertical axes are different and `ezplot` is missing part of the curve. Finally, in

```
f = @(x) x^3/(x^2 + 3*x - 10);
ezplot(f, [-10 +10])
```

the function blows up at $x = -5$ and 2 and part of the curve for $x \in (-5, 2)$ is not shown.

Polar plots can also be generated by the `polar` command. There is also an "easy" command for generating polar plots, namely `ezpolar`.

Since you often want to label the axes and put a title on the plot, there are specific commands for each of these. Entering

```
≫  xlabel(<string>)
≫  ylabel(<string>)
≫  title(<string>)
```

put labels on the x-axis, on the y-axis, and on top of the plot, respectively. Note that a title can contain more than one line as was discussed in section 3.

For example, typing `title(t)` where

```
t = ['The Dead     '
     'Parrot Sketch']
```

or

```
t = {'The Dead'
     'Parrot Sketch'}
```

or

```
t = {'The Dead'; 'Parrot Sketch'}
```

results in a two-line title. The former uses a "standard" array and so requires all the rows to have the same number of columns, whereas the latter uses a cell array and so each row can have a different length.

There are also a number of ways to plot data, in addition to the commands discussed above. The two we discuss here are histograms and error bars. To plot a histogram of the data stored in the vector `x`, type

```
≫ hist(x)
```

which draws ten bins between the minimum and maximum values of the elements in `x`. For example, to see how uniform the distribution of random numbers generated by `rand` is, type

```
≫ x = rand(100000, 1);
≫ hist(x)
```

To draw a histogram with a different number of bins, type

```
≫ hist(x, <number of bins>)
```

and to draw a histogram with the centers of the bins given by the vector `c`, type

```
≫ hist(x, c)
```

As another example, to see how uniform the distribution of Gaussian random numbers generated by `randn` is, type

```
≫ x = randn(1000, 1);
≫ hist(x)
```

Clearly you need more random numbers to get a "good" histogram — but, at the moment, we are interested in a different point. If you rerun this command a number of times, you will find that the endpoints of the histogram fluctuate. To avoid this "instability", fix the endpoints of the histogram by

```
≫ xmax = 4;
≫ nrbin = 20;
≫ nrdata = 1000;
≫ c = xmax*[ -1+1/nrbin : 2/nrbin : 1-1/nrbin ];
≫ x = randn(nrdata, 1);
≫ hist(x, c)
```

Note that `c` contains the *midpoints* of each bin and not their endpoints. Another way to calculate `c`, which might be clearer, is

```
≫ c = linspace(-xmax+xmax/nrbin, xmax-xmax/nrbin, nrbin);
```

Of course, to get a "good" histogram you should increase `nrbin`, say to 100, and `nrdata`, say to 100,000. If you now rerun this code you will see a much smoother histogram.

We have already seen how to plot the vector `x` vs. the vector `y` by using the `plot` command. If, additionally, you have an error bar of size e_i for each point y_i, you can plot the curve connecting the data points along with the error bars by

```
≫ errorbar(x, y, e)
```

Sometimes the error bars are not symmetric about the y values. In this case, you need vectors `l` and `u` where at x_i the error bars extend from $y_i - l_i$ to $y_i + u_i$. This is done by

```
≫ errorbar(x, y, l, u)
```

Note: All the elements of `l` and `u` are non-negative.

Data can also be entered into MATLAB from a separate data file. For example,

```
≫ M = csvread('<file name>')
```

reads in data from a file one row per line of input. The numbers in each line must be separated by commas. The data can then be plotted as desired. The command `csvwrite` writes the elements of a matrix into a file using the same format. (If desired, you can have much more control over how data is input and

output by using the `fscanf` and `fprintf` commands, which are similar to their C counterparts. These commands are discussed in detail in section 6.)

The `load` command can also be used to read a matrix into MATLAB from a separate data file. The data must be stored in the data file one row per line. The difference between this command and `csvread` is that the numbers can be separated by commas *or by spaces*. The matrix is input by entering

```
≫ load('<file name>')
```

and it is stored in the matrix named `<file name-no extension>` (i.e., drop the extension, if any, in the file name).†

Graphics can also be easily printed from within MATLAB. You can print directly from the graphics window by going into the "File" menu item. If desired, the plot can be sent to a file rather than to an output device. You can also store the plot in the text window by using the command `print`. There are an innumerable number of printer specific formats that can be used. (See `help print` or `doc print` for details.) If you want to save a file in postscript, you can save it in black-and-white postscript by

```
≫ print -deps <file name b&w>
```

or in color postscript by

```
≫ print -depsc <file name color>
```

There is a minor, but important, difference between these two files if they are printed on a black-and-white laser printer. When the black-and-white file is printed, all the non-white colors in the plot become black. However, when the color file is printed, the colors are converted to different grayscales. This makes it possible to differentiate lines and/or regions.

Note: The `print` command is also a MATLAB function where it is called by

```
≫ print('-deps', '<file name b&w>')
```

The advantage of using the `print` function is that the arguments can be variables. An oversimplified example is

```
≫ device = '-deps'
≫ file = '<file name b&w>'
≫ print(device, file)
```

It is oversimplifed because there is no need to use three lines when one will do. However, if many plots are to be printed then the print device can be changed once rather than in every print command. Also, if you are printing many plots then you can easily modify the file names as in

```
≫ i = 1
≫ file = ['fiddledum', num2str(i), '.eps']
≫ print(device, file)
≫ ...
≫ i = i + 1
≫ file = ['fiddlededum', num2str(i), '.eps']
≫ print(device, file)
≫ ...
```

---

†The `load` command is a little tricky because it can read in files generated both by MATLAB (using the `save` command) and by the user. For example,

```
≫ save allvariables;
≫ clear
```

or

```
≫ save allvariables.mat;
≫ clear
```

saves all the variables to the file `allvariables.mat` in *binary format* and then deletes all the variables. Entering

```
≫ load allvariables
```

or

```
≫ load allvariables.mat
```

loads all these variables back into MATLAB using the binary format. On the other hand, if you create a file, say `mymatrix.dat`, containing the elements of a matrix and enter it into MATLAB using

```
≫ load('mymatrix.dat')
```

you obtain a new matrix, called `mymatrix`, which contains these elements. Thus, the `load` command determines how to read a file depending on the extension.

Input-Output

`csvread('<file name>')`	Reads data into MATLAB from the named file, one row per line of input; the numbers in each line must be separated by commas.
`load('<file name>')`	Reads data into MATLAB from the named file, one row per line of input; the numbers in each line can be separated by spaces or commas. The name of the resulting matrix is `<file name>`.
`csvwrite('<file name>', A)`	Writes out the elements of a matrix to the named file using the same format as `csvread`.
`print`	Prints a plot or saves it in a file using various printer specific formats. For example, `print -deps <file name>` saves the plot in the file using encapsulated PostScript (so it can be plotted on a PostScript laser printer).

Two-Dimensional Graphics

`plot(x, y)`	Plots the data points in Cartesian coordinates. The general form of this command is `plot(x1, y1, s1, x2, y2, s2, ...)` where `s1`, `s2`, ... are optional character strings containing information about the type of line, mark, and color to be used. Some additional arguments that can be used: `plot(x)` plots `x` vs. the index number of the elements. `plot(Y)` plots each column of `Y` vs. the index number of the elements. `plot(x,Y)` plots each column of `Y` vs. `x`. If `z` is complex, `plot(z)` plots the imaginary part of `z` vs. the real part.
`semilogx`	The same as `plot` but the x axis is logarithmic.
`semilogy`	The same as `plot` but the y axis is logarithmic.
`loglog`	The same as `plot` but both axes are logarithmic.
`fplot(<function handle>, <limits>)`	Plots the specified function within the limits given. The limits can be `[xmin xmax]` or `[xmin xmax ymin ymax]`.

`ezplot(<function handle>)`
`ezplot(<fnc 1>, <fnc 2>)`
`ezplot(<2D fnc>)`
	Generates an "easy" plot (similar to `fplot`) given the function $f(x)$. It can also plot a parametric function, i.e., $\big(x(t), y(t)\big)$, or an implicit function, i.e., $f(x, y) = 0$. Limits can also be specified if desired.

`polar(r, theta)`	Plots the data points in polar coordinates.
`ezpolar(<function handle>)`	Generate an "easy" polar plot of $r = <$ `functionname` $>(\theta)$.
`xlabel(<string>)`	Puts a label on the x-axis.
`ylabel(<string>)`	Puts a label on the y-axis.
`title(<string>)`	Puts a title on the top of the plot.
`axis`	Controls the scaling and the appearance of the axes. `axis equal` and `axis([xmin xmax ymin ymax])` are two common uses of this command.
`hold`	Holds the current plot (`hold on`) or release the current plot (`hold off`).
`linspace(a, b, n)`	Generates `n` equally-spaced points between `a` and `b` (inclusive).
`hist(x)`	Plots a histogram of the data in a vector using 10 bins. `hist(x, <number of bins>)` changes the number of bins. `hist(x, c)` lets you choose the midpoint of each bin.
`errorbar(x, y, e)` `errorbar(x, y, l, u)`	The first plots the data points `x` vs. `y` with error bars given by `e`. The second plots error bars which need not be symmetric about `y`.
`subplot(m, n, p)`	Divides the graphics window into $m \times n$ rectangles and selects the p-th rectangle for the current plot.

4.2. Three-Dimensional Graphics

The MATLAB command `plot3` plots curves in three-dimensions. For example, to generate a helix

enter

```
≫  t = linspace(0, 20*pi, 1000);
≫  c = cos(t);
≫  s = sin(t);
≫  plot3(c, s, t)
```

and to generate a conical helix enter

```
≫  t = [0 : pi/100 : 20*pi];
≫  c = cos(t);
≫  s = sin(t);
≫  plot3(t.*c, t.*s, t)
```

Also, you can put a label on the z-axis by using the `zlabel` command. There is also an "easy" `plot3` command. It generates the curve $\big(x(t), y(t), z(t)\big)$ for $t \in (0, 2\pi)$ by

```
≫  ezplot3(x, y, z)
```

if `x`, `y`, and `z` have been defined using anonymous functions. Again, you change the domain of t by specifying the additional argument `[tmin, tmax]`.

MATLAB also plots surfaces `z = f(x,y)` in three-dimensions with the hidden surfaces removed. First, the underlying mesh must be created. The easiest way is to use the command `meshgrid`. This combines a discretization of the x axis, i.e., $\{ x_1, x_2, \ldots, x_m \}$, and the y axis, i.e., $\{ y_1, y_2, \ldots, y_n \}$, into the rectangular mesh $\big\{ (x_i, y_j) \bigm| i = 1, 2, \ldots, m\,,\ j = 1, 2, \ldots, n \big\}$ in the x–y plane. The function `f` can then be evaluated at these mesh nodes. For example,

```
≫  x = [-3:0.1:3]';
≫  y = [-2:0.1:2]';
≫  [X, Y] = meshgrid(x, y);
≫  F = (X + Y).*exp( -X.*X - 2*Y.*Y );
≫  mesh(X, Y, F)
```

generates a colored, wire-frame surface whereas

```
≫  surf(X, Y, F)
```

generates a colored, filled-in surface. We discuss how to change the colors, and even how to use the colors as another variable, in the next section.

You can change the view of a three-dimensional plot by using the `view` command. This command is called in either of two ways:

- First, you can give the angles from the origin of the plot to your eye by
  `view(<azimuth>, <elevation>)`
  where the azimuth is the angle in degrees in the x–y plane measured from the $-y$ axis (so 0° is the $-y$ axis, 90° is the x axis, 180° is the y axis, etc.) and the elevation is the angle in degrees up from the x–y plane toward the $+z$ axis (so 0° is in the x–y plane, 90° is on the $+z$ axis, etc.).
- Second, you can give the coordinates of a vector pointing from the origin of the plot to your eye by `view([x y z])`, where you enter the coordinates of the vector.

If you type

```
≫  contour(X, Y, F)
```

you will see contour plots of the surface. That is, you will be looking down the z axis at curves which represent lines of constant elevation (i.e., constant z values). If we type

```
≫  contour3(X, Y, F)
```

you will see contour plots of the surface in three dimensions. You can again change your view of these curves by using the `view` command.

If you do not want to bother with generating the mesh explicitly, you can generate "easy" plots by `ezcontour`, `ezcontour3`, `ezmesh`, and `ezsurf`.

Three-Dimensional Graphics

`plot3(x, y, z)`	Plots the data points in Cartesian coordinates. The general form of this command is `plot(x1, y1, z1, s1, x2, y2, z2, s2, ...)` where `s1`, `s2`, ... are optional character strings containing information about the type of line, mark, and color to be used.
`ezplot3(<fnc 1>, <fnc 2>, <fnc 3>)`	Generates an "easy" plot in 3-D.
`mesh(X, Y, Z)`	Plots a 3-D surface using a wire mesh.
`ezmesh(<2D fnc>)`	Generates an "easy" 3-D surface using a wire mesh.
`surf(X, Y, Z)`	Plots a 3-D filled-in surface.
`ezsurf(<2D fnc>)`	Generates an "easy" 3-D filled-in surface.
`view`	Changes the viewpoint of a 3-D surface plot by `view(<azimuth>, <elevation>)` or `view([x y z])`.
`meshgrid(x, y)`	Generates a 2-D grid given the x-coordinates and the y-coordinates of the mesh lines.
`zlabel(<string>)`	Puts a label on the z-axis.
`axis`	Controls the scaling and the appearance of the axes. `axis([xmin xmax ymin ymax zmin zmax])` changes the endpoints of the axes.
`contour(X, Y, Z)`	Plots a contour looking down the z axis.
`ezcontour(<2D fnc>)`	Generates an "easy" contour looking down the z axis.
`contour3(X, Y, Z)`	Plots a contour in 3-D.
`ezcontour3(<2D fnc>)`	Generates an "easy" contour in 3-D.
`subplot(m, n, p)`	Remember than `subplot` can also be called in 3-D to put a number of plots in one graphics window.

4.3. Advanced Graphics Techniques: Commands

In the previous subsections we have discussed how to use "simple" graphics commands to generate basic plots. MATLAB can also do much more "interesting" graphics, and even publication quality graphics. Here we discuss some of the more useful advanced features. We divide the topic into two subsections: the first discusses the commands themselves and the second discusses how to change some of the properties of these commands.

Note: The demonstration program shows many more of the graphics capabilities of MATLAB. Enter

```
demo
```

and then in `Help Navigator` click on `Graphics`.

It is possible to obtain the current position of the cursor within a plot by using the `ginput` command. For example, to collect any number of points enter

```
≫  [x, y] = ginput
```

Each position is entered by pressing any mouse button or any key on the keyboard except for the carriage return (or enter) key. To terminate this command press the return key. To enter exactly n positions, use

```
≫  [x, y] = ginput(n)
```

You can terminate the positions at any time by using the return key. Finally, to determine which mouse button or key was entered, use

```
≫  [x, y, button] = ginput(n)
```

The vector `button` contains integers specifying which mouse button (1 = left, 2 = center, and 3 = right) or key (its ASCII representation) was pressed.

Labels can also be added to a plot. Text can be placed anywhere inside the plot using

```
≫  text(xpt, ypt, <string>)
```

The text is placed at the point `(xpt,ypt)` in units of the current plot. The default is to put the center of the left-hand edge of the text at this point. You can also use the mouse to place text inside the plot using

```
≫ gtext(<string>)
```

The text is fixed by depressing a mouse button or any key.

If more than one curve appears on a plot, you might want to label each curve. This can be done directly using the `text` or `gtext` command. Alternatively, a legend can be put on the plot by

```
≫ legend(<string1>, <string2>, ... )
```

Each string appears on a different line preceded by the type of line (so you should use as many strings as there are curves). The entire legend is put into a box and it can be moved within the plot by using the left mouse button.

TEX commands can be used in these strings to modify the appearance of the text. The results are similar, but not quite identical, to the appearance of the text from the TEX program (so do some experimenting). Most of the "common" TEX commands can be used, including Greek letters; also, "`^`" and "`_`" are used for superscripts and subscripts. For example, the `x`-axis can be labelled α^2 and the `y`-axis $\int_0^\alpha f(x)\,dx$ by

```
≫ xlabel('\alpha^2')
≫ ylabel('\int_0^\pi\betaf(x) dx')
```

To see the complete list of TEX commands, enter

```
≫ doc text
```

and then click on the highlighted word `String`.

Note: For you TeXers note the funny control sequence "`\betaf(x)`" which generates $\beta f(x)$. If you would have typed "`\beta f(x)`" you would have obtained $\beta\ f(x)$ because MATLAB preserves spaces. If typing "`\betaf(x)`" sets your teeth on edge, try "`\beta{}f(x)`" instead.

It is often essential for the title to include important information about the plot (which would, otherwise, have to be written down "somewhere" and connected to this specific plot). For example, suppose you enter

```
≫ x = linspace(0, 2*pi, 100)
≫ c1 = 2
≫ c2 = -3
≫ p1 = 1
≫ p2 = 3
≫ y = c1*sin(x).^p1 + c2*cos(x).^p2
≫ plot(x, y)
```

and you want to "play around" with the two coefficients to obtain the most "pleasing" plot. Then you probably should have the title include a definition of the function — and you should not have to modify the title by hand every time you change the coefficients. This can be done by

```
≫ t = [num2str(c1), '*sin^', num2str(p1), '(x) + ', num2str(c2), ...
       '*cos^', num2str(p2), '(x)']
≫ title(t)
```

where we use the text variable `t`, rather than putting the string directly into `title`, simply to make the example easier to read. There is now a permanent record of the function which generated the curve. (Alright, this isn't a *great* example, but it's better than nothing.)

You can also put plots in a new graphics window by entering

```
≫ figure
```

where the figures are numbered consecutively starting with one (and appear at the top of the window). Or enter

```
≫ figure(n)
```

and the figure will have the specific number `n`. This creates a new window and makes it the current target for graphics commands. You can "bounce" between graphics windows by entering

```
≫ figure(n)
```

where `n` is the number of the graphics window you want to make current. New plots will now appear in this figure. In this way much more information can be generated and viewed on the computer terminal.

Occasionally, it is useful to clear a figure. For example, suppose you divide a window into a 2×2 array of plotting regions and use `subplot` to put a plot into each region; you then save the figure into a file. Next, you only want to put plots into two of these four regions. The difficulty is that the other two regions will still contain the previous plots. You can avoid this difficulty by clearing the figure using

```
≫ clf
```

which clears the current figure. You can clear a particular figure by `clf(n)`.

All the above MATLAB commands can be used for 3-D graphics except for `gtext`. The `text` command is the same as described above except that the position of the text requires three coordinates, i.e.,

```
≫ text(x, y, z, <string>)
```

The new command

```
≫ zlabel(<string>)
```

labels the z-axis.

As we discussed in the previous subsection, the `mesh` and `surf` commands allow us to plot a surface in three dimensions where the colors on the surface represent its height. We can add a rectangle which contains the correspondence between the color and the height of the surface by adding

```
≫ colorbar
```

We can also let the colors represent a separate quantity `C`, which is also defined at each mesh point, by changing the command to

```
≫ mesh(X, Y, F, C)
```

or

```
≫ surf(X, Y, F, C)
```

Each graphics window has a separate color map associated with it. This color map is simply an $n\times 3$ matrix, where each element is a real number between 0 and 1 inclusive. In each row the first column gives the intensity of the color red, the second column green, and the third column blue; these are called the *RGB components* of a color. For example, we show the RGB components of cyan, magenta, yellow, red, blue, green, white, and black in the table "Customizing Lines and Markers" at the beginning of this section; for further information, enter `doc colorspec`. The value input to this color map is the row representing the desired color.

For `mesh` or `surf` the value of `F` (or of `C` if there is a fourth argument) is linearly rescaled so its minimum value is 1 and its maximum value is n. To see the current color map, enter

```
≫ colormap
```

To change the color map, enter

```
≫ colormap(<color map>)
```

where `<color map>` can be an explicit $n\times 3$ matrix of the desired RGB components or it can be a string containing the name of an existing color map. The existing color maps can be found by typing

```
≫ doc graph3d
```

A useful color map for outputting to laser printers is `'gray'`. In this colormap all three components of each row have the same value so that the colors change gradually from black (RGB components [0 0 0]) through gray [.5 .5 .5]) to white [1 1 1]).

MATLAB can also plot a two-dimensional image (i.e., a picture) which is represented by a matrix $\texttt{X} \in \mathbb{R}^{m\times n}$. The (i,j)-th element of `X` specifies the color to use in the current color map. This color appear in the (i,j)-th rectilinear patch in the plot. For example, to display the color image of a clown enter

```
≫ load clown
≫ image(X);
≫ colormap(map)
```

The `image` command inputs the matrix `X` and the colormap `map` from `clown.mat`. Then the image is displayed using the new color map. Similarly,

```
≫ load earth
≫ image(X);
≫ colormap(map);
≫ axis image
```

displays an image of the earth. (The `axis` command forces the earth to be round, rather than elliptical.) (In the demonstration program, after clicking on "Visualization" double-click on "Image colormaps" to see the images which you can access in MATLAB and the existing color maps.)

MATLAB can also fill-in two-dimensional polygons using `fill` or three-dimensional polygons using `fill3`. For example, to draw a red circle surrounding a yellow square, enter

```
≫ t = linspace(0, 2*pi, 100);
≫ s = 0.5;
≫ xsquare = [-s s s -s]';
≫ ysquare = [-s -s s s]';
≫ fill(cos(t), sin(t), 'r', xsquare, ysquare, 'y')
≫ axis equal;
```

To obtain a more interesting pattern replace the above fill command by

```
≫ colormap('hsv');
≫ fill(cos(t), sin(t), [1:100], xsquare, ysquare, [100:10:130])
```

Rather than entering polygons sequentially in the argument list, you can enter

```
≫ fill(X, Y, <color>)
```

where each column of `X` and `Y` contain the endpoints of a different polygon. Of course, in this case the number of endpoints of each polygon must be the same, by padding if necessary. For example, to draw a cube with all the faces having a different solid color, input the matrices

$$\mathtt{X} = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}, \mathtt{Y} = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}, \mathtt{Z} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \end{pmatrix}.$$

Then enter

```
≫ fill3(X, Y, Z, [1:6])
≫ axis equal
```

Change your orientation using `view` to see all six faces. Read the documentation on `fill` and `fill3` for more details.

Advanced Graphics Features: Plots

`clf`	Clear a figure (i.e., delete everything in the figure)
`colorbar`	Adds a color bar showing the correspondence between the value and the color.
`colormap`	Determines the current color map or choose a new one.
`demo`	Runs demonstrations of many of the capabilities of MATLAB.
`figure`	Creates a new graphics window and makes it the current target. `figure(n)` makes the n-th graphics window the current target.
`fill(x, y, <color>)`	Fills one or more polygons with the color or colors specified by the vector or string `<color>`.
`fill3(x, y, z, <color>)`	Fills one or more 3D polygons with the color or colors specified by the vector or string `<color>`.
`image`	Plots a two-dimensional image.

Advanced Graphics Features: Text and Positioning

`ginput`	Obtains the current cursor position.
`text(x, y, <string>)` `text(x, y, z, <string>)`	Adds the text to the location given in the units of the current plot.
`gtext(<string>)`	Places the text at the point given by the mouse.
`legend(<string 1>, ...)`	Places a legend on the plot using the strings as labels for each type of line used. The legend can be moved by using the mouse.

4.4. Advanced Graphics Techniques: Handles and Properties

In this subsection we briefly discuss *handle graphics*. This is a collection of low-level graphics commands which do the actual work of generating graphics. In the previous parts of this section we have mainly discussed "high-level" graphics commands which allow us to create useful and high quality graphical images very easily. The low-level commands allow us to customize these graphical images, but at the cost of having to get much more involved in how graphical images are actually created. This subsection will be quite short because we do not want to get bogged down in this complicated subject. Instead, we will only discuss a few of — what we consider to be — the more useful customizations.

In handle graphics we consider every component of a graphical image to be an *object*, such as a subplot, an axis, a piece of text, a line, a surface, etc. Each object has *properties* and we customize an object by changing its properties. Of course, we have to be able to refer to a particular object to change its properties, and a *handle* is the unique identifier which refers to a particular object. (Each handle is a unique floating-point number.)

We will use a small number of examples to explain handle graphics. There are many properties of the text that can be changed in the `text` command by

```
>> text(xpt, ypt, <string>, '<Prop 1>', <Value 1>, '<Prop 2>', <Value 2>, ... )
```

or

```
>> h = text(xpt, ypt, <string>);
>> set(h, '<Prop 1>', <Value 1>, '<Prop 2>', <Value 2>, ... )
```

where `<Prop ?>` is the name of one of the properties for the text object and `<Value ?>` is one of the allowed values. (We show some names and values in the following table.) We have shown two ways to customize the properties. In the former all the properties are set in the `text` command. In the latter the text command creates an object, using its default properties, with handle `h`. The `set` command then changes some of the properties of the object whose handle is `h`. For example, entering

```
>> set(h, 'Color', 'r', 'FontSize', 16, 'Rotation', 90)
```

results in a large, red text which is rotated 90°. You can also change the default properties for `gtext`, `xlabel`, `ylabel`, `zlabel`, and `title`.

Text Properties

`Clipping`	`on` — (default) Any portion of the text that extends outside the axes rectangle is clipped `off` — No clipping is done.
`Color`	A three-element vector specifying a color in terms of its red, blue, and green components, or a string of the predefined colors.
`FontName`	The name of the font to use. (The default is `Helvetica`.)
`FontSize`	The font point size. (The default is 10 point.)
`HorizontalAlignment`	`left` — (default) Text is left-justified `center` — Text is centered. `right` — Text is right justified.
`Rotation`	The text orientation. The property value is the angle in degrees.
`VerticalAlignment`	`top` — The top of the text rectangle is at the point. `cap` — The top of a capital letter is at the point. `center` — (default) The text is centered vertically at the point. `baseline` — The baseline of the text is placed at the point. `bottom` — The bottom of the text rectangle is placed at the point.

The more common way of customizing parameters is by using the `set` command. The two functions `get` and `set` are used to obtain the value of one parameter and to set one or more parameters. For example, to get the font which is presenting being used enter

```
≫  s = get(h, 'FontName')
```

and the string `s` contains the name of the font. The two arguments to `get` are the handle of the object desired and the name of the property.

There are two other commands which can obtain a handle:

```
≫  hf = gcf
```

returns the handle of the current figure and

```
≫  ha = gca
```

returns the handle of the current axes in the current figure. There is one case where we frequently use handle graphics. When a figure is printed, the graphical images do not fill the entire page. The default size is approximately 6.5 inches wide and 5.5 inches high. When we want to use the full size of a sheet of paper we use

```
≫  figure('PositionPaper', [0 0 8.5 11])
```

or

```
≫  figure(n)
≫  set(gcf, 'PositionPaper', [0 0 8.5 11])
```

since the default units for this property are inches. This sets the graphical images to use the full paper size (the position is given as `[left bottom width height]`) with a one inch border. This is frequently useful if `subplot` is being used to put a number of plots on a page.

Finally, if `subplot` is being used it is sometimes useful to put a title on the entire page, not just in each subplot. This can be done by

```
≫  axes_handle = axes('Position', [0 0 1 0.95], 'Visible', 'off');
≫  title_handle = get(axes_handle, 'title');
≫  set(title_handle, 'String', <title>, 'Visible', 'on');
```

The first line specifies a rectangle for the axes in normalized units (so that (`[left bottom width height] = [0 0 1 1]` is the full size of the figure). The axes are invisible because they are only being created so that a title can go on top. The second line gets the handle for the title object of the new axes. The third line puts `<title>` into the title object and makes it visible.

Advanced Graphics Features: Properties

`get(<handle>, '<Prop>')`	Return the current value of the property of the object with this handle.
`set(<handle>, '<Prop 1>', <Value 1>, ...)`	Set the property, or properties, of the object with this handle.
`gca`	The current axes handle.
`gcf`	The current figure handle.

4.5. Be Able To Do

After reading this section you should be able to do the following exercises. The answers are given on page 135.

1. Plot e^x and one of its Taylor series approximations.
   (a) Begin by plotting e^x for $x \in [-1, +1]$.
   (b) Then plot

$$p_3(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!}$$

   on the same graph.

(c) Also plot the difference between e^x and this cubic polynomial, i.e., $e^x - p_3(x)$ on the same graph.
(d) Next, generate a new graph containing all three curves by using only *one* `plot` command, force the axes to be to the same scale, and let all three curves have different colors. Put labels on the x and y axes and a silly title on the entire plot.
(e) The above plot is not very instructive because $e^x - p_3(x)$ is much smaller than either e^x or $p_3(x)$. Instead, use two plots. The first plot contains e^x and $p_3(x)$ and the second plot, which is immediately below the first, contains $e^x - p_3(x)$. These two plots should fill an entire sheet of paper.

2. Consider the function

$$f(x, y) = (x^2 + 4y^2)\sin(2\pi x)\sin(2\pi y)\,.$$

(a) Plot this function for $x, y \in [-2, +2]$.
Note: Make sure you use the "$.*$" operator in front of each sine term. What does the surface look like if you don't?
(b) This surface has high peaks which interfere with your view of the surface. Change your viewpoint so you are looking down at the surface at such an angle that the peaks do not block your view of the central valley.
Note: There are an infinite number of answers to this part.

5. Solving Linear Systems of Equations

One of the basic uses of MATLAB is to solve the linear system

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
\vdots \qquad\qquad\qquad\qquad\quad &\vdots \\
a_{j1}x_1 + a_{j2}x_2 + \cdots + a_{jn}x_n &= b_j \\
\vdots \qquad\qquad\qquad\qquad\quad &\vdots \\
a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m\,,
\end{aligned}$$

or the equivalent matrix equation

$$\mathtt{Ax} = \mathtt{b}\,.$$

Note that there are m equations in n unknowns so that there may be zero solutions to this linear system, one solution, or an infinite number of solutions. We will discuss the case where $m \neq n$ in detail in subsection 5.3. Here we concentrate on $m = n$.

5.1. Square Linear Systems

As we discussed previously, when $m = n$ the MATLAB operation

```
≫  x = A\b
```

calculates the unique solution `x` by Gaussian elimination when `A` is nonsingular. We can also solve it by

```
≫  x = linsolve(A, b)
```

The advantage of using `linsolve` is that it can be much faster when `A` has a particular property. The third argument to `linsolve` gives the particular property. For our purposes the most important properties are lower triangular, upper triangular, symmetric, and positive definite. Enter

```
≫  x = linspace(A, b, prop)
```

where `prop` is a **logical** structure with the following elements:

`LT`	– the matrix is lower triangular,
`UT`	– the matrix is upper triangular,
`SYM`	– the matrix is symmetric triangular, and
`POSDEF`	– the matrix is positive definite.

Normally, all the elements are false; set the property you want to true by

```
≫  prop.??? = true
```

where `???` is one of the above properties. To turn this property back off, enter

```
≫  prop.??? = false
```

(We discuss logical variables in section 8.1.) If `A` has one (or more) of these properties, it can be solved much faster than using `x = A\b`.

When `A` is singular there are either zero solutions or an infinite number of solutions to this equation and a different approach is needed. The appropriate MATLAB command system is now `rref`. It begins by applying Gaussian elimination to the linear system of equations. However, it doesn't stop there; it continues until it has zeroed out all the elements it can, both above the main diagonal as well as below it. When done, the linear system is in *reduced row echelon form*:

- The first nonzero coefficient in each linear equation is a 1 (but a linear equation can be simply $0 = 0$ in which case it has no nonzero coefficient).
- The first nonzero term in a particular linear equation occurs later than in any previous equation. That is, if the first nonzero term in the j-th equation is x_{k_j} and in the j+1-st equation is $x_{k_{j+1}}$, then $k_{j+1} > k_j$.

To use `rref`, the linear system must be written in *augmented matrix form*, i.e.,

$$\begin{array}{cccc|c} x_1 & x_2 & \cdots & x_n & = & \text{rhs} \\ \left(\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & b_m \end{array}\right) \end{array}.$$

Warning: **It is very important to realize that an augmented matrix is <u>not</u> a matrix** (because the operations we apply to augmented matrices are not the operations we apply to matrices). It is simply a linear system of equations written in shorthand: the first column is the coefficients of the x_1 term, the second column is the coefficients of the x_2 term, etc., and the last column is the coefficients on the right-hand side. The vertical line between the last two columns represents the equal sign. Normally, an augmented matrix is written without explicitly writing the header information; however, the vertical line representing the equal sign should be included to explicitly indicate that this is an augmented matrix.

`rref` operates on this augmented matrix to make as many of the elements as possible zero by using allowed operations on linear equations — these operations are not allowed on matrices, but only on linear systems of equations. The result is an augmented matrix which, when written back out as a linear system of equations, is particularly easy to solve. For example, consider the system of equations

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= -1 \\ 4x_1 + 5x_2 + 6x_3 &= -1 \\ 7x_1 + 8x_2 + 10x_3 &= 0\,, \end{aligned}$$

which is equivalent to the matrix equation $\mathtt{Ax} = \mathtt{b}$ where

$$\mathtt{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{pmatrix} \quad \text{and} \quad \mathtt{b} = \begin{pmatrix} -1 \\ -1 \\ 0 \end{pmatrix}\,.$$

The augmented matrix for this linear system is

$$\begin{array}{ccc|c} x_1 & x_2 & x_3 & = & \text{rhs} \\ \left(\begin{array}{ccc|c} 1 & 2 & 3 & -1 \\ 4 & 5 & 6 & -1 \\ 7 & 8 & 10 & 0 \end{array}\right) \end{array}.$$

(We have included the header information for the last time.) Entering

```
≫  rref([A b])
```
returns the augmented matrix
$$\left(\begin{array}{ccc|c} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & -3 \\ 0 & 0 & 1 & 1 \end{array}\right) .$$
Clearly, the solution of the linear system is $x_1 = 2,\ x_2 = -3,$ and $x_3 = 1$.

Of course, you could just as easily have found the solution by
```
≫  x = A\b
```
so let us now consider the slightly different linear system
$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= -1 \\ 4x_1 + 5x_2 + 6x_3 &= -1 \\ 7x_1 + 6x_2 + 9x_3 &= -1 \,, \end{aligned}$$
This is equivalent to the matrix equation $\mathtt{Ax} = \mathtt{b}$ where
$$\mathtt{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \quad \text{and} \quad \mathtt{b} = \begin{pmatrix} -1 \\ -1 \\ -1 \end{pmatrix} .$$
Since `A` is a singular matrix, the linear system has either no solutions or an infinite number of solutions. The augmented matrix for this linear system is
$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & -1 \\ 4 & 5 & 6 & -1 \\ 7 & 8 & 9 & 0 \end{array}\right) .$$
Entering
```
≫  rref([A b])
```
returns the augmented matrix
$$\left(\begin{array}{ccc|c} 1 & 0 & -1 & 1 \\ 0 & 1 & 2 & -1 \\ 0 & 0 & 0 & 0 \end{array}\right) ,$$
so the solution of the linear system is $x_1 = 1 + x_3$ and $x_2 = -1 - 2x_3$ for any $x_3 \in \mathbb{R}$ (or $\mathbb{C}$ if desired). In vector form, the solution is
$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 + x_3 \\ -1 - 2x_3 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix} + \begin{pmatrix} x_3 \\ -2x_3 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix} + x_3 \begin{pmatrix} 1 \\ -2 \\ 1 \end{pmatrix} .$$

Suppose you modify the matrix equation slightly by letting $\mathtt{b} = (-1, -1, 0)^{\mathrm{T}}$. Now entering
```
≫  rref([A b])
```
results in the augmented matrix
$$\left(\begin{array}{ccc|c} 1 & 0 & -1 & 1 \\ 0 & 1 & 2 & -1 \\ 0 & 0 & 0 & 1 \end{array}\right) .$$
Since the third equation is $0 = 1,$ there is clearly no solution to the linear system.

Warning: The command `rref` does not always give correct results. For example, if
$$\mathtt{C} = \begin{pmatrix} 0.95 & 0.03 \\ 0.05 & 0.97 \end{pmatrix}$$

then the matrix `I − C` is singular (where `I` is the identity matrix). However, if you solve $(\mathtt{I}-\mathtt{C})\mathtt{x}=\mathbf{0}$ by

```
≫  C = [0.95 0.03; 0.05 0.97];
≫  rref([eye(size(C))-C  [0 0]'])
```

MATLAB displays

```
ans =
     1   0   0
     0   1   0
```

which indicates that the only solution is $\mathtt{x}=\mathbf{0}$. On the other hand, if you enter

```
≫  C = [0.95 0.03; 0.05 0.97]; b = 1;
≫  rref([eye(size(C))-C  [b 0]'])
```

then MATLAB realizes that `I − C` is singular. Clearly there is some value of `b` between 0 and 1 where MATLAB switches between believing that `I − C` is non-singular and singular.†

Solving Linear Systems

`linsolve(A, b, <properties>)`	Solve the linear system of equations $\mathtt{Ax}=\mathtt{b}$ where `A` has certain properties.
`rref`	Calculates the reduced row echelon form of a matrix or an augmented matrix.

5.2. Catastrophic Round-Off Errors

We have mentioned repeatedly that **computers cannot add, subtract, multiply, or divide correctly!** Up until now, the errors that have resulted have been **very small**. Now we present two examples where the errors are **very large**.

In this first example, the reason for the large errors is easy to understand. Consider the matrix

$$\mathtt{A}_\epsilon = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9+\epsilon \end{pmatrix},$$

which is singular when $\epsilon = 0$ and nonsingular otherwise. But how well does MATLAB do when $\epsilon \ll 1$? Enter

```
≫  eps = input('eps = '); A = [1 2 3; 4 5 6; 7 8 9+eps]; inv(A)*A - eye(size(A))
```

so that the final matrix should be `0`. Begin by letting $\epsilon = 0$ and observe that the result displayed is nowhere close to the zero matrix! However, note that MATLAB is warning you that it thinks something is wrong with the statement

---

†To understand this "switch", look at the actual coding of `rref`. It uses the variable `tol` to determine whether an element of the augmented matrix

$$\left(\begin{array}{cc|c} 0.05 & -0.03 & b_1 \\ -0.05 & 0.03 & b_2 \end{array}\right)$$

is "small enough" that it should be set to 0. `tol` is (essentially) calculated by

```
tol = max(size(<augmented matrix>)) * eps * norm(<augmented matrix>, inf);
```

The maximum of the number of rows and columns of the augmented matrix, i.e., `max(size(...))`, is multiplied by `eps` and this is multiplied by the "size" of the augmented matrix. (`norm` in section 7.) Since `b` is the last column of the augmented matrix, the "size" of this matrix depends on the size of the elements of `b`. Thus, the determination whether a number "should" be set to 0 depends on the magnitude of the elements of `b`.

You can obtain the correct answer to the homogeneous equation by entering

```
≫  rref([eye(size(C))-C  [0 0]', eps)
```

which decreases the tolerance to `eps`.

```
Warning: Matrix is close to singular or badly scaled.
Results may be inaccurate. RCOND = 1.541976e-18.
```

(`RCOND` is its estimate of the inverse of the condition number. See `cond` in section 7 for more details.) Now choose some small nonzero values for ϵ and see what happens. How small can ϵ be before MATLAB warns you that the matrix is "close to singular or badly scaled"? In this example, you know that the matrix is "close to singular" if ϵ is small (but nonzero) even if MATLAB does not. The next example is more interesting.

For the second example, consider the Hilbert matrix of order n, i.e.,

$$\mathtt{H}_n = \begin{pmatrix} 1 & 1/2 & 1/3 & \cdots & 1/n \\ 1/2 & 1/3 & 1/4 & \cdots & 1/(n+1) \\ 1/3 & 1/4 & 1/5 & \cdots & 1/(n+2) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1/n & 1/(n+1) & 1/(n+2) & \cdots & 1/(2n-1) \end{pmatrix},$$

which is generated in MATLAB by

```
≫  H = hilb(n)
```

There does not seem to be anything particularly interesting, or strange, about this matrix; after all, $h_{ij} = 1/(i+j-1)$ so the elements are all of "reasonable" size. If you type

```
≫  n = 10; H = hilb(n); ( H^(1/2) )^2 - H
```

the result is not particularly surprising. The resulting matrix should be the zero matrix, but, because of round-off errors, it is not. However, every element is in magnitude less than 10^{-15}, so everything looks fine.

However, suppose you solve the matrix equation

$$\mathtt{Hx} = \mathtt{b}$$

for a given `b`. How close is the numerical solution to the exact solution? Of course, the problem is: how can you know what the analytical solution is for a given `b`? The answer is to begin with `x` and calculate `b` by $\mathtt{b} = \mathtt{Hx}$. Then solve $\mathtt{Hx} = \mathtt{b}$ for `x` and compare the final and initial values of `x`. Do this in MATLAB by

```
≫  n = 10; x = rand(n, 1); b = H*x; xnum = H\b
```

and compare `x` with `xnum` by calculating their difference, i.e.,

```
≫  x - xnum
```

The result is not very satisfactory: the maximum difference in the elements of the two vectors is usually somewhere between 10^{-5} and 10^{-3}. That is, even though all the calculations have been done to approximately 16 significant digits, the result is only accurate to **three** to **five** significant digits! (To see how much worse the result can be, repeat the above commands for $\mathtt{n} = 12$.)

It is important to realize that most calculations in MATLAB are *very* accurate. It is not that solving a matrix equation necessarily introduces lots of round-off errors; instead, Hilbert matrices are very "unstable" matrices — working with them can lead to inaccurate results. On the other hand, most matrices are quite "stable". For example, if you repeat the above sequence of steps with a random matrix, you find that the results are quite accurate. For example, enter

```
≫  n = 1000; R = rand(n); x = rand(n, 1); b = R*x; xnum = R\b; max(abs(x - xnum))
```

The results are much more reassuring, even though `n` is 100 times as large for this random matrix as for the Hilbert matrix — and even though there are over 600,000 times as many floating point operations needed to calculate `x` by Gaussian elimination for this random matrix!

Note: By entering all the commands on one line, it is easy to repeat this experiment many times for different random numbers by simply rerunning this one line.

5.3. Overdetermined and Underdetermined Linear Systems

If $\mathtt{A} \in \mathbb{C}^{m \times n}$ where $m > n$, $\mathtt{Ax} = \mathtt{b}$ is called an *overdetermined system* because there are more equations than unknowns. In general, there are no solutions to this linear equation. (However, to be sure use `rref`.) However, you can find a "best" approximation by finding the solution for which the vector

$$\mathtt{r} = \mathtt{Ax} - \mathtt{b}$$

which is called the *residual*, is smallest in Euclidean length; that is,

$$\texttt{norm(r)} \equiv \sqrt{\sum_{i=1}^{n} r_i^2}$$

is minimized. (The `norm` function is discussed in section 7.) This is called the *least-squares solution*. This best approximation is calculated in MATLAB by typing

```
≫  A\b
```

Analytically, the approximation can be calculated by solving

$$\mathtt{A}^{\mathrm{T}}\mathtt{Ax} = \mathtt{A}^{\mathrm{T}}\mathtt{b}\,.$$

However, numerically this is less accurate than the method used in MATLAB.

Note that this is the same command used to find the solution to a square linear system. This cannot be the intent here since `A` is not a square matrix. Instead, MATLAB interprets this command as asking for the least-squares solution. Again, this command only makes sense if there is a unique solution which minimizes the length of the vector $\mathtt{Ax} - \mathtt{b}$. If there are an infinite number of least-squares solutions, MATLAB warns you of this fact and then returns one of the solutions. For example, if

$$\mathtt{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} \quad \text{and} \quad b = \begin{pmatrix} 2 \\ 1 \\ 2 \\ 4 \end{pmatrix}$$

then $\mathtt{Ax} = \mathtt{b}$ has no solutions, but has an infinite number of least-square approximations. If you enter

```
≫  A\b
```

the response is

```
Warning: Rank deficient, rank = 2 tol = 1.4594e-14.
```

It also returns the solution $(-1/4, 0, 29/60)^{\mathrm{T}}$ (after using the MATLAB command `rats` which we discuss below), which is one particular least-squares approximation. To find all the solutions, you can use `rref` to solve $\mathtt{A}^{\mathrm{T}}\mathtt{Ax} = \mathtt{A}^{\mathrm{T}}\mathtt{b}$. (If `A` is complex, solve $\mathtt{A}^{\mathrm{H}}\mathtt{Ax} = \mathtt{A}^{\mathrm{H}}\mathtt{b}$.)

Occasionally, if there are an infinite number of least-squares approximations, the solution desired is the "smallest" one, i.e., the `x` for which the length of the vector `x` is minimized. This can be calculated using the *pseudoinverse* of `A`, denoted by $\mathtt{A}^{+}$. Since `A` is not square, it cannot have an inverse. However, the pseudoinverse is the unique $n \times m$ matrix which satisfies the *Moore-Penrose conditions*:

- $\mathtt{AA}^{+}\mathtt{A} = \mathtt{A}$
- $\mathtt{A}^{+}\mathtt{AA}^{+} = \mathtt{A}^{+}$
- $(\mathtt{AA}^{+})^{\mathrm{T}} = \mathtt{AA}^{+}$
- $(\mathtt{A}^{+}\mathtt{A})^{\mathrm{T}} = \mathtt{A}^{+}\mathtt{A}$

In particular, if `A` is a square nonsingular matrix, then $\mathtt{A}^{+}$ is precisely $\mathtt{A}^{-1}$. This pseudoinverse is calculated in MATLAB by entering

```
≫  pinv(A)
```

The reason for mentioning the pseudoinverse of `A` is that the least-squares approximation to $\mathtt{Ax} = \mathtt{b}$ can also be calculated by

```
≫  pinv(A)*b
```

If there are an infinite number of least-squares approximations, this returns the one with the smallest length.

Next, suppose that $\mathtt{A} \in \mathbb{C}^{m \times n}$ with $m < n$. $\mathtt{Ax} = \mathtt{b}$ is called an *underdetermined system* because there are less equations than unknowns. In general, there are an infinite number of solutions to this equation. We can find these solutions by entering

```
≫  rref([A b])
```

and solving the result for `x`. We can find one particular solution by entering

```
≫  A\b
```

This solution will have many of its elements being 0. We can also find the solution with the smallest length by entering

```
≫  pinv(A)*b
```

Warning: It is possible for an overdetermined system to have one or even an infinite number of solutions (not least-squares approximations). It is also possible for an underdetermined system to have no solutions. The way to check the number of solutions is to use the `rref` command.

One command which is occasionally useful is `rats`. If all the elements of `A` and `b` are rational numbers, then the solution and/or approximation obtained is usually a rational number, although stored as a floating-point number. This command displays a "close" rational approximation to the floating-point number, which may or may not be the exact answer. For example, entering

```
≫  rats(1/3 - 1/17 + 1/5)
```

results in the text variable `121/255`, which is the correct answer.

Warning: Be careful when using this command. `rats(sqrt(2))` makes no sense (as was known in 500 BC).

Solving Linear Systems

`A\b`	When `Ax = b` is an overdetermined system, i.e., $m > n$ where `A` $\in \mathbb{C}^{m\times n}$, this is the least-squares approximation; when it is an underdetermined solution, i.e., $m < n$, this is a solution which has 0 in many of its elements.
`pinv(A)`	The pseudoinverse of `A`.
`rats(x)`	Calculates a "close" approximation to the floating-point number `x`. This is frequently the exact value.

6. File Input-Output

In section 4.1 we discussed the `csvread` and `csvwrite` commands which allow simple input from and output to a file. The MATLAB commands `fscanf` and `fprintf`, which behave very similarly to their C counterparts, allow much finer control over input and output. Before using them a file has to be opened by

```
≫  fid = fopen('<file name>', <permission string>)
```

where the file identifier `fid` is a unique nonnegative integer attached to the file. (Three file identifiers always exist as in C: 0 is the standard input, 1 is the standard output, and 2 is the standard error.) The permission string specifies how the file is to be accessed:

- `'r'` read only from the file.
- `'w'` write only to the file (anything previously contained in the file is overwritten). If necessary, the file is created.
- `'a'` append to the end of the file (everything previously contained in the file is retained).
- `'r+'` read from and write to the file (anything previously contained in the file is overwritten).
- `'w+'` read from and write to the file (anything previously contained in the file is overwritten). If necessary, the file is created.

If the `fopen` command fails, -1 is returned in the file identifier. Enter

```
≫  fclose(fid)
```

if a file needs to be closed.

To write formatted data to a file, enter

```
≫  fprintf(fid, <format string>, <variable 1>, <variable 2>, ...)
```

The elements contained in the variables are written to the file specified in a previous `fopen` command according to the format string. If `fid` is omitted, the output appears on the screen. The format string is very similar to that of C, with the exception that the format string is cycled through until the end of the file is reached or the number of elements specified by `size` is attained.

To briefly review some of the C format specifications, the conversion characters are:

`d` – The argument is converted to decimal notation.
`c` – The argument is a single character.
`s` – The argument is a string.
`e` – The argument is a floating-point number in "E" format.
`f` – The argument is a floating-point number in decimal notation.
`g` – The argument is a floating-point number in either "E" or decimal notation.

Each conversion character is preceded by "%". The following may appear between the "%" and the conversion character:

- A minus sign which specifies left adjustment rather than right adjustment.
- An integer which specifies a minimum field width.
- If the maximum field width is larger than the minimum field width, the minimum field width is preceded by an integer which specifies the maximum field width, and the two integers are separated by a period.

`fprintf` can also be used to format data on the screen by omitting the `fid` at the beginning of the argument list. Thus, it is possible to display a variable using as little or as much control as desired. For example, if `x` contains -23.6 three different ways to display it are

```
≫  x
≫  disp(['x = ', num2str(x)])
≫  fprintf('%12.6e\n', x)
```

and the results are

```
x =

  -23.6000

x = -23.6000
-2.360000e+01
```

Note: It is easy to print the matrix `A` in the MATLAB workspace as we just described. However, it is a little more difficult to print it to a file. The following works and can be entered on one line, although it is actually a number of statements.

```
≫  Str = num2str(A); for i = [1:size(Str, 1)] fprintf(fid, '%s\n', Str(i,:));
end
```

To read formatted data from a file, enter

```
≫  A = fscanf(fid, <format string>, <size>)
```

The data is read from the file specified in a previous `fopen` command according to the format string and put into the matrix `A`. The size argument, which puts an upper limit on the amount of data to be read, is optional. If it is a scalar, or is not used at all, `A` is actually a vector. If it is `[m n]`, then `A` is a matrix of this size.

Advanced Input-Output

`fopen('<file name>', <permission string>)`	Opens the file with the permission string determining how the file is to be accessed. The function returns the file identifier, which is a unique nonnegative integer attached to the file.
`fclose(fid)`	Closes the file with the given file identifier.
`fscanf(fid, <format string>)`	Behaves very similarly to the C command in reading data from a file using any desired format.
`fprintf(fid, <format string>, <variable 1>,...)`	Behaves very similarly to the C command in writing data to a file using any desired format.
`fprintf(<format string>, <variable 1>,...)`	Behaves very similarly to the C command in displaying data on the screen using any desired format.

7. Some Useful Linear Algebra Commands

We briefly describe in alphabetical order some of the MATLAB commands that are most useful in linear algebra. Most of these discussions can be read independently of the others. Where this is not true, we indicate which should be read first.

chol

Let $\mathtt{A} \in \mathbb{R}^{n\times n}$ be symmetric and positive definite[†]. Then there exists an upper triangular matrix R such that $\mathtt{R}^{\mathrm{T}}\mathtt{R} = \mathtt{A}$. R is calculated by

```
≫  R = chol(A)
```

If A is not positive definite, an error message is printed. (If $\mathtt{A} \in \mathbb{C}^{n\times n}$ then $\mathtt{R}^{\mathrm{H}}\mathtt{R} = \mathtt{A}$.)

cond

Note: Read the discussion on norm below first.

The condition number of $\mathtt{A} \in \mathbb{C}^{n\times n}$, which is denoted by cond(A), is a positive real number which is always ≥ 1. It measures how "stable" A is: if $\mathrm{cond}(\mathtt{A}) = \infty$ the matrix is singular, while if $\mathrm{cond}(\mathtt{A}) = 1$ the matrix is as nice a matrix as you could hope for — in particular, $\mathrm{cond}(\mathtt{I}) = 1$. To estimate the number of digits of accuracy you might lose in solving the linear system $\mathtt{Ax} = \mathtt{b}$, enter

```
log10(cond(A))
```

In subsection 5.2 we discussed the number of digits of accuracy you might lose in solving $\mathtt{Hx} = \mathtt{b}$ where H is the Hilbert matrix of order 10. In doing many calculations it was clear that the solution was only accurate to 3 to 5 significant digits. Since cond(H) is 1.6×10^{13}, it is clear that you should lose about 13 of the 16 digits of accuracy in this calculation. Thus, everything fits.

If A is nonsingular, the condition number is defined by

$$\mathrm{cond}_p(\mathtt{A}) = \|\mathtt{A}\|_p \, \|\mathtt{A}^{-1}\|_p \qquad \text{for } p \in [1,\infty]$$

or

$$\mathrm{cond}_{\mathrm{F}}(\mathtt{A}) = \|\mathtt{A}\|_{\mathrm{F}} \, \|\mathtt{A}^{-1}\|_{\mathrm{F}} \, .$$

It is calculated in MATLAB by

```
≫  cond(A, p)
```

where p is 1, 2, Inf, or 'fro'. If p = 2 the command can be shortened to

```
≫  cond(A)
```

Note that the calculation of the condition number of A requires the calculation of the inverse of A. The MATLAB command condest approximates the condition number without having to calculate this inverse. See the discussion of this command below for further information on when it might be preferable.

Note: Sometimes we want to solve, or find the "best" approximation to, $\mathtt{Ax} = \mathtt{b}$ when $\mathtt{A} \in \mathbb{C}^{m\times n}$ is not a square matrix. (This is discussed in detail in subsection 5.3.) Since we still want to know the accuracy of any solution, we want to generalize the condition number to nonsquare matrices. This is done by defining the condition number of a nonsquare matrix in the 2-norm to be the ratio of the largest to the smallest singular value of A, i.e., $\sigma_1/\sigma_{\min\{m,n\}}$.

condest

Note: Read the discussion on cond above first.

The calculation of the condition number of $\mathtt{A} \in \mathbb{C}^{n\times n}$ requires the calculation of its inverse. There are two reasons this might be inadvisable.

- The calculation of $\mathtt{A}^{-1}$ requires approximately $2n^3$ flops, which might take too long if n is **very large**.
- If A is a sparse matrix (i.e., most of its elements are zero), we discuss in section 9 how to store only the nonzero elements of A to conserve storage. (For example, if $n = 10{,}000$ and A is tridiagonal[‡], the number of nonzero elements in A is approximately 30,000 but the total number of elements in A

[†] $\mathtt{A} \in \mathbb{R}^{n\times n}$ is positive definite if $\mathtt{x}^{\mathrm{T}}\mathtt{Ax} \geq 0$ for all $\mathtt{x} \in \mathbb{R}^n$ and $\mathtt{x}^{\mathrm{T}}\mathtt{Ax} = 0$ only if $\mathtt{x} = \mathbf{0}$. In practical terms, it means that all the eigenvalues of A are positive. ($\mathtt{A} \in \mathbb{C}^{n\times n}$ is positive definite if $\mathtt{x}^{\mathrm{H}}\mathtt{Ax} \geq 0$ for all $\mathtt{x} \in \mathbb{C}^n$ and $\mathtt{x}^{\mathrm{T}}\mathtt{Ax} = 0$ only if $\mathtt{x} = \mathbf{0}$.)

[‡] A matrix is *tridiagonal* if its only nonzero elements occur on the main diagonal or on the first diagonal above or below the main diagonal

is 100,000,000.) Since the inverse of a sparse matrix is generally much less sparse (in fact it may have no zero elements at all), MATLAB may not be able to store `A`$^{-1}$.

The command `condest` calculates a lower bound to the condition number of a matrix in the 1-norm without having to determine its inverse. This approximation is almost always within a factor of ten of the exact value.

When MATLAB calculates `A\b` or `inv(A)`, it also calculates `condest(A)`. It checks if its estimate of the condition number is large enough that `A` is likely to be singular. If so, it returns an error message such as

```
Warning: Matrix is close to singular or badly scaled.
Results may be inaccurate. RCOND = 2.055969e-18.
```

where `RCOND` is the inverse of `condest(A)`.

det

Let $\mathtt{A} \in \mathbb{C}^{n\times n}$. The determinant of `A` is calculated by

```
≫  det(A)
```

$\det(\mathtt{A}) = 0$ if and only if `A` is singular. However, due to round-off errors it is very unlikely that you will obtain 0 numerically unless all the entries to `A` are integers. For example, consider the matrix

$$\mathtt{C} = \begin{pmatrix} 0.95 & 0.03 \\ 0.05 & 0.97 \end{pmatrix} .$$

$\mathtt{I} - \mathtt{C}$ is singular (where `I` is the identity matrix) but

```
≫  C = [0.95 0.03; 0.05 0.97]; det( eye(size(C)) - C )
```

does not return 0. However, the number it returns is much smaller than `eps` and so it seems "reasonable" that $\mathtt{I} - \mathtt{C}$ is singular. On the other hand,

```
≫  det(hilb(10))
```

returns 2.2×10^{-53}, but the Hilbert matrix is not singular for any n. (The singular value decomposition, which is described below, is a much better method for determining if a square matrix is singular.)

eig

Let $\mathtt{A} \in \mathbb{C}^{n\times n}$. A scalar $\lambda \in \mathbb{C}$ is an *eigenvalue* of `A` if there exists a nonzero vector $\mathtt{v} \in \mathbb{C}^n$ such that

$$\mathtt{A}\mathtt{v} = \lambda\mathtt{v} \, ;$$

`v` is called the *eigenvector* corresponding to λ. There are always n eigenvalues of `A`, although they need not all be distinct. MATLAB will very happily calculate all the eigenvalues of `A` by

```
≫  eig(A)
```

It will also calculate all the eigenvectors by

```
≫  [V, D] = eig(A)
```

$\mathtt{D} \in \mathbb{C}^{n\times n}$ is a diagonal matrix containing the n eigenvalues on its diagonal and the corresponding eigenvectors are found in the same columns of the matrix $\mathtt{V} \in \mathbb{C}^{n\times n}$.

A matrix is *defective* if it has less eigenvectors than eigenvalues. MATLAB normally cannot determine when this occurs. For example, the matrix

$$\mathtt{B} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

is defective since it has two eigenvalues, both of which are 1, but it only has one eigenvector, namely $(1,0)^{\mathrm{T}}$. If you enter

```
≫  B = [1 1; 0 1]; [V, D] = eig(B)
```

MATLAB calculates the two eigenvalues correctly, but it finds the two eigenvectors $(1,0)^{\mathrm{T}}$ and $(-1, 2.2\times10^{-16})^{\mathrm{T}}$. Clearly the latter eigenvector should be $(-1,0)^{\mathrm{T}}$ so that, in fact, there is only one eigenvector.

Note: If `A` is a sparse matrix (see Section 9), you cannot use `eig`. You either have to use the function `eigs` or do `eig(full(A))`.

`eigs`

Note: Read the discussion on `eig` above first.

Frequently, you do not need *all* the eigenvalues of a matrix. For example, you might only need the largest ten in magnitude, or the five with the largest real part, or the one which is smallest in magnitude, or ... In addition, you might need some eigenvalues of the *generalized eigenvalue problem*

$$\mathtt{A}\mathtt{x} = \lambda \mathtt{B}\mathtt{x}$$

where `B` is a symmetric positive definite matrix. (If `B` is complex, it must be Hermetian.) `eigs` can do all of this. Of course, this means that there are numerous possible arguments to this function so read the documentation carefully.

Why not just use `eig` anyway? Calculating all the eigenvalues of $\mathtt{A} \in \mathbb{R}^{n\times n}$ requires (very) approximately $10n^3$ flops, which can take a *very* long time if n is very large. On the other hand, calculating only a few eigenvalues requires *many, many* fewer flops. If `A` is a full matrix, it requires cn^2 flops where c is of "reasonable" size; if `A` is a sparse matrix (see Section 9), it requires cn flops.

Note: If `A` is sparse, you cannot use `eig` — you will first have to do `eig(full(A))`.

Also, this command generates lots of diagnostic output. To calculate the largest 3 eigenvalues of `A` in magnitude without generating any diagnostics, enter

```
>> op.disp = 0
>> eigs(A, 3, 'LM', op)
```

(`op.disp` is a structure, which was discussed in section 3.4.)

`inv`

To calculate the inverse of the square matrix $\mathtt{A} \in \mathbb{C}^{n\times n}$ enter

```
>> inv(A)
```

The inverse of `A`, denoted by $\mathtt{A}^{-1}$, is a matrix such that $\mathtt{A}\mathtt{A}^{-1} = \mathtt{A}^{-1}\mathtt{A} = \mathtt{I}$, where $\mathtt{I} \in \mathbb{R}^{n\times n}$ is the identity matrix. If such a matrix exists, it must be unique.

MATLAB cannot always tell whether this matrix does, in fact, exist. For example, the matrix

$$\mathtt{A} = \begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix}$$

does not have an inverse. If you try to take the inverse of this matrix, MATLAB will complain that

```
Warning: Matrix is singular to working precision.
```

It will display the inverse matrix, but all the entries will be `Inf`.

The above matrix was very simple. The matrix

$$\mathtt{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \tag{7.1}$$

also does not have an inverse. If you ask MATLAB to calculate the inverse of `A`, it will complain that

```
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 2.055969e-18.
```

(`RCOND` is the inverse of a numerical approximation to the condition number of `A`; see `condest` above.) That is, MATLAB is not *positive* that `A` is singular, because of round-off errors, but it thinks it is likely. However, MATLAB still does try to calculate the inverse. Of course, if you multiply this matrix by `A` the result is nowhere close to `I`. (Try it!) In other words, be careful — and read (and understand) all

warning messages.

lu

Let $\mathtt{A} \in \mathbb{C}^{n\times n}$. Then there exists an upper triangular matrix `U`, a unit lower triangular matrix `L`[†], and a permutation matrix `P`[‡] such that

$$\mathtt{LU} = \mathtt{PA}\,.$$

The MATLAB command `lu` calculates these matrices by entering

```
≫  [L, U, P] = lu(A)
```

If `A` is invertible, all the elements of `U` on the main diagonal are nonzero. If you enter

```
≫  A = [1 2 3; 4 5 6; 7 8 9]; [L, U, P] = lu(A)
```

where `A` is the singular matrix defined earlier, u_{33} should be zero. Entering

```
≫  U(3,3)
```

displays `1.1102e-16`, which clearly should be zero as we discussed in subsection 1.5.

Note: This is the first time we have had a function return more than one argument. We discuss this notation in detail in section 8.3. For now, we simply state that when `[V, D]` occurs on the right side of the equal sign it means the matrix whose first columns come from `V` and whose last columns come from `D`. However, on the left side of the equal sign it means that the function returns two arguments where the first is stored in the variable `V` and the second in `D`.

norm

The norm of a vector or matrix is a nonnegative real number which gives some measure of the "size" of the vector or matrix. The p-th norm of a vector is defined by

$$\|\mathtt{x}\|_p = \begin{cases} \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p} & \text{if } p \in [1,\infty) \\ \max\limits_{1\le i\le n} |x_i| & \text{if } p = \infty\,. \end{cases}$$

For $p = 1$, 2, or ∞ it is calculated in MATLAB by entering

```
≫  norm(x, p)
```

where `p` is `1`, `2`, or `Inf`. If `p = 2` the command can be shortened to

```
≫  norm(x)
```

The p-th norm of a matrix is defined by

$$\|\mathtt{A}\|_p = \max_{\mathtt{x}\neq\mathtt{0}} \frac{\|\mathtt{Ax}\|_p}{\|\mathtt{x}\|_p} \qquad \text{for } p \in [1,\infty]$$

and is calculated in MATLAB by entering

```
≫  norm(A, p)
```

where again `p` is `1`, `2`, or `Inf`. If `p = 2` the command can be shortened to

```
≫  norm(A)
```

There is another matrix norm, the Frobenius norm, which is defined for $\mathtt{A} \in \mathbb{C}^{m\times n}$ by

$$\|\mathtt{A}\|_{\mathrm{F}} = \left(\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2\right)^{1/2}$$

and is calculated in MATLAB by entering

```
≫  norm(A, 'fro')
```

null

Let $\mathtt{A} \in \mathbb{C}^{n\times n}$. We can calculate an orthonormal basis for the null space of `A` by

[†]A *unit* lower triangular matrix is lower triangular and, in addition, all the elements on the main diagonal are 1.

[‡]`P` is a *permutation matrix* if its columns are a rearrangement of the columns of `I`.

```
≫ null(A)
```

orth

Let $\mathtt{A} \in \mathbb{C}^{n\times n}$. We can calculate an orthonormal basis for the columns of A by

```
≫ orth(A)
```

qr

Let $\mathtt{A} \in \mathbb{R}^{m\times n}$. Then there exists an orthogonal matrix $\mathtt{Q} \in \mathbb{R}^{m\times m}$ † and an upper triangular matrix $\mathtt{R} \in \mathbb{R}^{m\times n}$ such that

$$\mathtt{A} = \mathtt{QR}\,.$$

(If $\mathtt{A} \in \mathbb{C}^{m\times n}$ then there exists an unitary matrix $\mathtt{Q} \in \mathbb{C}^{m\times m}$ and an upper triangular matrix $\mathtt{R} \in \mathbb{C}^{m\times n}$ such that $\mathtt{A} = \mathtt{QR}$.) We calculate Q and R in MATLAB by entering

```
≫ [Q, R] = qr(A)
```

It is frequently preferable to add the requirement that the diagonal elements of R be decreasing in magnitude, i.e., $|r_{i+1,i+1}| \le |r_{i,k}|$ for all i. In this case

$$\mathtt{AE} = \mathtt{QR}$$

for some permutation matrix E and

```
≫ [Q, R, E] = qr(A)
```

One reason for this additional requirment on R is that you can immediately obtain an orthonormal basis for the range of A and the null space of $\mathtt{A}^{\mathrm{T}}$. If $r_{k,k}$ is the last nonzero diagonal element of R, then the first k columns of Q are an orthonormal basis for the range of A and the final $n-k$ columns are an orthonormal basis for the null space of $\mathtt{A}^{\mathrm{T}}$. The command orth is preferable if all you want is an orthonormal basis for $R(\mathtt{A})$.

rank

Let $\mathtt{A} \in \mathbb{C}^{m\times n}$. The rank of A is the number of linearly independent columns of A and is calculated by

```
≫ rank(A)
```

This number is calculated by using the singular value decomposition, which we discuss below.

svd

Let $\mathtt{A} \in \mathbb{R}^{m\times n}$. A can be decomposed into

$$\mathtt{A} = \mathtt{U\Sigma V}^{\mathrm{T}}$$

where $\mathtt{U} \in \mathbb{R}^{m\times m}$ and $\mathtt{V} \in \mathbb{R}^{n\times n}$ are orthogonal matrices and $\mathtt{\Sigma} \in \mathbb{R}^{m\times n}$ is a diagonal matrix (although not necessarily square) with real nonnegative elements in decreasing order. That is,

$$\sigma_1 \ge \sigma_2 \ge \cdots \ge \sigma_{\min\{m,n\}} \ge 0\,.$$

(If $\mathtt{A} \in \mathbb{C}^{m\times n}$ then $\mathtt{U} \in \mathbb{C}^{m\times m}$ and $\mathtt{V} \in \mathbb{C}^{n\times n}$ are unitary matrices and $\mathtt{\Sigma} \in \mathbb{R}^{m\times n}$ is a diagonal matrix with real nonnegative elements in decreasing order.) These matrices are calculated by

```
≫ [U, S, V] = svd(A)
```

The diagonal elements of Σ are called the *singular values* of A. Although A need not be a square matrix, both $\mathtt{A}^{\mathrm{T}}\mathtt{A} \in \mathbb{R}^{n\times n}$ and $\mathtt{AA}^{\mathrm{T}} \in \mathbb{R}^{m\times m}$ are square symmetric matrices. (If A is complex, $\mathtt{A}^{\mathrm{H}}\mathtt{A}$ and $\mathtt{AA}^{\mathrm{H}}$ are both square Hermitian matrices.) Thus, their eigenvalues are nonnegative.‡ Their nonzero eigenvalues are the squares of the singular values of A.§ In addition, the eigenvectors of $\mathtt{A}^{\mathrm{T}}\mathtt{A}$ are the columns of

†$\mathtt{Q} \in \mathbb{R}^{m\times m}$ is *orthogonal* if $\mathtt{Q}^{-1} = \mathtt{Q}^{\mathrm{T}}$. ($\mathtt{Q} \in \mathbb{C}^{m\times m}$ is *unitary* if $\mathtt{Q}^{-1} = \mathtt{Q}^{\mathrm{H}}$.)

‡The eigenvalues of a real square symmetric matrix are nonnegative. (The eigenvalues of a complex square Hermitian matrix are real and nonnegative.)

§For example, if $m > n$ there are n singular values and their squares are the eigenvalues of $\mathtt{A}^{\mathrm{T}}\mathtt{A}$. The m eigenvalues of $\mathtt{AA}^{\mathrm{T}}$ consist of the squares of these n singular values and $m-n$ additional zero eigenvalues.

`V` and those of $\texttt{A}\texttt{A}^{\mathrm{T}}$ are the columns of `U`. (If `A` is complex, the eigenvectors of $\texttt{A}^{\mathrm{H}}\texttt{A}$ are the columns of `V` and those of $\texttt{A}\texttt{A}^{\mathrm{H}}$ are the columns of `U`.)

The best numerical method to determine the rank of `A` is to use its singular values. For example, to see that

$$\texttt{A} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

has rank 2, use the `svd` command to find that the singular values of `A` are 25.4368, 1.7226, and 8.1079×10^{-16}. Clearly the third singular value should be 0 and so `A` has 2 nonzero singular values and so has a rank of 2. On the other hand, the Hilbert matrix of order 15 has singular values

$$1.8\times10^{0},\ 4.3\times10^{-1},\ 5.7\times10^{-2},\ 5.6\times10^{-3},\ 4.3\times10^{-4},\ 2.7\times10^{-5},\ 1.3\times10^{-6},\ 5.5\times10^{-8},$$
$$1.8\times10^{-9},\ 4.7\times10^{-11},\ 9.3\times10^{-13},\ 1.4\times10^{-14},\ 1.4\times10^{-16},\ 1.2\times10^{-17},\ \text{and}\ 2.4\times10^{-18}$$

according to MATLAB. Following Principle 1.2, you can see there is no separation between the singular values which are clearly not zero and the ones which are "close to" `eps`. Thus, you cannot conclude that any of these singular values should be set to 0. Our "best guess" is that the rank of this matrix is 15.[†]

Some Useful Functions in Linear Algebra

`chol(A)`	Calculates the Cholesky decomposition of a symmetric, positive definite square matrix.
`cond(A)`	Calculates the condition number of a square matrix. `cond(A, p)` calculates the condition number in the p-norm.
`condest(A)`	Calculates a lower bound to the condition number of `A` in the 1-norm.
`det(A)`	Calculates the determinant of a square matrix.
`eig(A)`	Calculates the eigenvalues, and eigenvectors if desired, of a square matrix.
`eigs`	Calculates some eigenvalues, and eigenvectors if desired, of a square matrix. There are numerous possible arguments to this function so read the documentation carefully.
`inv(A)`	Calculates the inverse of a square invertible matrix.
`lu(A)`	Calculates the LU decomposition of a square invertible matrix.
`norm(v)`	Calculates the norm of a vector. `norm(v, p)` calculates the p-norm.
`norm(A)`	Calculates the norm of a matrix. `norm(A, p)` calculates the p-norm.
`null(A)`	Calculates an orthonormal basis for the null space of a matrix.
`orth(A)`	Calculates an orthonormal basis for the range of a matrix.
`qr(A)`	Calculates the QR decomposition of a matrix.
`rank(A)`	Estimates the rank of a matrix.
`svd(A)`	Calculates the singular value decomposition of a matrix.

[†]In fact, it can be proven that the Hilbert matrix of order n is nonsingular for all n, and so its rank is truly n. However, if you enter

```
≫  rank( hilb(15) )
```

you obtain `12`, so that MATLAB is off by three.

8. Programming in MATLAB

Using the commands we have already discussed, MATLAB can do very complicated matrix operations. However, sometimes there is a need for finer control over the elements of matrices and the ability to test, and branch on, logical conditions. Although prior familiarity with a high-level programming language is useful, MATLAB's programming language is so simple that it can be learned quite easily and quickly.

8.1. Flow Control and Logical Variables

MATLAB has four flow control and/or branching instructions: `for` loops, `while` loops, `if-else` branching tests, and `switch` branching tests.

Notation: All of these instructions end with an `end` statement, and it is frequently difficult to determine the extent of these instructions. *Thus, it is* **very important** *to use indentation to indicate the structure of a code*, as we do in the remainder of this tutorial. This greatly increases the readability of the code for human beings.

The general form of the `for` loop is

```
≫  for <variable> = <expression>
      <statement>
      ...
      <statement>
   end
```

where the variable is often called the *index* of the loop. The elements of the row vector `<expression>` are stored one at a time in the variable and then the statements up to the `end` statement are executed.† For example, you can define the vector $\mathtt{x} \in \mathbb{R}^n$ where $x_i = i\sin(i^2\pi/n)$ by

```
≫  x = zeros(n, 1);
≫  for i = 1:n
      x(i) = i * sin( i^2 *pi/n );
   end
```

(The first line is not actually needed, but it allows MATLAB to know exactly the size of the final vector before the `for` loops begin. This saves computational time and makes the code more understandable; it is discussed in more detail in section 8.5.) In fact, the entire `for` loop could have been entered on one line as

```
≫  for i = 1:n  x(i) = i * sin( i^2 *pi/n ); end
```

However, for readability it is best to split it up and to indent the statements inside the loop. Of course, you can also generate the vector by

```
≫  x = [1:n]' .* sin( [1:n]' .^2 *pi/n )
```

which is certainly "cleaner" and executes much faster in MATLAB.

Warning: In using `i` as the index of the `for` loop, `i` has just been redefined to be n instead of $\sqrt{-1}$. *Caveat emptor!*

A more practical example of the use of a `for` loop is the generation of the Hilbert matrix of order n, which we have already discussed a number of times. This is easily done using two `for` loops by

```
≫  H = zeros(n);
≫  for i = 1:n
      for j = 1:n
         H(i,j) = 1/(i + j - 1);
      end
   end
```

Warning: In using `i` and `j` as the indices of the `for` loops, `i` and `j` have just been redefined to be n instead of $\sqrt{-1}$. *Caveat emptor!*

† `<expression>` can be a matrix in which case each column vector is stored one at a time in `i`.

`for` loops often have branches in them. For this we need the `if` branch, which we now describe. The simplest form of the `if` statement is

```
» if <logical expression>
     <statement>
     ...
     <statement>
  end
```

where the statements are evaluated as long as the `<logical expression>` is true. The `<logical expression>` is generally of the form

`<arithmetic expression-left>` *rop* `<arithmetic expression-right>`

where *rop* is one of the *r*elational *op*erators shown below. Some examples of logical expressions are

```
i == 5
x(i) >= i
imag(A(i,i)) ~= 0
sin(1) - 1 > x(1) + x(i)^3
```

Warning: String variables cannot be easily compared by `==` or `~=`.† Instead, if `a` and `b` are text variables, enter

```
» strcmp(a, b)
```

The result is true if the two character strings are identical and false otherwise.

Relational Operators

<	Less than.	>	Greater than.
<=	Less than or equal to.	>=	Greater than or equal to.
==	Equal.	~=	Not equal to.
		`strcmp(a, b)`	Compares strings.

A second form of the `if` statement is

```
» if <logical expression>
     <statement group 1>
  else
     <statement group 2>
  end
```

where statement group 1 is evaluated if the `<logical expression>` is true and statement group 2 is evaluated if it is false. The final form of the `if` statement is

```
» if <logical expression 1>
     <statement group 1>
  elseif <logical expression 2>
     <statement group 2>
  elseif <logical expression 3>
     <statement group 3>
  ...
  elseif <logical expression r>
     <statement group r>
  else
     <statement group r+1>
  end
```

where statement group 1 is evaluated if the `<logical expression 1>` is true, statement group 2 is evalu-

†Compare the results of

```
» 'Yes'== 'yes'
```

and

```
» 'Yes'== 'no'
```

ated if the `<logical expression 2>` is true, etc. The final `else` statement is not required. If it occurs and if none of the logical expressions is true, statement group r+1 is evaluated. If it does not occur and if none of the logical expressions is true, then none of the statement groups are executed.

When a logical expression such as

```
≫  i == 5
```

is evaluated, the result is either the logical value "TRUE" or "FALSE". MATLAB calculates this as a numerical value which is returned in the variable `ans`. The value is 0 if the expression is false and 1 if it is true.

MATLAB also contains the logical operators "AND" (denoted by "&"), "OR" (denoted by "`|`"), "NOT" (denoted by "~"), and "EXCLUSIVE OR" (invoked by the function `xor`). These act on false or true statements which are represented by numerical values: zero for false statements and nonzero for true statements. Thus, if a and b are real numbers then

- the relational equation
  ```
  ≫  c = a & b
  ```
  means that c is true (i.e., 1) only if both a and b are true (i.e., nonzero); otherwise c is false (i.e., 0).
- the relational equation
  ```
  ≫  c = a | b
  ```
  means that c is true (i.e., 1) if a and/or b is true (i.e., nonzero); otherwise c is false (i.e., 0).
- the relational equation
  ```
  ≫  c = ~a
  ```
  means that c is true (i.e., 1) if a is false (i.e., 0); otherwise c is false (i.e., 0).
- the relational command
  ```
  ≫  c = xor(a, b)
  ```
  means that c is true (i.e., 1) if exactly one of a and b is true (i.e., nonzero); otherwise c is false (i.e., 0).

In the above statements `c` is a logical variable which has the logical value "TRUE" or "FALSE". Frequently — but not always — the variable can be set by `c = 1` or `c = 0`; but `c` is now **not** a logical variable, but a numerical variable. Frequently — but not always — a numerical variable can be used instead of a logical variable. The preferred ways to set a logical variable are the following. The logical variable can be set by `c = logical(1)` or `c = logical(0)` — and now `c` is a logical variable. A simpler way to set the logical variable `c` is `c = true` or `c = false`.

There are second logical operators "AND" (denoted by `&&`) and "OR" (`||`) which are rarely needed. The statement

```
≫  c = a && b
```

returns a (scalar) logical `true` if both inputs evaluate to `true` (so if they are variables they both must be scalars). The difference from `&` is that if `a` is `false` then `b` is not evaluated. Similarly,

```
≫  c = a || b
```

returns a (scalar) logical `true` if either input evaluates to `true`. If `a` is `true` then `b` is not evaluated (as in C, C++, and Java).

Logical Operators

`A & B`	AND.	`a && b`	Short-circuit AND. Returns logical 1 (true) or 0 (false). Only evaluates `b` if `a` is true.
`A | B`	OR.	`a || b`	Short-circuit OR. Returns logical 1 (true) or 0 (false). Only evaluates `b` if `a` is false.
`~A`	NOT.		
`xor(A, B)`	EXCLUSIVE OR.		

The second MATLAB loop structure is the `while` statement. The general form of the `while` loop is

```
≫  while <logical expression>
      <statement>
      ...
      <statement>
   end
```

where the statements are executed repeatedly as long as the `<logical expression>` is true. For example, `eps` can be calculated by

```
≫  eps = 1;
≫  while 1 + eps > 1
      eps = eps/2;
   end
≫  eps = 2*eps
```

It is possible to break out of a `for` loop or a `while` loop from inside the loop by using the `break` command as in C. This terminates the execution of the innermost `for` loop or `while` loop.

The `continue` statement is related to `break`. It causes the next iteration of the `for` or `while` loop to begin immediately.

The `switch` command executes particular statements based on the value of a variable or an expression. Its general form is

```
≫  switch <variable or expression>
      case <Value 1>,
         <statement group 1>
      case {<Value 2a>, <Value 2b>, <Value 2c>, ..., <Value 2m>},
         <statement group 2>
      ...
      case <value n>,
         <statement group r>
      otherwise,
         <statement group r+1>
   end
```

where statement group 1 is evaluated if the variable or expression has `<Value 1>`, where statement group 2 is evaluated if the variable or expression has values `<Value 2a>` or `<Value 2b>` or `<Value 2c>`, etc. (Note that if a case has more than one value, then all the values must be surrounded by curly brackets.) The final `otherwise` is not required. If it occurs and if none of the values match the variable or expression, then statement group `r+1` is evaluated. If it does not occur and if none of the values match, then none of the statement groups are executed.

Warning: The `switch` command is different in MATLAB than in C in two ways:

First, in MATLAB the `case` statement can contain more than one value; in C it can only contain one.

And, second, in MATLAB only the statements between the selected case and the following one or the following `otherwise` or `end` (whichever occurs first) are executed; in C *all* the statements following the selected case are executed up to the next `break` or the end of the block.

Flow Control

`break`	Terminates execution of a `for` or `while` loop.
`case`	Part of the `switch` command. The statements following it are executed if its value or values are a match for the `switch` expression.
`continue`	Begins the next iteration of a `for` or `while` loop immediately.
`else`	Used with the `if` statement.
`elseif`	Used with the `if` statement.
`end`	Terminates the `for`, `if`, `switch`, and `while` statements.
`for`	Repeats statements a specific number of times.
`if`	Executes statements if certain conditions are met.
`otherwise`	Part of the `switch` command. The statements following it are executed if no `case` value is a match for the `switch` expression.
`switch`	Selects certain statements based on the value of the `switch` expression.
`while`	Repeats statements as long as an expression is true.

Elementary Logical Matrices

`true`	Generates a logical matrix with all elements having the logical value true. Use `true` or `true(n)` or `true(m, n)`.
`false`	Generates a logical matrix with all elements having the logical value false. Use `false` or `false(n)` or `false(m, n)`.

8.2. Matrix Relational Operators and Logical Operators

Although MATLAB does have a quite powerful programming language, it is needed much less frequently than in typical high-level languages. Many of the operations and functions that can only be applied to scalar quantities in other languages can be applied to vector and matrices in MATLAB. For example, MATLAB's relational and logical operators can also be applied to vectors and matrices. In this way, algorithms that would normally require flow control for coding in most programming languages can be coded using simple MATLAB commands.

If $\mathtt{A}, \mathtt{B} \in \mathbb{R}^{m \times n}$ then the relational equation

≫ `C = A` *rop* `B`

is evaluated as $c_{ij} = a_{ij}$ *rop* b_{ij}, where *rop* is one of the relational operators defined previously. `C` is a logical array, that is, its data type is "logical" not "numeric". The elements of `C` are all 0 or 1: 0 if a_{ij} *rop* b_{ij} is a false statement and 1 if it is a true one. Also, the relational equation

≫ `C = A` *rop* `c`

is defined when c is a scalar. It is evaluated as if we had entered

≫ `C = A` *rop* `c*ones(size(A))`

Similar behavior holds for logical operators:

≫ `C = A & B`

means $c_{ij} = a_{ij} \,\&\, b_{ij}$,

≫ `C = A | B`

means $c_{ij} = a_{ij} \mid b_{ij}$,

≫ `C = ~A`

means $c_{ij} = \sim a_{ij}$, and

≫ `C = xor(A, B)`

means $c_{ij} = \mathtt{xor}(a_{ij}, b_{ij})$. Again the elements of `C` are all 0 or 1.

To show the power of these MATLAB commands, suppose we have entered

≫ `F = rand(m, n)`

and now we want to know how many elements of `F` are greater than 0.5. We can code this as

```
≫ nr_elements = 0;
≫ for i = 1:m
     for j = 1:n
        if F(i,j) > 0.5
           nr_elements = nr_elements + 1;
        end
     end
  end
≫ nr_elements
```

However, it can be coded much more simply, quickly, and efficiently since the relational expression

```
≫ C = F > 0.5;
```

or, to make the meaning clearer,

```
≫ C = (F > 0.5);
```

generates the matrix `C` where

$$c_{ij} = \begin{cases} 1 & \text{if } f_{ij} > 0.5 \\ 0 & \text{otherwise.} \end{cases}$$

Since the number of ones is the result we want, simply enter

```
≫ sum( sum( F > 0.5 ) )
```

or

```
≫ sum(sum(C))
```

or

```
≫ sum(C(:))
```

And suppose we want to replace all the elements of `F` which are ≤ 0.5 by zero. This is easily done by

```
≫ F = F.*(F > 0.5)
```

The relational expression `F > 0.5` generates a matrix with zeroes in all the locations where we want to zero the elements of `F` and ones otherwise. Multiplying this new matrix elementwise with `F` zeroes out all the desired elements of `F`. We can also replace all the elements of `F` which are ≤ 0.5 by $-\pi$ using

```
≫ C = (F > 0.5)
≫ F = F.*C - pi*(~C)
```

Shortly we will present two easier ways to do this.

There is even a MATLAB function which determines the location of the elements of a vector or a matrix where some property is satisfied. The command

```
≫ find(x)
```

generates a column vector containing the indices of `x` which are nonzero. (Recall that nonzero can also mean "TRUE" so that this command finds the elements where some condition is true.) For example, if $\mathtt{x} = (0, 4, 0, 1, -1, 0, \pi)^{\mathrm{T}}$ then the resulting vector is $(2, 4, 5, 7)^{\mathrm{T}}$. We can add 10 to every nonzero element of `x` by

```
≫ ix = find(x);
≫ x(ix) = x(ix) + 10;
```

Note: If no element of the vector is nonzero, the result is the empty matrix `[]`.

`find` can also be applied to a matrix. The command

```
≫ find(A)
```

first transforms `A` to a column vector (i.e., `A(:)`) and then determines the locations of the nonzero elements. Instead we can work with the matrix directly by entering

```
≫ [iA, jA] = find(A)
```

The two column vectors `iA` and `jA` contain the rows and columns, respectively, of the nonzero elements. We can also find the locations of the nonzero elements and their values by

```
≫ [iA, jA, valueA] = find(A)
```

As a simple example of the power of this command we can add 10 to every nonzero element of `A` by

```
≫ ijA = find(A); A(ijA) = A(ijA) + 10
```

Note: `ijA` contains the locations of the nonzero elements of `A` when considered to be a column vector. Since `A(k)` has no meaning in linear algebra if `k` is a scalar (since an element of `A` requires both

a row and a column number), MATLAB assumes that this is the element number of `A` as a column vector.

We can also find the elements of a vector or a matrix which satisfy a more general property than being nonzero. For example, to find the locations of all the elements of `x` which are greater than `5` enter

```
≫ find(x > 5)
```

and to find the locations of all the elements of `x` which are greater than `5` and less than `8` enter

```
≫ find( (x > 5) & (x < 8) )
```

We can find the number of elements which satisfy this last property by entering

```
≫ length( find( (x > 5) & (x < 8) ) )
```

Previously, we showed how to replace all the elements of `F` which are ≤ 0.5 by $-\pi$. A method which does not require any multiplication is

```
≫ ijF = find(F <= 0.5);
≫ F(ijF) = -pi
```

or even

```
≫ F( find(F <= 0.5) ) = -pi
```

The "beauty" of MATLAB commands such as these is they are so easy to use and to understand (once you get the hang of it) and they require so few keystrokes.

Another, slightly different method uses the matrix

```
≫ D = (F <= 0.5)
```

rather than the vector `ijF`. Recall that `ijF` is a vector which contains the actual locations of the elements we want to zero out, whereas `D` is a matrix of ones and zeroes which explicitly shows which elements should be zeroed. We can use `D` to determine which elements of `F` should be replaced by zero by

```
≫ F(D) = -pi
```

(We can even use

```
≫ F(F <= 0.5) = -pi
```

to combine everything into a single statement.) This requires some explanation. The matrix `D` is being used here as a "mask" to determine which elements of `F` should be replaced by $-\pi$: for every element of `D` which is nonzero, the corresponding element of `F` is replaced by $-\pi$; for every element of `D` which is zero, nothing is done.

How does MATLAB know that `D` should be used to "mask" the elements of `F`? The answer is that `D` is a *logical* matrix because it was defined using a logical operator, and only logical matrices and vectors can be used as "masks". To see that `D` is a logical variable and `F` is not, enter

```
≫ islogical(D)
≫ islogical(F)
```

And to see what happens when you try to use a non-logical variable as a "mask", enter

```
≫ F(2*D)
```

We can also convert a non-logical variable to a logical one by using the MATLAB command `logical`.

To explain logical arrays more clearly, we take a specific and very simple example. Enter

```
≫ v = [0:.25:1];
≫ c = (v >= .5);
```

so that `v = [0 .25 .5 .75 1.0]` and `c = [0 0 1 1 1]` where "0" denotes false and "1" denotes true. The result of

```
≫ v(c)
```

is `[.5 .75 1.0]`. That is, `c` is a logical vector and `v(c)` deletes the elements of `v` which are "false". On the other hand

```
≫ iv = find(v < .5);
```

returns `iv = [1 2]` and

```
≫ v(iv) = [];
```

returns `v = [.5 .75 1.0]`. The difference between `c` and `iv` is that `c` is a logical vector and `iv` is a scalar vector. If you enter

```
≫ v([0 0 1 1 1])      % WRONG
```

instead of

```
≫ v(c)
```

you obtain the error message

```
??? Subscript indices must either be real positive integers or logicals.
```

because `[0 0 1 1 1]` is a numeric vector and so must contain the numbers of the elements of `v` which are desired — but there is no element "0".

MATLAB also has two functions that test vectors and matrices for logical conditions. The command

```
≫  any(x)
```

returns 1 if *any* element of the vector `x` is nonzero (i.e., "TRUE"); otherwise 0 is returned. When applied to a matrix, it operates on each column and returns a row vector. For example, we can check whether or not a matrix is tridiagonal by

```
≫  any( any( triu(A, 2) + tril(A, -2) ) )
```

Here we check all the elements of `A` except those on the main diagonal and on the two adjacent ones. A result of 1 means that at least one other element is nonzero. If we want a result of 1 to mean that `A` is tridiagonal we can use

```
≫  ~any( any( triu(A, 2) + tril(A, -2) ) )
```

instead. The command

```
≫  any(A)
```

operates columnwise and returns a row vector containing the result of `any` as applied to each column.

The complementary function `all` behaves the same as `any` except it returns 1 if *all* the entries are nonzero (i.e., "TRUE"). For example, you can determine if a matrix is symmetric by

```
≫  all( all(A == A.') )
```

A result of 1 means that `A` is identical to $\mathtt{A}^{\mathrm{T}}$.

For completeness we mention that MATLAB has a number of other functions which can check the status of variables, the status of the elements of vectors and matrices, and even of their existence. For example, you might want to zero out all the elements of a matrix `A` which are `Inf` or `NaN`. This is easily done by

```
≫  A( find( ~isfinite(A) ) ) = 0
```

where `isfinite(A)` generates a matrix with 1 in each element for which the corresponding element of `A` is finite. To determine if the matrix `A` even exists, enter

```
exist('A')
```

See the table below for more details and more functions.

Logical Functions

`all`	True if all the elements of a vector are true; operates on the columns of a matrix.
`any`	True if any of the elements of a vector are true; operates on the columns of a matrix.
`exist('<name>')`	False if this name is not the name of a variable or a file. If it is, this function returns: 1 if this is the name of a variable, 2 if this is the name of an m-file, 5 if this is the name of a built-in MATLAB function.
`find`	The indices of a vector or matrix which are nonzero.
`logical`	Converts a numeric variable to a logical one.
`ischar`	True for a character variable or array.
`isempty`	True if the matrix is empty, i.e., `[]`.
`isfinite`	Generates a matrix with 1 in all the elements which are finite (i.e., not `Inf` or `NaN`) and 0 otherwise.
`isinf`	Generates a matrix with 1 in all the elements which are `Inf` and 0 otherwise.
`islogical`	True for a logical variable or array.
`isnan`	Generates a matrix with 1 in all the elements which are `NaN` and 0 otherwise.

8.3. Function M-files

We have already discussed script m-files, which are simply an easy way to collect a number of statements and execute them all at once. *Function m-files*, on the other hand, are similar to functions or procedures or subroutines or subprograms in other programming languages. Ordinarily, variables which are created in a function file exist only inside the file and disappear when the execution of the file is completed — these are called *local variables*. Thus you do not need to understand the internal workings of a function file; you only need to understand what the input and output arguments represent.

Note: The generic term for script files and function files is *m-files*, because the extension is "`m`".

Unlike script files, function files must be constructed in a specific way. The first line of the file `<file name>.m` must begin with the keyword `function`. Without this word, the file is a script file. The complete first line, called the *function definition line*, is

```
function <out> = <function name>(<in 1>, ..., <in n>)
```

or

```
function [<out 1>, ..., <out m>] = <file name>(<in 1>, ..., <in n>)
```

where the name of the function must be the same as the name of the file (but without the extension). The input arguments are `<in 1>`, `<in 2>`, ... The output arguments must appear to the left of the equal sign: if there is only one output argument, i.e., `<out>`, it appears by itself; if there is more than one, i.e., `<out 1>`, etc., they must be separated by commas and must be enclosed in square brackets.

Variables in MATLAB are stored in a part of memory called a *workspace*. The *base workspace* contains all the variables created during the interactive MATLAB session, which includes all variables created in script m-files which have been executed. Each function m-file contains its own *function workspace* which is independent of the base workspace and every other function workspace. The only way to "connect" these workspaces is through the arguments of a function or by using the `global` command (which we will discuss shortly).

There is great flexibility in the number and type of input and output arguments; we discuss this topic in great detail later. The only detail we want to mention now is that the input arguments are all passed "by value" as in C. (That is, the values of the input arguments are stored in temporary variables which are local to the function.) Thus, the input arguments can be modified in the function without affecting any input variables in the calling statement.[†]

Warning: *The name of the file and the name of the function must agree. This is also the name of the command that executes the function.*

Comment lines should immediately follow. A comment line begins with the percent character, i.e., "%". All comment lines which immediately follow the function definition line constitute the documentation for this function; these lines are called the *online help entry* for the function. When you type

```
≫  help <function name>
```

all these lines of documentation are typed out. If you type

```
type <function name>
```

the entire file is printed out. In addition, the first line of documentation, i.e., the second line of the file, can be searched for keywords by entering

```
≫  lookfor <keyword>
```

Make sure this first comment line contains the name of the command and important keywords which describe its purpose.

Note: Comments can be placed anywhere in an m-file, including on a line following a MATLAB statement. The initial comment lines in a script file and the comment lines in a function file which

[†]If you are worried because passing arguments by value might drastically increase the execution time of the function, we want to reassure you that this does not happen. To be precise, MATLAB does not actually pass all the input arguments by value. Instead, an input variable is only passed by value if it is modified by the function. If an input variable is not modified, it is passed "by reference". (That is, the input argument is the actual variable used in the calling statement and not a local copy.) In this way you get the benefit of "call by value" without any unnecessary overhead. And how does MATLAB know if an input argument is modified? It can only be modified if it appears on the left-hand side of an equal sign inside the function!

immediately follow the first line are special: they appear on the screen when you type

```
>> help <function name>
```

Before discussing functions at *great* length, there is one technical detail it is important to consider before it trips you up: how does MATLAB find the m-files you have created? Since MATLAB contains *thousands* of functions, this is not an easy task. Once MATLAB has determined that the word is not a variable, it searches for the function in a particular order. We show the order here and then discuss the items in detail throughout this subsection.

(1) It checks if `<function name>` is a built-in function (i.e., coded in C).
(2) It checks if `<function name>` is a function, i.e., the primary function, a subfunction, or a nested function in the current scope, in the current file. (We discuss all these terms shortly.)
(3) It checks if the file `<function name>.m` exists in the current directory.
(4) It checks if the current directory has a subdirectory called "private"; if it does, MATLAB checks if the file `<function name>.m` exists in this subdirectory.
(5) It searches the directories in the *search path* for the file `<function name>.m`.

Note from (3) that MATLAB searches in the current directory for the function by searching for the m-file with the same name. If the m-file is not in the current directory, the simplest way to enable MATLAB to find it is have the subdirectory in your search path. If you type

```
>> path
```

you will see all the directories that are searched. If you have created a subdirectory called "matlab" in your main directory, this is usually the first directory searched (unless the search path has been modified). Thus, you can put your m-files in this subdirectory and be sure that MATLAB will find them. You can also add directories to the search path by

```
>> path('new_directory', path)
```

or

```
>> path(path, 'new_directory')
```

(The former puts "new_directory" at the beginning of the search path while the latter puts it at the end.)

Warning: When you begin a MATLAB session, it always checks if the subdirectory "matlab" exists in your main directory. If you create this subdirectory after you start a MATLAB session, it will not be in the search path.

Now we return to our discussion of creating functions. We begin with a simple example of a function file which constructs the Hilbert matrix (which we have already used a number of times).

```
function H = hilb_local(n)
% hilb_local: Hilbert matrix of order n (not from MATLAB)
% hilb_local(n) constructs the n by n matrix with elements 1/(i+j-1).
% This is one of the most famous examples of a matrix which is
% nonsingular, but which is very badly conditioned.
H = zeros(n);
for i = 1:n
   for j = 1:n
      H(i,j) = 1/(i+j-1);
   end
end
```

The input argument is `n` and the output argument is `H`. The first line of the documentation includes the name of the function as well as a brief description that `lookfor` uses. The following lines of documentation also appear on the screen if we enter

```
>> help hilb_local
```

Note: The above code is not presently used in MATLAB (although it was in early versions.) The actual MATLAB code for this function is shown in subsection 8.5.

We follow by defining `H` to be an $n \times n$ matrix. Although not essential, this statement can greatly increase the speed of the function because space can be preallocated for the matrix. For example, consider

the following code.

```
function a = prealloc(n, which)
% prealloc: testing how well preallocating a vector works
% n = the size of the vector
% which = 1 - preallocate the vector
%       = 2 - do not
if which == 1
   a = zeros(n,1);
end
a(1) = 1;
for i = 2:n
   a(i) = a(i-1) + 1;
end
```

If `which = 0` the vector `a` is not preallocated, while if `which = 1` it is. We find that

```
≫  prealloc(100000, 1)
```

runs over 4000 (that's right, four thousand) times as fast as

```
≫  prealloc(100000, 0)
```

Note that `i` and `j` are redefined from $\sqrt{-1}$ since they appear as `for` loop indices. However, since `i` and `j` are local to this function, this does not have any effect when this command is executed. Also, the variable `H` is local to the function. If we type

```
≫  Z = hilb_local(12)
```

then the matrix `Z` contains the Hilbert matrix and `H` is undefined.

Normally functions are completed when the end of the file is reached (as above). If the flow control in a function file is complicated enough, this might be difficult to accomplish. Instead, you can use the `return` command, which can appear anywhere in the function and force an immediate end to the function. In addition, you can force the function to abort by entering

```
error(<string>)
```

If the string is not empty, the string is displayed on the terminal and the function is aborted; if the string is empty, the statement is ignored.

One feature of function files which is occasionally very useful is that they can have a variable number of input and output variables. For example, the norm of a vector `x` can be calculated by entering

```
≫  norm(x, p)
```

if $p = 1$, 2, or inf or, more simply, by

```
≫  norm(x)
```

if $p = 2$. Similarly, if only the eigenvalues of a matrix $\texttt{A} \in \mathbb{C}^{n\times n}$ are desired, enter

```
≫  eigval = eig(A)
```

However, if both the eigenvalues and eigenvectors are desired, enter

```
≫  [V, D] = eig(A)
```

where $\texttt{D} \in \mathbb{C}^{n\times n}$ is a diagonal matrix containing the n eigenvalues on its diagonal and the corresponding eigenvectors are found in the same columns of the matrix $\texttt{V} \in \mathbb{C}^{n\times n}$.

Note: On the right side of an equation, `[V D]` or `[V, D]` is the matrix whose initial columns come from `V` and whose final columns come from `D`. This requires that `V` and `D` be matrices which have the same number of rows. On the left side, `[V, D]` denotes the two output arguments which are returned by a function. `V` and `D` can be completely different variables. For example, one can be a character variable and the other a matrix.

MATLAB can also determine the number of input and output arguments: `nargin` returns the number of input arguments and `nargout` returns the number of output arguments. For example, suppose we want to create a function file which calculates

$$f(x, \xi, a) = e^{-a(x-\xi)^2} \sin x \, .$$

We can "spruce" this function up to have default values for ξ and a and also to calculate its derivative

with the following function file.

```
function [out1, out2] = spruce(x, xi, a)
% spruce: a silly function to make a point, f(x,b,a) = sin(x)*exp(-a*(x-b)^2)
% if only x is input, xi = 0 and a = 1
% if only x and xi are input, a = 1
% if only one output argument, f(x,xi,a) is calculated
% if two output arguments, f(x,xi,a) and f'(x,xi,a) are calculated
if nargin == 1
   xi = 0;
   a = 1;
elseif nargin == 2
   a = 1;
end
out1 = exp(-a.*(x-xi).^2).*sin(x);
if nargout == 2
   out2 = exp(-a.*(x-xi).^2).*(cos(x) - 2.*a.*(x-xi).*sin(x));
end
```

If there is only one input argument then ξ is set to 0 and a is set to 1 (which are useful default values) while if there are only two input arguments then a is set to 1. If there is only one output argument then only $f(x)$ is calculated, while if there are two output arguments then both $f(x)$ and $f'(x)$ are calculated.

Also, note that x can be a scalar (i.e., a single value) or it can be a vector. Similarly, ξ and a can each be a scalar or a vector. If x is a vector, i.e., $(x_1, x_2, \ldots, x_n)^{\mathrm{T}}$, while ξ and a are scalars, then the function is

$$f(x_i, \xi, a) = \sin(x_i) e^{-a(x_i - \xi)^2} \quad \text{for } i = 1, 2, \ldots, n \,,$$

and all the values can be calculated in one call to `spruce`. If, on the other hand, x, ξ, and a are all vectors, then the function is

$$f(x_i, \xi_i, a_i) = \sin(x_i) e^{-a_i(x_i - \xi_i)^2} \quad \text{for } i = 1, 2, \ldots, n \,,$$

and, again, all the values can be calculated in one call to `spruce`.

We have now presented all the essential features of the MATLAB programming language, and it certainly is a "minimal" language. MATLAB can get away with this because most matrix operations can be performed directly — unlike in most other programming languages. You only need to write your own function if MATLAB cannot already do what you want. If you want to become proficient in this language, simply use the `type` command to look at the coding of some functions.

Now that we have discussed the essentials of programming in MATLAB, it is necessary to discuss how to program *correctly*. When you are entering one statement at a time in the text window, you immediately see the result of your calculation and you can determine whether or not it is correct. However, in an m-file you have a sequence of statements which normally end with semicolons so that you do not see the intermediate calculations. What do you do if the result is incorrect? In other words, how do you debug your m-file?

There are a number of simple techniques you can use and we discuss them in turn. In a script m-file intermediate calculations are normally not printed out, but they are still available to look at. This can frequently lead to an understanding of where the calculation first went wrong. However, this is not true of function m-files since all the local variables in the function disappear when the function ends. Of course, with any m-file you can selectively remove semicolons so that intermediate results are printed out. This is probably the most common method of debugging programs — no matter what programming language is being used.

When loops are involved (either using `for` or `while` statements), the printed output can seem to be **endless** (and it *is* endless if you are in an infinite loop). And it is usually impossible to read the output since it is zipping by at (what appears to be) nearly the speed of light! The `pause` command can slow down or even stop this output. By itself `pause` stops the m-file until some key is pressed while

`pause(<floating-point number>)` pauses execution for this many seconds — or fractions of a second. (This is computer dependent, but `pause(0.01)` should be supported on most platforms.) You can even turn these pauses on and off by using `pause on` and `pause off`.

The `echo` command is also useful for debugging script and function files, especially when `if` statements are involved. Typing

```
≫  echo on
```

turns on the echoing of statements in all script files (but not printing the results if the statements end with semicolons), and `echo off` turns echoing off again. However, this does not affect function files. To turn echoing on for a particular function, type

```
≫  echo <function name> on
```

and to turn echoing on for all functions, type

```
≫  echo on all
```

The `keyboard` command is also very useful for debugging m-files. It stops execution of the m-file, similar to the `pause` command. However, it returns complete control to the user to enter any and all MATLAB commands. In particular, you can examine any variables in the function's workspace. If desired, you can also change the value of any of these variables. The only way you will recognize this is not a "standard" MATLAB session is that the prompt is

```
K≫
```

for Keyboard. To terminate the "keyboard" session and return control to the m-file, enter

```
K≫  return
```

To terminate both the "keyboard" session and the execution of the m-file, enter

```
K≫  dbquit
```

When using the debugger, you are not running your program. Instead, you are running the debugger which is running your program. Thus, many of the commands you enter are commands for the debugger; to distinguish these commands from "normal" MATLAB commands they begin with `db`. There are two ways to run the debugger: you can type the debugger commands into the workspace, or in a MATLAB window you can use the mouse and click on the commands.

In addition, you are still in the workspace so many MATLAB commands can still be executed. For example, to see the values of variables just type the variable name into the workspace. (Alternatively, you can move the mouse over the variable name in the window and its description and value(s) will be shown.) Also, you can run most MATLAB commands as long as you do not try to create new variables (but you can modify existing variables).

We will not discuss the commands in this debugger in detail, but only provide a brief description of each one, because these are similar to commands in any debugger. If you have experience with using a debugger, `help` or `doc` will give you complete details.

Debugging Commands

`keyboard`	Turns debugging on.
`dbstep`	Execute the next executable line.
`dbstep` n	Execute the next n lines.
`dbstep in`	The same as `dbstep` except that it will step into another m-file (rather than over it).
`dbstep out`	Executes the remainder of the current function and stops afterwards.
`dbcont`	Continue execution.
`dbstop`	Set a breakpoint.
`dbclear`	Remove a breakpoint.
`dbup`	Change the workspace to the calling function or the base workspace.
`dbdown`	Change the workspace down to the called function.
`dbstack`	Display all the calling functions.
`dbstatus`	List all the breakpoints.
`dbtype`	List the current function, including the line numbers.
`dbquit`	Quit debugging mode and terminate the function.
`return`	Quit debugging mode and continue execution of the function.

Now we want to discuss the arguments in a MATLAB function, since they are are used somewhat differently than in other programming languages. For example, in

```
function out = funct1(a, t)
```

`a` and `t` are the input arguments and `out` is the output argument. Any and all input variables are local to the function and so can be modified without affecting the arguments when the function `funct1` is called. (This is true no matter what type of variables they are.) In

```
function [out1, out2, out3] = funct2(z)
```

`z` is the only input argument and there are three output arguments, each of which can be any type of variable. There is no requirement that all three of these output arguments actually be used. For example, the calling statement might be any of the following:

```
≫  art = funct2(1.5)
≫  [physics, chemistry] = funct2([1 2 3])
≫  [math, philosophy, horticulture] = funct2(reshape([1:30], 6, 5))
```

(just to be somewhat silly).

In a programming language such as C, Fortran, or Pascal `funct1` would be written in a form such as

```
function funct1(a, t, out)
```

or even as

```
function funct1(a, out, t)
```

Similarly, `funct2` would be written as

```
function funct2(z, out1, out2, out3)
```

or even as

```
function funct2(out3, out2, z, out1)
```

It would be up to the user to control which were the input arguments, which were the output arguments, and which were both (i.e., one value on input and another on output). In MATLAB input arguments occur on the right side of the equal sign and output arguments occur on the left. Arguments which are to be modified by the function must occur on both sides of the equal sign in the calling statement. For example, in `funct2` if `z` is modified and returned in `out1` then the calling sequence should be

```
≫  [a, b, c] = funct2(a)
```

where `a` appears on both sides of the equal sign. (There is an alternative to this awkward use of parameters which are modified by the function: you can make a variable global, as we discuss at the end of this section. However, this is not usually a good idea.)

There is another difference between MATLAB and most other programming languages where the type of each variables has to be declared, either explicitly or implicitly. For example, a variable might be an integer, a single-precision floating-point number, a double-precision floating-point number, a character string, etc. In MATLAB, on the other hand, there is no such requirement. For example, the following statements can follow one another in order and define `x` to be a string variable, then a vector, then a scalar, and finally a matrix.

```
≫  x = 'WOW?'
≫  x = x + 0
≫  x = sum(x)
≫  x = x*[1 2; 3 4]
```

It is particularly important to understand this "typelessness" when considering output arguments. For example, there are three output arguments to `funct2` and any of them can contain any type of variable. In fact, you can let the type of these arguments depend on the value or type of the input argument. This is probably not something you should want to do frequently, but it is sometimes very useful.

Occasionally, there is a need to pass values from the workspace to a function or to pass values between different functions without using the input arguments. (As we discussed earlier, this may be desirable if a variable is modified by a function.) In C this is done by using global variables. MATLAB also has global variables which are defined by declaring the variables to be global using

```
≫  global <variable 1> <variable 2> <variable 3> ...
```

By the way, a variable is a *global variable* if it appears in a global statement and a *local variable* if it does not. (Note that a variable can be a local variable in one function and a global variable in another.)

Warning: Spaces, not commas, must separate the variables in a `global` statement

This statement must appear in every function which is to share the variables. If the workspace is also to share these variables, you must type this statement (or be put into a script file which you execute) before

these variables are used.

Instead of using a global variable, it is frequently preferable to save the value of a local variable between calls to the function. Normally, local variables come into existence when the function is called and disappear when the function ends. Sometimes it is very convenient to be able to "save" the value of a local variable so that it will still be in existence when the function is next called. In C, this is done by declaring the variable `static`. In MATLAB it is done by declaring the variable `persistent` using

```
≫  persistent <variable 1> <variable 2> <variable 3> ...
```

Warning: Spaces, not commas, must separate the variables.

Note: The first time you enter the function, a persistent variable will be empty, i.e., `[]`, and you can test for this by using `isempty`.

We now present a simple example where persistent variables are very helpful. Suppose we want to write a function m-file to evaluate

$$\mathtt{z(y)} = \begin{pmatrix} y_2 \\ y_1(1-\beta y_1^2) - \alpha y_2 + \Gamma\cos\omega t \end{pmatrix}$$

where α, β, Γ, and ω are parameters which will be set initially and then left unchanged during a run. We might be studying a mathematical model where this function will be evaluated many, many times for different values of `y`. For each experiment these parameters will be fixed, but they will be different for each experiment. We do not want to "hardcode" the values in the function because we would have to repeatedly change the function — which is very undesirable. Certainly we can write the function as

```
function z = fncz1(y, alpha, beta, Gamma, omega)
z = [ y(2) ; -y(1)*(1-beta*y(1)^2)-alpha*y(2)+Gamma*cos(omega*t) ];
```

but then we have to include these four parameters in each call to the function. We can always simplify this function by combining the four parameters into one vector by

```
function z = fncz2(y, par)
% alpha = par(1), beta = par(2), Gamma = par(3), omega = par(4)
z = [ y(2) ; -y(1)*(1-par(2)*y(1)^2)-par(1)*y(2)+par(3)*cos(par(4)*t) ];
```

but then it is harder to read the equation. (If this function was more complicated it would be *much* harder to read.) To make this last function easier to read we could write it as

```
function z = fncz3(y, par)
alpha = par(1)
beta = par(2)
Gamma = par(3)
omega = par(4)
z = [ y(2) ; -y(1)*(1-beta*y(1)^2)-alpha*y(2)+Gamma*cos(omega*t) ];
```

but we want to propose another alternative.

There are many reasons that evaluating `z(y)` using `fncz1`, `fncz2`, or `fncz3` might not be desirable — or even practical. As one example, this function might be called repeatedly inside a general purpose function m-file, say `general`, which we have written. In `general` we only want to call the function as `z = fnctn(y)` and not have to worry about how parameters are passed to the function. If we instead write the function m-file for `z(y)` as

```
function z = fncz4(y, alpha, beta, Gamma, omega)
persistent alpha_p beta_p Gamma_p omega_p
if nargin > 1
   alpha_p = alpha;
   beta_p = beta;
   Gamma_p = Gamma;
   omega_p = omega;
end
z = [ y(2) ; -y(1)*(1-beta_p*y(1)^2)-alpha_p*y(2)+Gamma_p*cos(omega_p*t) ];
```

we will *initially* call the function as `fncz4(y, alpha, beta, Gamma, omega)` and then afterwards call it as `fncz2(y)`. In the initial call all the parameters are saved in persistent variables. Later calls do not need to input these parameters because they have been saved in the function. That is, we would call `general` by

```
fncz4(y, alpha, beta, Gamma, omega);
general('fncz4', ...)
```

where `general` would be written as

```
function general(fnctn, ...)
...
z = feval(fnctn, y)
...
```

The function `feval(fnctn,y)`, which is discussed in the next section, means `fnctn(y)`. It is used when the name of a function has been passed as an argument into another function. Our discussion of saving parameters in function has been somewhat lengthy, but it has many uses.

The final technical detail about function m-files concerns an important element of programming style in any computer language. It frequently happens that a block of code occurs two or more times in a function. Sometimes these blocks can be combined by using a loop, but, even if possible, this often makes the code unwieldly. Instead, this block of code can be put into a new function and called from the original function. Another reason for splitting a block of code off into a new function is when the function has grown large enough to be hard to comprehend. The remedy is to split the code up into a number of functions, each of which can be easily understood and debugged. In MATLAB functions normally have to be separated into different files so that each function and its file name agree; otherwise, MATLAB cannot find the function. This can be annoying if a number of files have to be created: for example, it can be difficult to remember the purpose of all these functions, and it can be difficult to debug the primary function. MATLAB has a feature to handle this proliferation of files; function m-files can contain more than one function. The first function in the file is called the *primary function* and its name must agree with the name of the file. There are two further types of functions: Any remaining functions are called *subfunctions* and *nested functions*. (At the end of subsection 10.2 we code the function `gravity` using a number of nested functions.)

Note: The primary function or a subfunction begins with the function definition line (i.e., the line which begins with the keyword `function`). It is possible to end the primary function and each subfunction with the command `end`, but this is not necessary because MATLAB recognizes that a function has ended when it encounters the next `function` statement. However, if a nested function is used then it — and all other functions — must end with the `end` statement.

First, we discuss subfunctions, which are quite simple. They are placed following the primary function and between or following other subfunctions. They are only visible to the primary function and to other subfunctions in the same file. Thus, different m-files can contain subfunctions with the same name. Also, the `help`, `lookfor`, and `type` commands can only access the primary file. It is crucial to understand that variables in the primary function or in a subfunction are local to that function and unknown outside it (unless they are declared to be `global`). The only way to pass variables between these functions is through the argument list.

Usually, subfunctions are sufficient — and they are much easier to describe. When they are not sufficient, we have *nested functions*. Its main advantage (as far as we are concerned) is that variables can be passed into and out of a nested function without being in the argument list. Nested functions are more complicated than subfunctions and we will only provide a brief discussion.[†]

To make this discussion specific, consider the following function m-file.

[†]They are similar to internal functions in Fortran 95, and they are somewhat related to internal classes in Java — but not in C++.

```
function [p1, p2, p3] = nested_ex(x, y, z) % 1
p1 = x;                                    % 2
p2 = y;                                    % 3
nest_1;                                    % 4
   function nest_1                         % 5
   n1a = p1 + z;                           % 6
   n1b = p2 + z;                           % 7
   p1 = n1a + n1b;                         % 8
   p2 = n1a*n1b;                           % 9
   p3 = sub_1(p1, p2);                     % 10
   nest_2;                                 % 11
   end                                     % 12
   function nest_2                         % 13
   n2a = p1;                               % 14
   p1 = n2a^2;                             % 15
   p2 = p2^2;                              % 16
   % p3 = n1a;     % WRONG                 % 17
   end                                     % 18
end                                        % 19
function s3 = sub_1(a, b)                  % 20
s1 = 10;                                   % 21
s2 = a + b;                                % 22
s3 = s2 + nest_3(s1);                      % 23
   function n3b = nest_3(n3a)              % 24
   n3b = s1^2 + n3a;                       % 25
   end                                     % 26
end                                        % 27
```

A nested function is within another function. For example, the nested functions `nest_1` and `nest_2` are nested within the primary function `nested_ex`, and the nested function `nest_3` is nested within the subfunction `sub_1`. (Nested functions can have other nested functions within them, but enough is enough.)

The important concept to understand when using nested functions is the *scope* of variables in the function m-file. The scope of a variable is the context within which it is defined, i.e., where it can be set, modified, and used. Now let us consider a function workspace. The workspace of the primary function is also independent of the workspace of each subfunction. However, since a nested function is *within* one or more other functions, it is within the workspace of this function or these functions. In the function `nested_ex` the nested functions `nest_1` (lines 5–12) and `nest_2` (lines 13–19) have access to the variables `p1`, `p2`, and `p3` of the primary function (lines 1–20). They also have access to the subfunction `sub_1` (lines 20–27) as shown in line 10. Note that `nest_2` also has access to `nest_1` as shown in line 11. It is *very* important to understand that a nested function also has its own workspace. For example, the variables `n1a` and `n1b` in `nest_1`, `n2a` in `nest_2`, and `n3a` and `n3b` in `nest_3` are local to their respective functions. In particular, `n1a` cannot be accessed in `nest_2` as shown in line 17. (If `n1a` really needs to be passed to `nest_2`, then it must be in the workspace of `nested_ex`. This could be done by adding `n1a = 0` after line 3.) Finally, note that nested functions can also have arguments passed in the argument list as in `nest_3` on line 24.

Now let us return to the topic of how MATLAB finds a function. As we stated previously (but did not discuss), when a function is called from within an m-file, MATLAB first checks if the function named is the primary function or a subfunction in the current file. If it is not, MATLAB searches for the m-file in the current directory. Then MATLAB searches for a private function by the same name (described below). Only if all this fails does MATLAB use your search path to find the function. Because of the way that MATLAB searches for functions, you can replace a MATLAB function by a subfunction in the current

m-file — but make sure you have a good reason for doing so![†]

In the previous paragraph we described how to create a subfunction to replace one function by another of the same name. There is another, more general, way to handle this replacement: you can create a subdirectory in your current directory with the special name "private". Any m-files in this subdirectory are visible only to functions in the current directory. The functions in this subdirectory are called *private functions*. For example, suppose we are working in the directory "personal" and have created a number of files which use `rref` to solve linear systems. And suppose we have written our own version of this command, because we think we can calculate the reduced row echelon of a matrix more accurately. The usual way to test our new function would be to give it a new name, say `myrref`, and to change the call to `rref` in every file in this directory to `myrref`. This would be quite time-consuming, and we might well miss some. Instead, we can code and debug our new function in the subdirectory "private", letting the name of our new function be `rref` and the name of the m-file be `rref.m`. All calls in the directory to `rref` will use the new function we are testing in the subdirectory "private", rather than MATLAB's function. Even more important, any function in any other directory which calls `rref` will use the MATLAB function and not our "new, improved version".

Function Commands

`function`	Begins a MATLAB function.
`end`	Ends a function. This statement is only required if the function m-file contains a nested function, in which case it must be used by *all* the functions in the file.
`error('<message>')`	Displays the error message on the screen and terminates the m-file immediately.
`echo`	Turns echoing of statements in m-files on and off.
`global`	Defines a global variable (i.e., it can be shared between different functions and/or the workspace).
`persistent`	Defines a local variable whose value is to be saved between calls to the function.
`keyboard`	Stops execution in an m-file and returns control to the user for debugging purposes. The command `return` continues execution and `dbquit` aborts execution.
`return`	Terminates the function immediately.
`nargin`	Number of input arguments supplied by the user.
`nargout`	Number of output arguments supplied by the user.
`pause`	Halts execution until you press some key.

8.4. Odds and Ends

In MATLAB it is possible for a program to create or modify statemennts "on the fly", i.e., as the program is running. Entering

```
≫  eval(<string>)
```

executes whatever statement or statements are contained in the string. For example, entering

```
≫  s = 'x = linspace(0, 10, n); y = x.*sin(x).*exp(x/5); plot(x, y)'
≫  eval(s)
```

[†]Since MATLAB contains *thousands* of functions, this means you do not have to worry about one of your subfunctions being "hijacked" by an already existing function. When you think up a name for a primary function (and, thus, for the name of the m-file) it is important to check that the name is not already in use. However, when breaking a function up into a primary function plus subfunctions, it would be very annoying if the name of every subfunction had to be checked — especially since these subfunctions are not visible outside the m-file.

executes all three statements contained in the string `s`. In addition, if an executed statement generates output, this is the output of `eval`. For example, if we type

```
≫  A = zeros(5,6);
≫  [m, n] = eval('size(A)');
```

then `m` is 5 and `n` is 6.

There is a very practical applications for this command since it can can combine a number of statements into one. For example, suppose we want to work with the columns of the Hilbert matrix of size `n` and we want to create variables to hold each column, rather than using `H(:,i)`. We can do this by hand by typing

```
≫  c1=H(:,1);
≫  c2=H(:,2);
   ...
```

which gets tiring very quickly. Instead, we can do this by typing

```
≫  for i = 1:n
      eval( ['c' num2str(i) '=H(:,i)'] )
   end
```

This requires some explanation. It might be a little clearer if we separate the statement inside the `for` loop into two statements by

```
      s = ['c', num2str(i), '=H(:,i)']
      eval(s)
```

(where we include commas in the first statement for readability). `s` is a text variable which contains `c1=H(:,1)` the first time the loop is executed, then `c2=H(:,2)` the second time, etc. (To understand how `s` is created, recall that `s` is really just a row vector with each element containing the ASCII representation of the corresponding character.)

Finally, there is a very esoteric application for this command that allows it to catch errors. This is similar to the "catch" and "throw" commands in C++ and Java. To use this feature of `eval`, call it using two arguments as

```
≫  eval(<try_string>, <catch_string>)
```

The function executes the contents of `<try_string>` and ignores the second argument if this execution succeeds. However, if it fails then the contents of `<catch_string>` are executed. (This might be a call to a function which can handle the error.) If there is an error, the command `lasterr` returns a string containing the error message generated by MATLAB.

A MATLAB command which is occasionally useful in a function is `feval`. It executes a function, usually defined by an m-file, whose name is contained in a string by

```
≫  feval(<string>, x1, x2, ..., xn)
```

(See below for other ways to pass the function in the argument list.) Here `x1, x2, ..., xn` are the arguments to the function. For example, the following two statements are equivalent

```
≫  A = zeros(5,6)
≫  A = feval('zeros', 5, 6)
```

Suppose that in the body of one function, say `sample`, we want to execute another function whose name we do not know. Instead, the name of the function is to be passed as an argument to `sample`. Then `feval` can be used to execute this text variable. For example, suppose in function `sample` we want to generate either linear or logarithmic plots. We can input the type of plot to use by

```
function sample(<type of plot>)
...
feval(<type of plot>, x, y1, x, y2, '--', xx, y3, ':')
...
```

There are two common ways to pass the function `<type of plot>` in the argument list:
(1) use a character string, e.g., `feval('loglog', ...)`, or
(2) use a function handle, e.g., `feval(@logval, ...)`, or

Note: `eval` and `feval` serve similar purposes since they both evaluate something. In fact, `feval` can always be replaced by `eval` since, for example, `feval('zeros', 5, 6)` can always be replaced by `eval('zeros(5,6)')`. However, there is a fundamental difference between them: `eval` requires

the MATLAB interpreter to completely evaluate the string, whereas `feval` only requires MATLAB to evaluate an already existing function. `feval` is much more efficient, especially if the string must be evaluated many times inside a loop.

Odds and Ends

`eval`	Executes MATLAB statements contained in a text variable. Can also "catch" an error in a statement and try to fix it.
`feval`	Executes a function specified by a string. (Can be used to pass a function name by argument.)
`lasterr`	If `eval` "catches" an error, it is contained here.

8.5. Advanced Topic: Vectorizing Code

As long as your MATLAB code executes "quickly", there is no need to try to make it faster. However, if your code is executing "slowly", you might be willing to spend some time trying to speed it up.† There are three standard methods to speed up a code:

(0) **Preallocate matrices** as shown in the function `prealloc` on page 79. This is very simple and very effective if the matrices are "large".

(1) Use MATLAB functions, whenever possible, rather than writing your own. If a MATLAB function is built-in, then it has been written in C and is faster than anything you can do. Even if it is not, much time has been spent optimizing the functions that come with MATLAB; you are unlikely to do better.

(2) Replace flow control instructions with vector operations. We have already discussed this topic at length in subsection 8.2. Here we will focus on some advanced techniques.

As a simple example of method (0), consider the function `hilb` on page 79. `hilb_local(2000)` runs over 300 times slower if the line `H = zeros(n)` is omitted.

Continuing with this example, currently the MATLAB function `hilb` is written as

```
J = 1:n;     % J is a row vector
J = J(ones(n, 1),:);     % J is now an n by n matrix with each row being 1:n
I = J';     % I is an n by n matrix with each column being 1:n
E = ones(n, n);
H = E./(I+J-1);
```

as you can see by entering

```
≫  type hilb
```

In the past this code ran nearly 20 times as fast as `hilb_local`. However, now `hilb_local` runs 50% faster. The reason is that MATLAB has greatly improved its handling of `for` and `while` statements. Thus, it is frequently not necessary to convert simple loops into complicated vector code.

As a realistic example of method (2), suppose you have a large vector `y` which is the discretization of a smooth function and you want to know some information about it. In particular, consider the intervals in `y` where $y_i > R$. What is the average length of these intervals and what is their standard deviation? Also, only include intervals which lie completely within `y` (i.e., ignore any intervals which begin or end `y`). It is not difficult to write such a code using flow control statements:

†We have put "quickly" and "slowly" in quotes because this is quite subjective. Remember that your time is valuable: if it takes you longer to optimize your code than you will save in running it more quickly, stifle the urge to muck around with it. Also remember that the amount of time it *actually* takes to optimize a code is usually a factor of two or three or ... longer than the time you *think* it will take before you get started.

```
function ylen_intvl = get_intervals_slow(y, R)
n = length(y);
if y(1) > R     % check if the first point is in an interval
   in_intvl = 1;     % yes
   intvl_nr = 1;
   yin(intvl_nr) = 1;
else
   in_intvl = 0;     % no
   intvl_nr = 0;
end
for i = [2: n]     % check the rest of the points
   if in_intvl == 1     % we are currently in an interval
      if y(i) <= R     % check if this point is also in the interval
         yout(intvl_nr) = i;     % no, so end the interval
         in_intvl = 0;
      end
   else     % we are currently not in an interval
      if y(i) > R     % check if this point is in the next interval
         intvl_nr = intvl_nr + 1;     % yes, so begin a new interval
         yin(intvl_nr) = i;
         in_intvl = 1;
      end
   end
end
if y(1) > R     % check if we have begun in an interval
   yin(1) = [];     % yes, so delete it
   yout(1) = [];
end
if length(yin) > length(yout)     % check if we have ended in an interval
   yin( length(yin) ) = [];     % yes, so delete it
end
ylen_intvl = yout - yin;
```

When completed, `yin` and `yout` contain the element numbers where an interval begins and where it ends, respectively. This is straightforward — but **very** slow if `y` has millions of elements.

To write a vectorized code, we have to think about the problem differently:

(1) We do not care about the actual values in `y`, only whether they are greater than R or not. So we construct a logical matrix corresponding to `y` by `yr = (y > R)`.
(2) We do not actually care about the 0's and 1's — only about where the value changes because these mark the boundaries of the intervals. So we take the difference between adjacent elements of `yr` by `yd = diff(yr)`.
(3) We actually only need to know the elements which contain nonzero values so we find the element numbers by `ye = find(yd)`, i.e., `ye = find(yd~=0)`.
(4) We do not care about the actual locations of the beginning and end of each interval, only the lengths of these intervals. So we take the difference again by `ylen = diff(ye)`.
(5) Finally, `ylen` contains the lengths of both the intervals and the distances between successive intervals. So we take every other element of `ylen`. We also have to be a little careful and check whether `y` begins and/or ends in an interval.

Here is the code:

```
function ylen_intvl = get_intervals_fast(y, R)
yr = (y > R);     % (1)
yd = diff(yr);     % (2)
ye = find(yd);     % (3)
ylen = diff(ye);     % (4)
if y(1) > R     % (5), check if we begin in an interval
   ylen(1) = [];     % yes
end
ylen_intvl = ylen( 1:2:length(ylen) );     % get every other length
```

Finally, the question remains: is the time savings significant? For "large" `y` the CPU time is reduced by approximately 50.

9. Sparse Matrices

Many matrices that arise in applications only have a small proportion of nonzero elements. For example, if $\mathtt{T} \in \mathbb{C}^{n\times n}$ is a tridiagonal matrix, then the maximum number of nonzero elements is $3n-2$. This is certainly a small proportion of the total number of elements, i.e., n^2, if n is "large" (which commonly means in the hundreds or thousands or ...)

For *full* matrices (i.e., most of the elements are nonzero) MATLAB stores all the elements, while for *sparse* matrices (i.e., most of the elements are zero) MATLAB only stores the nonzero elements: their locations (i.e., their row numbers and column numbers) and their values. Thus, sparse matrices require much less storage space in the computer. In addition, the computation time for matrix operations is significantly reduced because zero elements can be ignored.

Once sparse matrices are generated, MATLAB is completely responsible for handling all the details of their use: there are no special commands needed to work with sparse matrices. However, there are a number of commands which are inappropriate for sparse matrices, and MATLAB generally generates a warning message and refers you to more appropriate commands. For example, `cond(S)` has to calculate $\mathtt{S}^{-1}$, which is generally a full matrix; instead, you can use `condest` which estimates the condition number by using Gaussian elimination. You have two alternatives: first, use `full` to generate a full matrix and use the desired command; or, second, use the recommended alternative command.

There are two common commands in MATLAB for creating sparse matrices. You can enter all the nonzero elements of $\mathtt{S} \in \mathbb{C}^{m\times n}$ individually by

```
≫  S = sparse(i, j, s, m, n)
```

where `i` and `j` are vectors which contain the row and column indices of nonzero elements and `s` is the vector which contains the corresponding values. For example, the square bidiagonal matrix

$$\mathtt{S} = \begin{pmatrix} n & -2 & & & & \\ & n-1 & -4 & & & 0 \\ & & n-2 & -6 & & \\ & & & \ddots & \ddots & \\ & & & & 2 & -2n+2 \\ 0 & & & & & 1 \end{pmatrix}$$

has the following nonzero elements

i	j	$s_{i,j}$
1	1	n
2	2	$n-1$
3	3	$n-2$
$\vdots$	$\vdots$	$\vdots$
$n-1$	$n-1$	2
n	n	1

i	j	$s_{i,j}$
1	2	-2
2	3	-4
3	4	-6
$\vdots$	$\vdots$	$\vdots$
$n-2$	$n-1$	$-2n+4$
$n-1$	n	$-2n+2$

A simple way to generate this matrix is by entering

```
» S = sparse([1:n], [1:n], [n:-1:1], n, n) + ...
sparse([1:n-1], [2:n], [-2:-2:-2*n+2], n, n)
```

We could, of course, generate `S` using one `sparse` command, but it would be more complicated. The above command is easier to understand, even if it does require adding two sparse matrices. Since the output from this command is basically just the above table, it is difficult to be sure that `S` is precisely what is desired. We can convert a sparse matrix to full by

```
» full(S)
```

and check explicitly that `S` is exactly what is shown in the above matrix.

In addition, a full (or even an already sparse) matrix `A` can be converted to sparse form with all zero elements removed by

```
» S = sparse(A)
```

Finally, a zero $m \times n$ matrix can be generated by

```
» SZ = sparse(m, n)
```

which is short for

```
» SZ = sparse([], [], [], m, n)
```

The second common command for generating sparse matrices is

```
» S = spdiags(B, d, m, n)
```

which works with entire diagonals of `S`. `B` is an $\min\{ m, n \} \times p$ matrix and its *columns* become the diagonals of `S` specified by $\mathtt{d} \in \mathbb{C}^p$. (For example, if $\mathtt{d} = (0, 1)^{\mathrm{T}}$ then the first column of `B` contains the elements on the main diagonal and the second column contains the elements on the diagonal which is one above the main diagonal.) Thus, we can also generate the matrix `S` given above by

```
» B = [ [n:-1:1]'  [0:-2:-2*n+2]' ]
» S = spdiags(B, [0 1]', n, n)
```

Warning: **Be Careful!** The command `spdiags` is somewhat similar to `diag` but must be handled more carefully. Note that the element $b_{1,2}$ is 0, which does not appear in `S`. The difficulty is that the number of rows of `B` is generally larger than the lengths of the diagonals into which the columns of `B` are to be placed and so some padding is required in `B`. The padding is done so that all the elements in the k-th *row* of `B` come from the k-th *column* of `S`.

For example, the matrix

$$\mathtt{S1} = \begin{pmatrix} 0 & 0 & 6 & 0 & 0 \\ 1 & 0 & 0 & 7 & 0 \\ 0 & 2 & 0 & 0 & 8 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 \end{pmatrix}$$

can be generated as a sparse matrix by

```
≫  A = diag([1:4], -1) + diag([6:8], 2)
≫  S1 = sparse(A)
```

or by

```
≫  B = [ [1:4]  0; 0  0  [6:8] ]′
≫  S1 = spdiags(B, [-1  2], 5, 5)
```

In the latter case note that the columns of `B` have to be padded with zeroes so that each column has five elements, whereas in the former case the vector which becomes the particular diagonal precisely fits into the diagonal. The element $s_{1,3}$ of `S1` contains the value 6. It appears in the 3-rd *row* of `B` because it occurs in the 3-rd *column* of `S1`. Note that the element $b_{n,2}$ is not used since it would go into the element $s_{n,n+1}$.

A slight variation of the above command is

```
≫  T = spdiags(B, d, S)
```

where `T` is equated to `S` and then the columns of `B` are placed in the diagonals of `T` specified by `d`. Thus, a third way to generate the matrix `S` given above is

```
≫  S = spdiags([n:-1:1]′, [0], n, n)
≫  S = spdiags([0:-2:-2*n+2]′, [1], S)
```

Just as with the `diag` command, we can also extract the diagonals of a sparse matrix by using `spdiags`. For example, to extract the main diagonal of `S`, enter

```
≫  B = spdiags(S, [0])
```

The number of nonzero elements in the sparse matrix `S` are calculated by

```
≫  nnz(S)
```

(Note that this is not necessarily the number of elements stored in `S` because all these elements are checked to see if they are nonzero.) The locations and values of the nonzero elements can be obtained by

```
≫  [iA, jA, valueA] = find(A)
```

The locations of the nonzero elements is shown in the graphics window by entering

```
≫  spy(S)
```

These locations are returned as dots in a rectangular box representing the matrix which shows any structure in their positions.

All of MATLAB's intrinsic arithmetic and logical operations can be applied to sparse matrices as well as full ones. In addition, sparse and full matrices can be mixed together. The type of the resulting matrix depends on the particular operation which is performed, although usually the result is a full matrix. In addition, intrinsic MATLAB functions often preserve sparseness.

You can generate sparse random patrices by `sprand` and sparse, normally distributed random matrices by `sprandn`. There are a number of different arguments for these functions. For example, you can generate a random matrix with the same sparsity structure as `S` by

```
≫  sprand(S)
```

or you can generate an $m \times n$ matrix with the number of nonzero random elements being approximately ρmn by

```
≫  sprand(m, n, rho)
```

Finally, you can generate sparse random *symmetric* matrices by `sprandsym`; if desired, the matrix will also be positive definite. (There is no equivalent command for non-sparse matrices so use `full(sprandsym(...))`

Additionally, sparse matrices can be input from a data file with the `spconvert` command. Use `csvread` or `load` to input the sparsity pattern from a data file into the matrix `<sparsity matrix>`. This data file should contain three columns: the first two columns contain the row and column indices of the nonzero elements, and the third column contains the corresponding values. Then type

```
≫  S = spconvert(<sparsity matrix>)
```

to generate the sparse matrix `S`. Note that the size of `S` is determined from the maximum row and the maximum column given in `<sparsity matrix>`. If this is not the size desired, one row in the data file should be "`m n 0`" where the desired size of `S` is $m \times n$. (This element will not be used, since its value is zero, but the size of the matrix will be adjusted.)

Sparse Matrix Functions

`speye`	Generates a sparse identity matrix. The arguments are the same as for `eye`.
`sprand`	Sparse uniformly distributed random matrix.
`sprand`	Sparse uniformly distributed random symmetric matrix; the matrix can also be positive definite.
`sprandn`	Sparse normally distributed random matrix.
`sparse`	Generates a sparse matrix elementwise.
`spdiags`	Generates a sparse matrix by diagonals or extracts some diagonals of a sparse matrix.
`full`	Converts a sparse matrix to a full matrix.
`find`	Finds the indices of the nonzero elements of a matrix.
`nnz`	Returns the number of nonzero elements in a matrix.
`spfun('<function name>', A)`	Applies the function to the nonzero elements of `A`.
`spy`	Plots the locations of the nonzero elements of a matrix.
`spconvert`	Generates a sparse matrix given the nonzero elements and their indices.
`sprandsym`	Generates a sparse uniformly distributed symmetric random matrix; the matrix can also be positive definite.

10. Initial-Value Ordinary Differential Equations

Most initial-value ordinary differential equations cannot be solved analytically. Instead, using MATLAB we can obtain a numerical approximation to the ode system

$$\frac{d}{dt}\mathtt{y} = \mathtt{f}(t, \mathtt{y}) \qquad \text{for } t \geq t_0$$

where $\mathtt{y} \in \mathbb{R}^n$ with initial condition $\mathtt{y}(t_0) = \mathtt{y}_0$. The basic MATLAB commands are easily learned. However, the commands become more involved if we want to explore the trajectories in more detail. Thus, we divide this section into the really basic commands which are needed to generate a simple trajectory and into a more advanced section that goes into many technical details. We also provide a large number of examples, many more than in other sections of this overview, to provide a template of how to actually use the advanced features.

10.1. Basic Commands

In this subsection we focus on the particular example

$$y'' + \alpha y' - y(1 - \beta y^2) = \Gamma \cos \omega t \,,$$

which is called *Duffing's equation*. This ode has many different types of behavior depending on the values of the parameters α, β, Γ, and ω.

As written, this is not in the form of a first-order system. To transform it we define $y_1 = y$ and $y_2 = y_1' = y'$ so that

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= y_1'' = y'' = y_1(1 - \beta y_1^2) - \alpha y_2 + \Gamma \cos \omega t \end{aligned}$$

or

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}' = \begin{pmatrix} y_2 \\ y_1(1 - \beta y_1^2) - \alpha y_2 + \Gamma \cos \omega t \end{pmatrix} .$$

Note: This same "trick" can be applied to an n-th order by defining $y_1 = y$, $y_2 = y_1'$, $y_3 = y_2'$, $\ldots$, $y_n = y_{n-1}'$.

Before discussing how to solve Duffing's equation specifically, we discuss the commands which solve time-evolution odes. To obtain a numerical solution to a time-evolution first-order ode system, enter

```
>> <ode solver>(<function handle>, tspan, y0)
```

or

```
>> [t, Y] = <ode solver>(<function handle>, tspan, y0)
```

or

```
>> sol = <ode solver>(<function handle>, tspan, y0)
```

First, we have to choose which of the ode solvers to use, which is discussed in detail shortly. It would be possible for MATLAB itself to decide which numerical method to employ. However, there are good reasons why the decision should be left in the hand of the user.

Warning: Make sure you understand how to enter the name of the function handle. This is discussed at length in section 3.2, and we also briefly discuss it below.

All of the solvers use the same input and output arguments, which we now discuss. The input parameters are:

`function` The name of the function handle that calculates $\mathtt{f}(t, \mathtt{y})$.

`tspan` The vector that specifies the time interval over which the solution is to be calculated. If this vector contains two elements, these are the initial time and the final time; in this case the ode solver determines the times at which the solution is output. If this vector contains more than two elements, these are the only times at which the solution is output.
Note: the final time can be less than the initial time, in which case the trajectory is moving backwards in time.

`y0` The vector of the initial conditions for the ode.

If there are no output parameters, the individual elements of the solution, i.e., $y_1(t)$, $y_2(t)$, $\ldots$, $y_n(t)$, are plotted vs. t on a single plot. The circles on the trajectories show the actual times at which the solution is calculated.

If there are two output parameters, these are:

`t` The column vector of the times at which the solution is calculated.[†]

`Y` The matrix which contains the numerical solution at the times corresponding to `t`. The first column of `Y` contains y_1, the second column y_2, etc.[‡]

If there is one output parameter, then it is a structure. The output is now

`sol.x` The column vector of the times at which the solution is calculated.

`sol.y` The matrix which contains the numerical solution at the times corresponding to `t`.

There are seven distinct ode solvers which can be used, as shown in the table below. All these ode solvers use an adaptive step size to control the error in the numerical solution. Each time step is chosen to *try* to keep the local error within the prescribed bounds as determined by the relative error and the absolute error tolerances (although it does not always succeed). That is, e_i, which is the error in y_i, is *supposed* to satisfy

$$e_i \leq \max\{\, \mathtt{RelTol} \cdot |y_i|, \mathtt{AbsTol(i)} \,\}$$

where the default value of `RelTol` is 10^{-3} and of the vector `AbsTol` is 10^{-6} for each element. (However, there is no guarantee that the error in the numerical calculation actually satisfies this bound.)

[†]The `t` in `[t, Y]` is unrelated to the `t` argument in the function `duffing`.

[‡]We have capitalized the `Y` in `[t, Y]` to indicate that the output is a matrix whereas the argument `y` is a vector in the function.

ODE Solvers

`ode45`	Non-stiff ode solver; fourth-order, one-step method.
`ode23`	Non-stiff ode solver; second-order, one-step method.
`ode113`	Non-stiff ode solver; variable-order, multi-step method.
`ode15s`	Stiff ode solver; variable-order, multi-step method.
`ode23s`	Stiff ode solver; second-order, one-step method.
`ode23t`	Stiff ode solver; trapezoidal method.
`ode23tb`	Stiff ode solver; second-order, one-step method.

It is up to *you* to decide which ode solver to use. As a general rule, unless you believe that the ode is stiff (which we discuss in the next paragraph), try `ode45` or `ode113`. For a given level of accuracy, these methods should run "reasonably fast". (Which one runs faster is very dependent on the ode.) If you know (or believe) that the ode is stiff, or if these two non-stiff solvers fail, then try `ode15s`.

And what is a *stiff* ode? There is no precise definition. Instead, we say it is stiff if the time step required to obtain a stable and accurate solution is "unreasonably" small. The best way to explain this rather vague impression is through some simple examples.

Consider the second-order time-evolution ode

$$y'' + 999y' + 1000y = 0 \quad \text{for } t \geq 0$$

with the initial conditions $y(0) = \eta_1$ and $y'(0) = \eta_2$. The solution to this ode is

$$y(t) = c_1 e^{t} + c_2 e^{-1000t}$$

where

$$c_1 = \frac{1}{1001}(\eta_1 - \eta_2) \quad \text{and} \quad c_2 = \frac{1}{1001}(1000\eta_1 + \eta_2)\,.$$

There are two time scales in this solution: there is a rapid decay due to the e^{-1000t} term and there is a slow growth due to the e^{t} term. Initially, the time step will be "very small" so that the rapid decay is calculated accurately (i.e., $\Delta t \ll {}^{1}/_{1000}$). However, soon it will be negligible and the time step should increase so that it calculates the slow growth accurately (i.e., $\Delta t \ll 1$). However, if a non-stiff solver, such as `ode45` or `ode23`, is used, the time step must *always* be "very small". That is, it must accurately track the rapidly decaying term — even after this term has disappeared in the numerical solution. The reason is that a numerical instability will cause the trajectory to blow up if the time step increases. However, if a stiff solver is used, the time step can increase by many orders of magnitude when the rapidly decaying term has disappeared.

The same is true for the ode

$$y'' + 1001y' + 1000y = 0$$

whose solution is

$$y(t) = c_1 e^{-t} + c_2 e^{-1000t}\,.$$

Initially, the time step will be "very small" so that the rapid decay is calculated accurately (i.e., $\Delta t \ll {}^{1}/_{1000}$). However, soon it will be negligible and the time step should increase so that it calculates the slowly decaying mode accurately (i.e., $\Delta t \ll 1$).

On the other hand, consider the ode

$$y'' - 1001y' + 1000y = 0$$

whose solution is

$$y(t) = c_1 e^{t} + c_2 e^{1000t}\,.$$

the time step must always be "very small" so that the rapidly growing mode e^{1000t} is calculated accurately (i.e., $\Delta t \ll {}^{1}/_{1000}$). Thus, this is not a stiff ode.

The above examples are *very* simple. They are only designed to show that an ode is stiff if there is a rapidly decaying mode and any growth in the solution occurs on a much slower time scale. (This frequently happens in chemical reaction models, where some reactions occur on a very fast time scale and

and other occur on a much slower time scale.) In the next subsection we discuss van der Pol's equation, a second-order ode which is either non-stiff or stiff depending on the value of one parameter. You can plot the solution and observe the separation of the fast scale and the slow scale as this parameter increases.

One difficulty with a stiff ode solver is that you may well have to supply the Jacobian of the ode yourself. The Jacobian of $\mathtt{f}(t,\mathtt{y})$ is the $n\times n$ matrix

$$\mathtt{J}(t,\mathtt{y}) = \left(\frac{\partial f_i}{\partial y_j}(t,\mathtt{y})\right) ,$$

i.e., the element in the i-th row and j-th column of `J` is

$$\frac{\partial f_i}{\partial y_j} .$$

Any of the stiff methods can approximate this matrix numerically. However, if the ode is "bad" enough, this may not be enough. You may have to calculate all these partial derivatives yourself and include them in your function file. (We show an example of this later.)

The reason for this large choice of ode solvers is that some odes are very, very, very NASTY. It is possible that most of the ode solvers will fail and only one, or maybe two, will succeed. SAY MORE???

To conclude this subsection, we return to Duffing's equation. Suppose we want to solve the ode for $t \in [0, 100]$ with initial conditions $\mathtt{y} = (2, 1)^{\mathrm{T}}$ and plot the results. Since this is a very well-behaved ode for the parameters given, we can use `ode45`. The simplest approach is to use an anonymous function to input the right-hand side.

```
≫ alpha = 0.05;
≫ beta = 1.0;
≫ Gamma = 0.5;
≫ omega = 1.0;
≫ duffing_a = @(t, y)[ y(2) ; y(1)*(1-beta*y(1)^2)-alpha*y(2)+Gamma*cos(omega*t) ];
≫ ode45(duffing_a, [0 100], [2 1]);
```

(The "a" denotes the fact that `duffing_a` is an **a**nonymous function handle.) The solution will now be plotted as y_1 and y_2 vs. t. (This plot is rather "cluttered" because, not only is the trajectory plotted, but in addition markers are put at each of the points of the numerical solution.)

Warning: There are a number of parameters which are needed by the function and these must be defined before the function is created. Also, the function handle `duffing_a` will always use these parameters, even if they are later changed.

Note: Since `duffing_a` is already a function handle, we merely need to use its name as the first argument to `ode45`.

To obtain complete control over what is plotted, you should let `ode45` output the trajectory and do the plots yourself. This is easily accomplished by changing the last line of the previous code to

```
≫ [t, Y] = ode45(duffing_a, [0 100], [2 1]);
≫ figure(1)
≫ subplot(2, 1, 1)
≫ plot(t, Y(:,1))
≫ subplot(2, 1, 2)
≫ plot(t, Y(:,2))
≫ figure(2)
≫ plot(Y(:,1), Y(:,2))
```

This results in a plot of y vs. t and a separate plot of y' vs. t, so that both plots are visible even if they have vastly different scales. There is also a separate plot of y' vs. y, which is called a *phase plane*.

The next simplest approach is to use a nested function, and so there must also be a primary function.

```
function duffing_ode(alpha, beta, Gamma, omega, y0, final_time)
ode45(@duffing_n, [0 final_time], y0);
   function deriv = duffing_n(t, y)
   % duffing_n: Duffing's equation, nested function
   deriv = [ y(2) ; y(1)*(1-beta*y(1)^2)-alpha*y(2)+Gamma*cos(omega*t) ];
   end
end
```

(The "n" denotes the fact that `duffing_n` is a **n**ested function.) Note that the parameters are input to the primary function and so are immediately accessible to the nested function. Clearly, this second approach (of using a nested function) requires more coding that the first approach (of using an anonymous function). However, the first approach only works if the right-hand side can be defined using one MATLAB statement. If the right-hand side is more complicated, then a nested function is the simplest choice.

Note: Since `duffing_n` is a function, and not a function handle, we have to include "`@`" before the name of the function.

The third, and oldest, approach is to create a separate function m-file (i.e., a primary function) which calculates the right hand side of this ode system.

```
function deriv = duffing_p(t, y)
% duffing_p: Duffing's equation, primary function
alpha = 0.05;
beta = 1.0;
Gamma = 0.5;
omega = 1.0;
deriv = [ y(2) ; y(1)*(1-beta*y(1)^2)-alpha*y(2)+Gamma*cos(omega*t) ];
```

(The "p" denotes the fact that `duffing_p` is a **p**rimary function.) Note that all the parameters are defined in the m-file so that it will have to be modified whenever we want to modify the parameters. *This is a very bad approach because this file will have to be repeatedly modified.*

Note: Since `duffing_p` is a function, and not a function handle, we have to include "`@`" before the name of the function.

Finally, it is very inconvenient that the parameters in Duffing's equation are determined in the function itself. We should be able to "explore" the rich behavior of Duffing's equation without having to constantly modify the function — in fact, once we have the function exactly as we want it, we should never touch it again. (This is not only true for esthetic reasons; the more we fool around with the function, the more likely we are to screw it up!)

This is easily done by adding parameters to the function file.

```
function deriv = duffing_p2(t, y, alpha, beta, Gamma, omega)
% duffing_p2: Duffing's equation, primary function
% with coefficients passed through the argument list
deriv = [ y(2) ; y(1)*(1-beta*y(1)^2)-alpha*y(2)+Gamma*cos(omega*t) ];
```

(The "p2" denotes the fact that `duffing_p2` is another **p**rimary function.) However, this function cannot be called directly by the ode solver. Instead it is called indirectly by

```
≫  alpha = 0.05;
≫  beta = 1.0;
≫  Gamma = 0.5;
≫  omega = 1.0;
≫  duffing_c = @(t, y) duffing_p2(t, y, alpha, beta, Gamma, omega)
≫  ode45(duffing_c, [0 100], [2 1]);
```

Notice that the function `duffing_c` takes only two arguments: t and `y`. But the function it invokes is `duffing_p2` which takes six arguments. Thus, `ode45` thinks it is only passing two arguments to `duffing_c`, but it is actually passing six arguments to `duffing_p2`. In computer science this is a very simple example of *closure*[†]. (The "c" denotes the fact that `duffing_c` is an anonymous function handle

[†]A closure is a complicated term to explain. In this context it means that the parameters used when the function `duffing_c` was defined are saved and can be referenced inside `ode45`.

which is also a **c**losure.)

To see a sampling of the different type of behavior in Duffing's equation, let $\alpha = 0.15$, $\beta = 1$, $\Gamma = 0.3$ and $\omega = 1$, and let the initial condition be $\mathtt{y}(0) = (0,1)^{\mathrm{T}}$. After a short initial transient, the solution settles down and appears to be "regular" by $t = 100$: in fact, it appears to be exactly periodic with a period of 2π due to the $0.3\cos t$ term. (In fact, to the accuracy of the computer it *is* exactly periodic.) However, if we merely change the initial condition to $\mathtt{y} = (1,0)^{\mathrm{T}}$ the behavior appears to be chaotic, even at $t = 1000$. Here is an example of a ode which has periodic motion for one initial condition and is chaotic for another! If we now change α from 0.15 to 0.22 we find periodic motion with a period of 6π. This is just a sampling of the behavior of Duffing's equation in different parameter regions.

By the way, to separate the initial transient behavior from the long-time behavior, you can use the script m-file

```
initial_time = ???
final_time = ???
y0 = ???
alpha = ???;
beta = ???;
Gamma = ???;
omega = ???;
duffing_a = @(t, y)[ y(2) ; y(1)*(1-beta*y(1)^2)-alpha*y(2)+Gamma*cos(omega*t) ]; [t,
Y] = ode45(duffing_a, [0 initial_time], y0);
figure(1)
plot(Y(:,1), Y(:,2))
l_t = length(t);
[t, Y] = ode45(duffing_c, [t(l_t) final_time], Y(l_t,:));
figure(2)
plot(Y(:,1), Y(:,2))
```

10.2. Advanced Commands

There are a number of parameters that we can use to "tune" the particular ode solver we choose. The MATLAB function `odeset` is used to change these parameters from their default values by

```
≫  params = odeset('<Prop 1>', <Value 1>, '<Prop 2>', <Value 2>, ...)
```

where each parameter has a particular name and it is followed by the desired value. The result of this command is that the parameters are contained in the variable `params`. You include these parameters in the ode solver by adding this variable to the argument list of the ode solver function as

```
≫  [t, Y] = <ode solver>(<function handle>, tspan, y0, params)
```

Some of the more common parameters are shown in the table below; they will be discussed further later. To determine all the parameters, their possible values and the default value, enter

```
≫  odeset
```

Initial-Value ODE Solver Parameters

`odeset('<Prop 1>', <Value 1>, ...)`	Assigns values to properties; these are passed to the ode solver when it is executed.
`AbsTol`	The absolute error tolerance. This can be a scalar in which case it applies to all the elements of `y` or it can be a vector where each element applies to the corresponding element of `y`. (Default value: 10^{-6}.)
`Events`	A handle to a function which determines when an event occurs.
`Jacobian`	A handle to a function which returns the Jacobian.
`JPattern`	A sparse matrix whose nonzero elements (which should be 1) correspond to the possible nonzero elements of the Jacobian. This is only used when the Jacobian is calculated numerically, i.e., when the `Jacobian` property is not used.
`OutputFcn`	A handle to a function which is called after each successful time step. For example, a plot of the trajectory can be generated automatically as it is being calculated. Useful MATLAB functions are: `'odeplot'` which generates a plot of time versus all the components of the trajectory, i.e., t vs. $y_1, y_2, \ldots, y_n$; `'odephas2'` which generates a plot of y_1 vs. y_2, i.e., `Y(:,1)` vs. `Y(:,2)`; `'odephas3'` which generates a plot of y_1 vs. y_2 vs. y_3, i.e., `Y(:,1)` vs. `Y(:,2)` vs. `Y(:,3)`. It is possible to plot different components of `y` using `OutputSel`.
`OutputSel`	A vector containing the components of `Y` which are to be passed to the function specified by the `OutputFcn` parameter.
`Refine`	Refines the times which are output in `t`. This integer value increases the number of times by this factor. (Default value: 1 for all ode solvers except `ode45`, 4 for `ode45`.)
`RelTol`	The relative error tolerance. (Default value: 10^{-3}).
`Stats`	Whether statistics about the run are output on the terminal (value: `'on'`) after the trajectory is calculated or they are not (value: `'off'`). (Default value: `'off'`.)

For example, if you want to use `ode45` with the relative error tolerance set to 10^{-6} for Duffing's equation, enter

```
≫  params = odeset('RelTol', 1.e-6);
≫  [t, Y] = ode45(duffing_a, tspan, y0, params);
```

The trajectory will be more accurate — but the command will run slower. If you also want the statistics on the performance of the particular ode solver used, enter

```
≫  params = odeset('RelTol', 1.e-6, 'Stats', 'on');
≫  [t, Y] = ode45(@duffing_a, tspan, y0, params);
```

and the number of successful steps, the number of failed steps, and the number of times `f(t,y)` was evaluated will be printed on the terminal. This might be useful in "optimizing" the performance of the ode solver if the command seems to be running excessively slowly. For implicit methods where the Jacobian needs to be calculated, the number of times the Jacobian was evaluated, the number of `LU` decompositions, and the number of times the linear system was solved will also be returned.

The ode solver can also record the time and the location when the trajectory satisfies a particular condition: this is called an *event*. For example, if we are calculating the motion of the earth around the sun, we can determine the position of the earth when it is closest to the sun and/or farthest away; or, if we are following the motion of a ball, we can end the calculation when the ball hits the ground — or we can let it continue bouncing. Enter

```
≫  ballode
```

to see a simple example.

For example, suppose we want to record where and when a trajectory of Duffing's equation passes through $y_1 = \pm 0.5$. That is, we define an "event" to be whenever the first component of `y` passes

through -0.5 or $+0.5$. This can be done by modifying the primary function `duffing_ode` and replacing the `ode45` statement by

```
params = odeset('RelTol', 1.e-6, 'Events', @duffing_event);
[t, Y, tevent, Yevent, indexevent] = ode45(@duffing_n, tspan, y0, params);
```

where we create a new nested function in the primary function `duffing_ode`.

```
function [value, isterminal, direction] = duffing_event(t, y)
value = [y(1)+0.5; y(1)-0.5];     % check whether y(1) passes through    ±0.5
isterminal = [0; 0];              % do not halt when this occurs
direction = [0; 0];               % an event occurs when y(1) passes through
                                  % zero in either direction
end
```

Note that we can define the right-hand side of Duffing's equation by using `duffing_a`, `duffing_n`, `duffing_p`, or `duffing_p2` and `duffing_c`. We have chosen `duffing_n` since we have created the nested function `duffing_event`. (We could let `duffing_event` be a primary function, but there is no reason to do so.)

There are a number of steps we have to carry out to turn "events" on. First, we have to use the `odeset` command. However, this only tells the ode solver that it has to watch for one or more events; it does not state what event or events to watch for. Instead, we describe what an event *is* in this new function. Three vector arguments are output:

`value` – A vector of values which are checked to determine if they pass through zero during a time step. No matter how we describe the event, as far as the ode solver is concerned an event only occurs when an element of this vector passes through zero. In some cases, such as this example it is easy to put an event into this form. In other cases, such as determining the apogee and perigee of the earth's orbit, the calculation is more complicated.

`isterminal` – A vector determining whether the ode solver should terminate when this particular event occurs: 1 means yes and 0 means no.

`direction` – A vector determining how the values in `value` should pass through zero for an event to occur:

- 1 means the value must be increasing through zero for an event to occur,
- -1 means the value must be decreasing through zero for an event to occur, and
- 0 means that either direction triggers an event.

The final step is that the left-hand side of the calling statement must be modified to

```
[t, Y, tevent, Yevent, index_event] = ode45(...);
```

Any and all events that occur are output by the ode solver through these three additional variables:

`tevent` is a vector containing the time of each event,
`Yevent` is a matrix containing the location of each event, and
`index_event` is a vector containing which value in the vector `value` passed through zero.

If the result is stored in the structure `sol`, the new output is

`sol.xe` is a vector containing the time of each event,
`sol.ye` is a matrix containing the location of each event, and
`sol.ie` is a vector containing which value in the vector `value` passed through zero.

Since the function `duffing_event` might appear confusing, we now discuss how an event is actually calculated. At the initial time, t and `y` are known and `duffing_event` is called so that the vector

$$\mathtt{e}^{(0)} = \begin{pmatrix} y_1^{(0)} + 0.5 \\ y_1^{(0)} - 0.5 \end{pmatrix},$$

i.e., `value`, can be calculated. In addition, `isterminal` and `direction` are returned. Next, `duffing` is called and the solution $\mathtt{y}^{(1)}$ is calculated at time $t^{(1)}$. `duffing_event` is called again and $\mathtt{e}^{(1)}$ is calculated and compared elementwise to $\mathtt{e}^{(0)}$. If the values have different signs in some row, then `direction` is checked to determine if the values are passing through zero in the correct direction or if either direction is allowed. If so, the time at which the element is zero is estimated and the ode is solved again to obtain

a more accurate estimate. This procedure continues until the zero is found to the desired accuracy. Then `isterminal` is checked to see if the run should be continued or should be stopped.

Another interesting ode is van der Pol's equation

$$y'' - \mu(1 - y^2)y' + y = 0$$

where $\mu > 0$ is the only parameter. As a first order system it is

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}' = \begin{pmatrix} y_2 \\ \mu(1 - y_1^2)y_2 - y_1 \end{pmatrix}$$

and its Jacobian is

$$\mathtt{J} = \begin{pmatrix} 0 & 1 \\ -2\mu y_1 y_2 - 1 & \mu(1 - y_1^2) \end{pmatrix}.$$

The right-hand side can be coded as a nested function inside a primary function by

```
function vdp_ode(mu, y0, final_time)
ode45(@vdp_n, [0 final_time], y0);
   function deriv = vdp_n(t, y)
   % vdp_n: van der Pol's equation
   deriv = [y(2); mu*(1-y(1)^2)*y(2)-y(1)];
   end
end
```

This is not stiff unless μ is "large". For example, let $\mu = 1$ and solve the ode with initial conditions $y(0) = 1$ and $y'(0) = 0$ for $t \in [0, 100]$ using `ode45`. Then, plot the result and note the number of elements in `t`. Repeat this procedure using $\mu = 10$ and increase the final time, if necessary, so that you still see a few complete oscillations. Then let $\mu = 100$, etc., until the time required to plot a few oscillations becomes "very large". Then use `ode15s` and note the huge difference in the time required. ADD ALL VDPS???

There is no need to use the ode solver parameters `JPattern` or `Jacobian` in this example because this ode is so "nice". However, since they might be needed for a NASTIER ode, we include them by using

```
Vdp_pattern = sparse([1 2 2], [2 1 2], [1 1 1], 2, 2);
params = odeset('Jacobian', @vdpj_n, 'JPattern', Vdp_pattern):
[t, Y] = <ode solver>(@vdp_n, tspan, y0, opt);
```

where the Jacobian is calculated numerically using the nested function

```
function J = vdpj_n(t, y)
% vdpj_n: Jacobian for van der Pol's equation
J = [ 0 1; -2*mu*y(1)*y(2)-1 mu*(1-y(1)^2) ];
end
```

for the elements determined by `Vdp_pattern`. By the way, if we use the property `JPattern` but not `Jacobian` then the Jacobian is calculated numerically just for the elements determined by the sparse matrix.

Note: Plotting the trajectory by

```
plot(t, Y)
```

is not very instructive. Instead, use

```
subplot(2,1,1)
plot(t, Y(:,1))
subplot(2,1,2)
plot(t, Y(:,2))
```

Our final example is slightly more complicated. Suppose we kick a ball into the air with initial speed s and at an angle of α, and we want to follow its motion until it hits the ground. Let the x axis be the horizontal axis along the direction of flight and z be the vertical axis. Using Newton's laws we obtain the ode system

$$x'' = 0 \quad \text{and} \quad z'' = -g$$

where $g = 9.8$ meters/second is the acceleration on the ball due to the earth's gravity. The initial conditions are

$$x(0) = 0\,,\ x'(0) = s\cos\alpha\,,\ z(0) = 0, \text{ and } z'(0) = s\sin\alpha$$

where we assume, without loss of generality, that the center of our coordinate system is the initial location of the ball. We also want to determine four "events" in the ball's flight: the highest point of the trajectory of the ball and the time it occurs, the distance it travels and the time it hits the ground, and the x values and times when the ball reaches the height $h > 0$. But beware because the ball may never attain this height!

Although these odes can be solved analytically (consult any calculus book), our aim is to give an example of how to use many of the advanced features of MATLAB's ode solvers. (If we would include the effects of air resistance on the ball, then these odes would become nonlinear and would not be solvable analytically.) We convert Newton's laws to the first-order system

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix}' = \begin{pmatrix} y_2 \\ 0 \\ y_4 \\ -g \end{pmatrix}$$

by letting $y_1 = x,\ y_2 = x',\ y_3 = z,$ and $y_4 = z'$ The initial conditions are

$$y_1(0) = 0\,,\ y_2(0) = s\cos\alpha\,,\ y_3(0) = 0\,, \text{ and } y_4(0) = s\sin\alpha\,.$$

One complication with solving this system numerically is that we do not know when the ball will hit the ground, so we cannot give the final time. Instead, we use a time, $10s/g$ which is much greater than needed and we let the program stop itself when the ball hits the ground. In addition, we want the relative error to be 10^{-6}. Finally, we want the trajectory (i.e., z vs. x) to be plotted automatically.

The following is a completely self-contained example using nested functions.

```
function [times, values] = gravity_ode(speed, angle, height)
% gravity_ode: The trajectory of a ball thrown from (0,0) with initial
%               speed and angle (in degrees) given.
%    times: (1) = time ball at peak, (2) = time ball hits ground
%           (3,4) = time ball attains height
%    values: (1) = z value at peak, (2) = x value when ball hits ground
%            (3,4) = x values when ball attains height
%    Note: (3,4) will not be used if height > z value at peak
g = 9.8;
gravity_init()
[t, Y, tevent, Yevent, index_event] = ode45(@gravity, tspan, y0, params);
if length(t) == 2
   times = tevent;
   values = [Yevent(3,1) Yevent(1,2)];
else
   times = tevent([2 4 1 3]);
   values = [Yevent(3,2) Yevent(1,4) Yevent(1,1) Yevent(1,3)];
end
%%%% nested functions follow
   function gravity_init
   % gravity_init: Initialize everything
   tspan = [0 10*speed/g];
   y0 = [ 0; speed*cos(angle*pi/180); 0; speed*sin(angle*pi/180) ];
   params = odeset('RelTol', 1.e-6, ...
                   'Events', @gravity_event, ...
                   'Refine', 20, ...
                   'OutputFcn', 'odephas2', ...
                   'OutputSel', [1 3]);
   end
   function deriv = gravity(t, y)
   % gravity: Calculates the right-hand side of the ode
   deriv = [y(2); 0; y(4); -g];
   end
   function [value, isterminal, direction] = gravity_event(t, y)
   % gravity_event: determines the events
   value = [y(3) y(3)-height y(4)];      % z = 0, z-height = 0, z' = 0
   isterminal = [1 0 0];      % halt only when z = 0
   direction = [-1 0 -1];      % an event occurs when z or z' decrease through 0
                               % or z-height passes through 0 in either direction
   end
end
```

Note that the parameters `g`, `speed`, `angle`, and `height` do not need to be passed into the nested functions. Similarly, `tspan`, `y0`, and `params` do not need to be passed out.

MATLAB also has the function `ode15i` which solves fully implicit odes. It is very similar to the functions we have already discussed, but there is one important difference. Although it is a very powerful function, we only provide a very simple example which uses it.

We consider a linear second-order ode in a neighborhood of a regular singular point. Consider the ode

$$P(t)y''(t) + Q(t)y'(t) + R(t)y(t) = 0$$

where $P(t)$, $Q(t)$, and $R(t)$ are polynomials with no common factors. The *singular points* of this ode are the values of t for which $P(t) = 0$. If t_0 is a singular point, it is a *regular singular point* if $\lim_{t\to t_0}(t-t_0)Q(t)/P(t)$ and $\lim_{t\to t_0}(t-t_0)^2R(t)/P(t)$. A "common" ode of this type is Bessel's equation

$$t^2y''(t) + ty'(t) + (t^2 - n^2)y(t) = 0 \quad \text{for } t \geq 0 \tag{10.1}$$

where n is a nonnegative integer and the initial condition is given at $t = 0$. The solution is denoted by $J_n(t)$ and, for specificity, we will concentrate on $n = 1$. At $t = 0$ the ode reduces to $-y(0) = 0$ and so we require $y(0) = 0$. The free initial condition is $y'(0)$ and for this example we choose $y'(0) = 1$.

If we write Bessel's equation as

$$y''(t) + \frac{1}{t}y'(t) + \left(1 - \frac{n^2}{t^2}\right) y(t) = 0 \tag{10.2}$$

we clearly have a problem at $t = 0$ and for $t \approx 0$. The ode solvers we discussed previously can handle (10.2) for $t \geq 1$ with the initial conditions that $y(1)$ and $y'(1)$ are given. However, a completely different method of solution is required for $t \geq 0$ and the form (10.1) is preferred to (10.2).

When we convert Bessel's equation to the first order system we again let $y_1(t) = y(t)$ and $y_2(t) = y'(t)$ and leave the t^2 in the numerator to obtain

$$\begin{pmatrix} y_1' \\ t^2 y_2' \end{pmatrix} = \begin{pmatrix} y_2 \\ -ty_2 - (t^2 - 1)y_1 \end{pmatrix}$$

Previously, we have always written the first-order system as $\mathtt{y}' = \mathtt{f}(t, y)$, but this form has a problem when $t = 0$. Instead, we write it as $\mathtt{g}(t, \mathtt{y}, \mathtt{y}') = \mathbf{0}$ so that

$$\mathtt{g}(t, \mathtt{y}, \mathtt{y}') = \begin{pmatrix} y_1' - y_2 \\ t^2 y_2' + ty_2 + (t^2 - 1)y_1 \end{pmatrix} .$$

Finally, we not only have to input the initial condition $\mathtt{y}(0) = (0, 1)^{\mathrm{T}}$, but we also have to input $\mathtt{y}'(0) = \big(y_1'(0), y_2'(0)\big)^{\mathrm{T}}$. It is easy to calculate $y_1'(0) = y_2(0)$, but $y_2'(0) \equiv y''(0)$ is more complicated. Differentiate (10.1) with respect to t to obtain

$$t^2 y'''(t) + 3ty''(t) + t^2 y'(t) + 2ty(t) = 0$$

and differentiate it again to obtain

$$t^2 y''''(t) + 5ty'''(t) + (t^2 + 3)y''(t) + 4ty'(t) + 2y(t) = 0 .$$

Now set $t = 0$ to obtain $y''(0) = 0$. We can solve Bessel's equation for $t \in [0, 10]$ by

```
≫  g = @(t, y, yp) [yp(1)-y(2);t^2 *yp(2)+t*y(2)+(t^2 -1)*y(1)];
≫  tspan = [0 10]
≫  y0 = [0;1]
≫  yp0 = [1;0]
≫  [t,Y] = ode15i(g, tspan, y0, yp0)
≫  plot(t, Y(:,1))
```

Implicit ODE Solver

`ode15i`	Stiff ode solver for the fully implicit ode $\mathtt{f}(t, y, y') = \mathbf{0}$.

11. Boundary-Value Ordinary Differential Equations

In addition to initial-value ordinary differential equations there is a second type of odes that MATLAB can solve numerically. Boundary-value odes are also odes of the form

$$\frac{d}{dx}\mathtt{y} = \mathtt{f}(x, \mathtt{y}) \qquad \text{for } x \in [a, b]$$

where $\mathtt{y} \in \mathbb{R}^n$ but conditions are given at *both* ends of the interval. If the boundary conditions are separated, then k conditions are given at $x = a$ and $n - k$ other conditions are given at $x = b$. If the boundary conditions are non-separated, then the conditions at $x = a$ and at $x = b$ are related. To allow any of these boundary conditions we write the boundary conditions as $\phi\big(\mathtt{y}(a), \mathtt{y}(b)\big) = \mathbf{0}$ where $\phi \in \mathbb{R}^n$.

For simplicity, we will only consider two closely related second-order odes, i.e., $n = 2$. This example should enable you to study any boundary-value ode. Consider the two nonlinear boundary-value ordinary differential equations

$$\frac{d^2y}{dx^2}(x) + 2\frac{dy}{dx}(x) + \epsilon e^{y(x)} = 0 \tag{11.1a}$$

and

$$\epsilon\frac{d^2y}{dx^2}(x) + 2\frac{dy}{dx}(x) + e^{y(x)} = 0 \tag{11.1b}$$

for $x \in [0, 1]$ where $\epsilon > 0$. Our boundary conditions are

$$\phi\big(y(0), y(1)\big) = \begin{pmatrix} y(0) \\ y(1) \end{pmatrix} = \mathbf{0}\,, \tag{11.2}$$

which are called *Dirichlet* boundary conditions. These two odes are quite simple, but also quite interesting and challenging to solve for certain intervals in ϵ.

We could use the *Neumann* boundary conditions $y'(0) = 4$ and $y'(1) = -7$ by

$$\phi\big(y(0), y(1)\big) = \begin{pmatrix} y'(0) - 4 \\ y'(1) + 7 \end{pmatrix} = \mathbf{0}\,. \tag{11.3}$$

Or we could use the *mixed* boundary conditions $y(0) - y'(0) = 1$ and $y(1) + 2y'(1) = 3$ by

$$\phi\big(y(0), y(1)\big) = \begin{pmatrix} y(0) - y'(0) - 1 \\ y(1) + 2y'(1) - 3 \end{pmatrix} = \mathbf{0}\,. \tag{11.4}$$

Finally, we could use periodic boundary conditions, which are non-separated, by

$$\phi\big(y(0), y(1)\big) = \begin{pmatrix} y(1) - y(0) \\ y'(1) - y'(0) \end{pmatrix} = \mathbf{0}\,. \tag{11.5}$$

The primary MATLAB function is `bvp4c`. However, the functions `bvpinit` and `deval` are also needed. We solve the boundary value problem by

```
>>  sol = bvp4c(<right-hand side>, <boundary conditions>, <initial guess>)
```

There are two functions we need to write: `odefun` is $\mathtt{f}(x, \mathtt{y})$ and `bcfun` is the boundary conditions. For our example the ode given by

```
function yp = nlode(x, y)
global which_ode eps
if which_ode == 1
   yp = [y(2); -eps*exp(y(1))-2*y(2)];
else
   yp = [y(2); -(exp(y(1))+2*y(2))/eps];
end
```

where we use `global` to input which ode to use and ϵ. The boundary condition is given by

```
function bc = nlode_bc(ya, yb)
bc = [ya(1); yb(1)];
```

Since these boundary conditions are particularly simple, we also include the function

```
function bc = nlode_bc2(ya, yb)
bc = [ya(1)-ya(2)-1; yb(1)+2*yb(2)-3];
```

for mixed boundary conditions (11.4). In addition, we have to choose an initial guess for $\mathtt{y}(x)$ using `bvpinit` by either

```
» bvpinit(x, y_init)
```

or

```
» bvpinit(x, <initial guess function>)
```

For example, if we want the initial iterate to be a parabola which is zero at $x = 0$ and 1 and has maximum value A then $y(x) = y_1(x) = 4Ax(1-x)$ and $y'(x) = y_2(x) = 4A(1-2x)$ then we can write

```
» x = linspace(0, 1, 21);
» solinit = bvpinit(x, @nlode_y_ic);
```

where `nrode_y_ic` is written as

```
function y_ic = nlode_y_ic(x)
global A
y_ic = [4*A*x.*(1 - x); 4*A*(1-2*x)];
```

The only alternative is to write

```
» x = linspace(0, 1, 21);
» y1_val = ???;
» y2_val = ???;
» solinit = bvpinit(x, [y1_val; y2_val]);
```

where `y1_val` and `y2_val` are scalar values. Thus the initial guess is `y1 = y1_val*ones(size(x))` and `y2 = y2_val*ones(size(x))`. This is often unacceptable because constant initial guesses may be so far from the solution that convergence cannot be obtained. What we would **like** to do is

```
» x = linspace(0, 1, 21);
» y1 = 4*A*x.*(1 - x);
» y2 = 4*A*(1 - 2*x);
» solinit = bvpinit(x, [y1; y2]); % WRONG
```

This fails because `y1` and `y2` must be scalar variables and not vectors. If you **really**, **really** need `y1` and `y2` to be vectors, then do not use `bvpinit`. Instead, specify the structure `solinit` directly by entering

```
» x = linspace(0, 1, 21);
» y1 = 4*A*x.*(1 - x);
» y2 = 4*A*(1 - 2*x);
» solinit.x = x;
» solinit.y = [y1;y2];
```

Warning: This is dangerous because future versions of Matlab might change the fieldnames of the structure `solinit`. However, it works for now.

We are finally ready to solve this ode by

```
» global which_ode eps
» global A
» which_ode = 1;
» A = 1;
» eps = 3;
» x = linspace(0, 1, 21);
» solinit = bvpinit(x, @nlode_y_ic);
» sol = bvp4c(@nlode, @nlode_bc, solinit);
```

The solution is contained in `sol` and is extracted by `deval`. For example, if $x_i = (i-1)\Delta x$ where $x_1 = 0$ and $x_n = 1$ then we determine, and plot, the numerical solution `y` by

```
» xpt = linspace(0, 1, 101);
» ypt = deval(sol, xpt);
» plot(xpt,ypt(1,:), xpt, ypt(2,:), 'r')
```

Having done all this work, we now combine everything into the function m-file `nlode_all` to show how

much easier it is to use nested functions and to combine everything into one primary function.

```
function sol = nlode_all(which_ode, eps, A, nr_points)
% nlode_all: boundary-value solver using bvp4c
x = linspace(0, 1, nr_points);
solinit = bvpinit(x, @nlode_y_ic);
sol = bvp4c(@nlode, @nlode_bc, solinit);
xpt = linspace(0, 1, 101);
ypt = deval(sol, xpt);
plot(xpt,ypt(1,:), xpt, ypt(2,:), 'r')
%%%% nested functions follow
   function y_ic = nlode_y_ic(x)
   y_ic = [4*A*x.*(1 - x); 4*A*(1-2*x)];
   end
   function yp = nlode(x, y)
   if which_ode == 1
      yp = [y(2); -eps*exp(y(1))-2*y(2)];
   else
      yp = [y(2); -(exp(y(1))+2*y(2))/eps];
   end
   end
   function bc = nlode_bc(ya, yb)
   bc = [ya(1); yb(1)];
   end
end
```

This m-file is easy to read and easy to debug and easy to modify. Also, the solution is returned so it can be used in the MATLAB workspace.

The reason we chose these particular odes is to "check out" `bvp4c`. For the ode (11.1a) there are two solutions for $0 \le \epsilon \lessapprox 3.82$ and **no** solutions for $\epsilon \gtrapprox 3.82$. This is a good test of any boundary-value solver. For the ode (11.1a) there is one solution for any $\epsilon > 0$. The "interesting" feature of this ode is that for $\epsilon \ll 1$ the solution rises rapidly from $y(0) = 0$ to $y(x) \approx \log 2$ for $x = \mathcal{O}(\epsilon)$ and then decays gradually so that $y(1) = 0$. It is very challenging for a boundary-value solver to be able to capture this rapid rise.

One final point needs to be emphasized. Sometimes, any "halfway decent" initial choice of `y` will converge to a solution. In fact, this is true for our example — but it is not true for many examples. Sometimes it takes a "good" initial choice to obtain convergence; a "bad" choice will never converge to the desired solution. The standard method to use to obtain a "good" initial iterate is the *continuation method*. Frequently there are values of the parameter(s) for which "good" initial iterates are known. For example, for the ode (11.1a) if $\epsilon \ll 1$ we can approximate ϵe^y by the Taylor series expansion $\epsilon(1 + y)$ and solve the resulting linear ode. If $\epsilon = 0.1$ the resulting analytical solution is a very good approximation to the numerical solution. You can use this solution as the initial guess for $\epsilon = 0.2$. The numerical solution can then be used as an initial guess for a larger value of ϵ, etc.

The only difficulty with this method is that there might be *more* solutions. When $\epsilon = 0.1$ there is a second solution whose maximum is over 8. For this solution $y'(0) \approx 35$ which indicates how rapidly the solution is growing at the left endpoint. This solution can only be found by trying "large" initial guesses (e.g., choosing `A` to be large in `nlode_y_ic`).

For the ode (11.1b) it is very difficult to determine "good" initial guesses when $\epsilon \ll 1$ since the solution grows so rapidly. Again, the continuation method is very helpful. Start with a "large" value of ϵ, say $\epsilon = 1$, and choose a "reasonable" initial guess. (Since the two odes are identical when $\epsilon = 1$ you can use the solution you found to ode (11.1a).) Then slowly decrease ϵ. For example, when $\epsilon = 0.01$ we have $y'(0) \approx 130$ and when $\epsilon = 0.001$ we have $y'(0) \approx 1300$. In conclusion, we want to remind you that for the odes we have discussed here almost any "halfway reasonable" initial choice for the ode (11.1a) will converge to one of the two solutions and for the ode (11.1b) will converge to the single solution. However, you might well find an ode for which this is not true.

Boundary-Value Solver

`bvp4c(<right-hand side>,`
`  <boundary conditions>,`
`  <initial guess>)`
	Numerically solves $\mathtt{y}'(x) = \mathtt{f}(x, \mathtt{y})$ for $x \in [a, b]$ with given boundary conditions and an initial guess for `y`. The user supplied functions are $\mathtt{f}(x, \mathtt{y}) =$ `right_hand_side(x,y)` and `boundary_conditions(ya,yb)` where $\mathtt{ya} = \mathtt{y}(a)$ and $\mathtt{yb} = \mathtt{y}(b)$.

`bvpinit(x,y)` `bvpinit(x, <initial guess function>)`	Calculates the initial guess either by giving `y` directly or by using a function `y =` `initial_guess_function`(x).
`deval(x,y_soln)`	Interpolate to determine the solution at `x`.

12. Polynomials and Polynomial Functions

In MATLAB the polynomial

$$p(x) = c_1 x^{n-1} + c_2 x^{n-2} + \cdots + c_{n-1} x + c_n \, .$$

is represented by the vector $\mathtt{q} = (c_1, c_2, \ldots, c_n)^{\mathrm{T}}$. You can easily calculate the roots of a polynomial by

```
≫  r = roots(q)
```

Conversely, given the roots of a polynomial you can recover the coefficients of the polynomial by

```
≫  q = poly(r)
```

Warning: Note the order of the coefficients in the polynomial. c_1 is the coefficient of the highest power of x and c_n is the coefficient of the lowest power, i.e., 0.

The polynomial can be evaluated at `x` by

```
≫  y = polyval(q, x)
```

where `x` can be a scalar, a vector, or a matrix. If `A` is a square matrix, then

$$p(A) = c_1 A^{n-1} + c_2 A^{n-2} + \cdots + c_{n-1} A + c_n$$

is calculated by

```
≫  polyvalm(q, A)
```

(See section 15 for more details on this type of operation.)

A practical example which uses polynomials is to find the "best" fit to data by a polynomial of a particular degree. Suppose the data points are

$$\big\{\, (-3,-2), (-1.2,-1), (0,-0.5), (1,1), (1.8,2) \,\big\}$$

and we want to find the "best" fit by a straight line. Defining the data points more abstractly as $\big\{\, (x_i, y_i) \;\big|\; i = 1, 2, \ldots, n \,\big\}$ and the desired straight line by $y = c_1 x + c_2$, the matrix equation for the straight line is

$$\begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \, .$$

In general, there is no solution to this overdetermined linear system. Instead, we find the least-squares solution $\mathtt{c} = (c_1, c_2)^{\mathrm{T}}$ by

```
≫  c = [x ones(n, 1)] \ y
```

We can plot the data points along with this straight line by

```
≫  xx = linspace(min(x), max(x), 100);
≫  yy = polyval(c, xx);
≫  plot(xx, yy, x, y, 'o')
```

We can find the "best" fit by a polynomial of degree $m < n$, i.e., $y = c_1x^m + c_2x^{m-1} + \cdots + c_{m+1}$, by calculating the least-squares solution to

$$\mathtt{V}\mathtt{c} = \mathtt{y}$$

where

$$\mathtt{V} = \begin{pmatrix} x_1^m & x_1^{m-1} & \cdots & x_1 & 1 \\ x_2^m & x_2^{m-1} & \cdots & x_2 & 1 \\ \vdots & \vdots & & \vdots & \vdots \\ x_n^m & x_n^{m-1} & \cdots & x_n & 1 \end{pmatrix} \quad \text{and} \quad \mathtt{c} = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} .$$

The matrix `V` is called a *Vandermonde matrix* . The statement

```
≫  V = vander(x);
```

generates the square Vandermonde matrix with $m = n - 1$. To generate the $n \times (m-1)$ Vandermonde matrix we want, enter

```
≫  V = vander(x)
≫  V(: , 1:m-1) = [];
```

This entire procedure can be carried out much more easily by entering

```
≫  q = polyfit(x, y, m-1)
```

where the third argument is the order of the polynomial (i.e., the number of coefficients in the polynomial).

Warning: The Vandermonde matrix is approximately as badly conditioned as the Hilbert matrix which was discussed in section 5.2. For example, `cond(vander([1 : 10]))` $= 2\times10^{12}$ whereas `cond(hilb(10))` $= 2\times10^{13}$.

You can also find a local maximum or minimum of the polynomial $p(x)$ by finding the zeroes of $p'(x)$. The coefficients of $p'(x)$ are calculated by

```
≫  polyder(q)
```

where `q` is the vector of the coefficients of $p(x)$.

Given a set of data points $\left\{ (x_i, y_i) \right\}$ there is sometimes a need to estimate values that lie within these data points (this is called *interpolation*) or outside them (this is called *extrapolation*). This estimation is generally done by fitting data which is "near" the desired value to a polynomial and then evaluating this polynomial at the value.

There are a number of commands to interpolate data points in any number of dimensions. The simplest command in one dimension to interpolate the points $\left\{ (x_i, y_i) \;\middle|\; 1 \le i \le n \right\}$ is

```
≫  yvalues = interp1(x, y, xvalues, <method>)
```

where `xvalues` is a vector of the values to be interpolated, `yvalues` is the vector of the interpolated values, and `<method>` is an optional argument specifying the method to be used. One additional requirement for this command is that the elements of `x` are monotonic, i.e., either all in increasing order or in decreasing order, to make it easy for the function to determine which data points are "near" the desired value. Five of the interpolation methods which can be used are the following:

`'nearest'`: The interpolated value is the value of the nearest data point.

`'linear'`: Linear splines are used to connect the given data points. That is, straight lines connect each pair of adjacent data points. (This is the default.)

`'spline'`: Cubic splines are used to connect the given data points. That is, cubic polynomials connect each pair of adjacent data points. The additional constraints needed to obtain unique polynomials are that the the two polynomials which overlap at each interior data point have the same first and second derivatives at this point.

`'pchip'`: Piecewise cubic Hermite polynomials connect each pair of adjacent data points. This is similar to `spline` but the second derivatives need not be continuous at the interior data points. Instead, this interpolation is better at preserving the shape of the data. In particular, on intervals where the data is monotonic so is the piecewise polynomial, and on intervals where the data is concave up or down so is the piecewise polynomial.

'cubic': The same as pchip.

An alternate way to interpolate these points is by using the two commands

```
≫  pp = spline(x, y)
≫  yvalues = ppval(pp, xvalues)
```

to generate and interpolate the cubic spline or

```
≫  pp = pchip(x, y)
≫  yvalues = ppval(pp, xvalues)
```

to generate and interpolate the piecewise cubic Hermite polynomials. The first command generates the structure pp which contains all the information required to obtain a unique piecewise polynomial. The second command interpolates the piecewise polynomial at the x values given by the vector xvalues.

Interpolation really means *interpolation*. If a value lies outside the interval $[x_1, x_n]$ then, by default, NaN is returned. This can be changed by adding a fifth argument:

- If the fifth argument is a number, this value is returned whenever the value lies outside the interval.
- If the fifth argument is 'extrap', extrapolation (using the same method) is used.

The command spline can be used instead of using interp1 with the method spline. With it you can specify precisely the boundary conditions to use. Similarly, the command pchip can be used instead of using interp1 with the method pchip or cubic.

Polynomial Functions

interp1(x, y, xvalues, <method>)	Interpolates any number of values using the given data points and the given method.
interp2	Interpolates in two dimensions.
interp3	Interpolates in three dimensions.
interpn	Interpolates in n dimensions.
pchip	Cubic Hermite interpolation.
poly(<roots>)	Calculates the coefficients of a polynomials given its roots.
polyder(q)	Calculates the derivative of a polynomial given the vector of the coefficients of the polynomial.
polyfit(x, y, <order>)	Calculates the coefficients of the least-squares polynomial of a given order which is fitted to the data $\{ (x_i, y_i) \}$.
polyval(q, x)	Evaluates the polynomial p(x).
polyvalm(q, A)	Evaluates the polynomial p(A) where A is a square matrix.
ppval	evaluates the piecewise polynomial calculated by pchip or spline.
roots(q)	Numerically calculates all the zeroes of a polynomial given the vector of the coefficients of the polynomial.
spline	Cubic spline interpolation.
vander	Generates the Vandermonde matrix.

13. Numerical Operations on Functions

MATLAB can also find a zero of a function by

```
≫  fzero(<function handle>, x0)

≫  fzero(<function handle>, x0)
```

x0 is a guess as to the location of the zero. Alternately,

```
≫  fzero(<function handle>, [xmin xmax])
```

finds a zero in the interval $x \in$ (xmin, xmax) where the signs of the function must differ at the endpoints of the interval.

Note: The function must cross the x-axis so that, for example, fzero cannot find the zero of the function $f(x) = x^2$.

The full argument list is

```
≫ fzero(<function handle>, xstart, <options>)
```

where xstart is either x0 or [xmin xmax], as we discussed previously. We can "tune" the zero finding algorithm by using the function optimset to create a structure which changes some of the default parameters for fzero. That is,

```
≫ opt = optimset('<Prop 1>', <Value 1>, '<Prop 2>', <Value 2>, ...)
```

changes the options included in the argument list and

```
≫ fzero(<function handle>, xstart, opt, <arg 1>, <arg 2>, ... )
```

executes fzero with the new options. Enter

```
≫ help optimset
```

for a discussion of how optimset works and

```
≫ optimset(@fzero)
```

to see the default parameters.

Frequently, the function will have parameters that need to be set. For example, we can find a zero of the function $f(x) = \cos ax + bx$ by using an anonymous function

```
≫ a = ???;
≫ b = ???;
≫ fcos_a = @(x) cos(a*x) + b*x;
≫ yzero = fzero(fcos_a, xstart);
```

or by using a nested function

```
function fzero_example(a, b, xstart)
yzero = fzero(@fcos_n, xstart);
   function y = fcos_n(x)
   % fcos: f(x) = cos(a*x) + b*x
   y = a*cos(x) + b*x;
   end
end
```

It sometimes happens that the function has already been coded in a separate file, i.e., it is a primary function m-file, such as

```
function y = fcos_p(x, a, b)
% fcos: f(x) = cos(a*x) + b*x
y = a*cos(x) + b*x;
```

Then we can use closure, as already discussed in section [Macro:[ode: basic]chap], so that the parameters can be set outside of fzero. This is easily done by entering

```
≫ a = ???;
≫ b = ???;
≫ fcos_c = @(x) fcos_p(x, a, b);
≫ yzero = fzero(fcos_p, xstart);
```

The parameters a and b are determined when the function fcos_c is generated and so are passed indirectly into fzero

MATLAB can also find a local minimum of a function of a single variable in an interval by

```
≫ fminbnd(<function handle>, xmin, xmax)
```

As with fzero, the full argument list is

```
≫ fminbnd(<function handle>, xmin, xmax, options)
```

MATLAB can also find a local minimum of a function of several variables by

```
≫ fminsearch(<function handle>, iterate0)
```

where iterate0 is a vector specifying where to begin searching for a local minimum. For example, if we enter

```
≫  fnctn = @(x) (x(1) - 1)^2 + (x(2) + 2)^4; ≫  fminsearch(fnctn, [0 0]′)
```

we obtain $(1.0000 - 2.0003)^{\mathrm{T}}$ (actually $(1.00000004979773, -2.00029751371046)^{\mathrm{T}}$). The answer might not seem to be very accurate. However, the value of the function at this point is 1.03×10^{-14}, which is quite small. If our initial condition is $(1, 1)^{\mathrm{T}}$, the result is $(0.99999998869692, -2.00010410231166)^{\mathrm{T}}$. Since the value of `funct` at this point is 2.45×10^{-16}, the answer is about as accurate as can be expected. In other words, the location of a zero and/or a local minimum of a function might not be as accurate as you might expect. **Be careful**. To determine the accuracy MATLAB is using to determine the minimum value type

```
≫  optimset(@fminsearch)
```

The value of `TolX`, the termination tolerance on `x`, is 10^{-4} and the value of `TolFun`, the termination tolerance on the function value, is the same.

There is no direct way to find zeroes of functions of more than one dimension. However, it can be done by using `fminsearch`. For example, suppose we want to find a zero of the function

$$\mathtt{f(x)} = \begin{pmatrix} x_1 + x_2 + \sin(x_1 - x_2) \\ x_1 - x_2 + 2\cos(x_1 + x_2) \end{pmatrix} .$$

Instead, we can find a minimum of $g(\mathtt{x}) = f_1^2(\mathtt{x}) + f_2^2(\mathtt{x})$. **If the minimum value is** 0, we have found a zero of `f` — if it is not zero, we have not found a zero of `f`. For example, if `f` is defined an an anonymous function the result of

```
≫  xmin = fminsearch(f, [0 0])
```

is $\mathtt{x}_{\min} = (-.1324\ldots, 1.0627\ldots)$. We are not done since we still have to calculate $g(\mathtt{x}_{\min})$. This is $\approx 2.4\times10^{-9}$ which is small — but is it small enough? We can decrease the termination tolerance by

```
≫  opt = optimset('TolX', 1.e-8, 'TolFun', 1.e-8)
≫  xmin = fminsearch(f, [0 0], opt)
```

Since $g(\mathtt{x}_{\min}) = 2.3\times10^{-17}$ we can assume that we have found a zero of `f`.

MATLAB can also calculate definite integrals using two functions. The first is `quad` which uses adaptive Simpson's method. To evaluate $\int_a^b f(x)\,dx$ by Simpson's method enter

```
≫  quad(<function handle>, a, b)
```

The full argument list is

```
≫  quad(<function handle>, a, b, tol, trace)
```

where `tol` sets the relative tolerance for the convergence test and information about each iterate is printed if `trace` is non-zero.

The second is `quadl` which uses adaptive Gauss-Lobatto quadrature, which is a variant of Gauss quadrature.

`quadl` uses the more accurate formula and so should require many fewer function evaluations. For example, `quad` calculates the exact integral (up to round-off errors) for polynomials of degree five whereas `quadl` calculates the exact integral (up to round-off errors) for polynomials of degree nine.

MATLAB can also calculate the double integral

$$\int_{x_{\min}}^{x_{\max}} \int_{y_{\min}}^{y_{\max}} f(x, y)\,dx\,dy$$

by

```
≫  dblquad(<function handle>, xmin, xmax, ymin, ymax)
```

It can also calculate the triple integral

$$\int_{x_{\min}}^{x_{\max}} \int_{y_{\min}}^{y_{\max}} \int_{z_{\min}}^{z_{\max}} f(x, y, z)\,dx\,dy\,dz$$

by

```
≫  triplequad(<function handle>, xmin, xmax, ymin, ymax, zmin, zmax)
```

Numerical Operations on Functions

`dblquad(<function handle>, a, b, c, d)`	Numerically evaluates a double integral.
`fminbnd(<function handle>, xmin, xmax)`	Numerically calculates a local minimum of a one-dimensional function given the endpoints of the interval in which to search
`fminsearch(<function handle>, iterate0)`	Numerically calculates a local minimum of a multi-dimensional function given the the initial iterate vector.
`fzero(<function handle>, x0)`	Numerically calculates a zero of a function given the initial iterate. `x0` can be replaced by a 2-vector of the endpoints of the interval in which a zero lies.
`optimset`	Allows you to modify the parameters used by `fzero`, `fminbnd`, and `fminsearch`.
`quad(<function handle>, a, b)`	Numerically evaluates an integral using Simpson's method.
`quadl(<function handle>, a, b)`	Numerically evaluates an integral using the adaptive Gauss-Lobatto method.

14. Discrete Fourier Transform

There are a number of ways to define the discrete Fourier transform; we choose to define it as the discretization of the continuous Fourier series. In this section we show exactly how to discretize the continuous Fourier series and how to transform the results of MATLAB's discrete Fourier transform back to the continuous case. We are presenting the material in such detail because there are a few slightly different definitions of the discrete Fourier transform; we present the definition which follows directly from the real Fourier series. xdi A "reasonable" continuous function f which is periodic with period T can be represented by the real trigonometric series

$$f(t) = a_0 + \sum_{k=1}^{\infty} \left(a_k \cos \frac{2\pi k t}{T} + b_k \sin \frac{2\pi k t}{T} \right) \quad \text{for all } t \in [0, T] \tag{14.1}$$

where

$$\left.\begin{aligned} a_0 &= \frac{1}{T} \int_0^T f(t)\, dt \\ a_k &= \frac{2}{T} \int_0^T f(t) \cos kt\, dt \\ b_k &= \frac{2}{T} \int_0^T f(t) \sin kt\, dt \end{aligned}\right\} \quad \text{for } k \in \mathbb{N}[1, \infty)\,.$$

The coefficients $a_0, a_1, a_2, \ldots$ and $b_1, b_2, \ldots$ are called the real Fourier coefficients of f, and a_k and b_k are the coefficients of the k-th mode. The *power* of the function $f(t)$ is[†]

$$P = \frac{1}{T} \int_0^T \big| f(t) \big|^2 \, dt$$

[†]The term "power" is a misnomer because the function f need not be related to a physical quantity for which the power makes any sense. However, we will stick to the common usage.

To understand the physical significance of power, we begin with the definition of work. Consider a particle which is under the influence of the constant force $\vec{F}$. If the particle moves from the point P_0 to P_1 then the work done to the particle is $\vec{F} \bullet \vec{r}$, where $\vec{r}$ is the vector from P_0 to P_1. The power of the particle is the work done per unit time, i.e., $\vec{F} \bullet \vec{v}$ where $\vec{v} = \vec{r}/t$.

Next, consider a charge q which is moving between two terminals having a potential difference of V. The

so that

$$P = |a_0|^2 + \frac{1}{2}\sum_{k=1}^{\infty}\big(\,|a_k|^2 + |b_k|^2\big)\ .$$

The power in each mode, i.e., the power spectrum, is

$$P_k = \begin{cases} |a_0|^2 & \text{if } k = 0 \\ \frac{1}{2}\big(\,|a_k|^2 + |b_k|^2\big) & \text{if } k > 0 \end{cases}$$

and the *frequency* of the k-th mode is k/T cycles per unit time.

Since

$$\cos\alpha t = \frac{e^{i\alpha t} + e^{-i\alpha t}}{2} \quad \text{and} \quad \sin\alpha t = \frac{e^{i\alpha t} - e^{-i\alpha t}}{2i}\ ,$$

we can rewrite the real Fourier series as the complex Fourier series

$$f(t) = a_0 + \sum_{k=1}^{\infty}\big[\,\tfrac{1}{2}(a_k - ib_k)e^{2\pi ikt/T} + \tfrac{1}{2}(a_k + ib_k)e^{-2\pi ikt/T}\big]$$

so that

$$f(t) = \sum_{k=-\infty}^{\infty} c_k e^{2\pi ikt/T} \quad \text{for all } t \in [0,T] \tag{14.2}$$

where

$$\left.\begin{aligned} c_0 &= a_0 \\ c_k &= \tfrac{1}{2}(a_k - ib_k) \\ c_{-k} &= \tfrac{1}{2}(a_k + ib_k) \end{aligned}\right\} \quad \text{for } k > 0\,. \tag{14.3}$$

The coefficients $\ldots, c_{-2}, c_{-1}, c_0, c_1, c_2, \ldots$ are called the complex Fourier coefficients of f, and c_k and c_{-k} are the coefficients of the k-th mode. (Note that these Fourier coefficients are generally complex.) We can also calculate c_k directly from f by

$$c_k = \frac{1}{T}\int_0^T f(t)e^{-2\pi ikt/T}\,dt \quad \text{for } k = \ldots, -2, -1, 0, 1, 2, \ldots$$

Note that if f is real, then $c_{-k} = c_k^*$ (by replacing k by $-k$ in the above equation). The power of $f(t)$ is

$$P = |c_0|^2 + \sum_{k=1}^{\infty}\big(\,|c_k|^2 + |c_{-k}|^2\big)$$

and the power in each mode is

$$P_k = \begin{cases} |c_0|^2 & \text{if } k = 0 \\ \big(\,|c_k|^2 + |c_{-k}|^2\big) & \text{if } k > 0\,. \end{cases}$$

---

work done on the charge is $W = qV = ItV$, where I is the current and t is the time it takes for the charge to move between the two terminals. If R is the resistance in the circuit, $V = IR$ and the power is

$$P = \frac{W}{t} = IV = I^2R = \frac{V^2}{R}\ .$$

Thus, if we consider $f(t)$ to be the voltage or the current of some signal, the instantaneous power in the signal is proportional to $f^2(t)$ and the average power is proportional to

$$\frac{1}{T}\int_0^T \big|\,f(t)\big|^2\,dt\,.$$

We can only calculate a finite number of Fourier coefficients numerically and so we truncate the infinite series at the M-th mode. We should choose M large enough that

$$f(t) \approx \sum_{k=-M}^{M} c_k e^{2\pi i k t/T} \quad \text{for all } t \in [0,T]\,.$$

There are now

$$N = 2M+1$$

unknowns (which is an odd number because of the $k=0$ mode). We require N equations to solve for these N unknown coefficients. We obtain these equations by requiring that the two sides of this approximation be equal at the N equally spaced abscissas $t_j = jT/N$ for $j = 0,1,2,\ldots,N-1$ (so that $0 = t_0 < t_1 < \cdots < t_{N-1} < t_N = T$).† That is,

$$f(t_j) = \sum_{k=-M}^{M} \gamma_k e^{2\pi i k t_j/T} \quad \text{for } j = 0,1,2,\ldots,N-1$$

or, written as a first-order system,

$$f_j = \sum_{k=-M}^{M} \gamma_k e^{2\pi i j k/N} \quad \text{for } j = 0,1,2,\ldots,N-1 \tag{14.4}$$

where $f_j \equiv f(t_j)$. This linear system can be solved to obtain

$$\gamma_k = \frac{1}{N}\sum_{j=0}^{N-1} f_j e^{-2\pi i j k/N} \quad \text{for } k = -M, -M+1, \ldots, M\,. \tag{14.5}$$

The reason we have replaced the coefficients $c_{-M}, c_{-M+1}, \ldots, c_{M-1}, c_M$ by $\gamma_{-M}, \gamma_{-M+1}, \ldots, \gamma_{M-1}, \gamma_M$ is that the c's are the coefficients in the continuous complex Fourier series, eq. (14.2), and are calculated by (14.3). The γ's are the coefficients in the discrete complex Fourier series, eq. (14.4), and are calculated by (14.5).

Note: To repeat: the discrete Fourier coefficient γ_k is a function of M, i.e., $\gamma_k(M)$, and is generally not equal to the continuous Fourier coefficient c_k. However, as $M \to \infty$ we have $\gamma_k(M) \to c_k$. For a fixed M we generally only have $\gamma_k(M) \approx c_k$ as long as $|k|$ is "much less than" M. Of course, it takes practice and experimentation to determine what "much less than" means.

We define the discrete Fourier series by

$$f_{\rm FS}(t) = \sum_{k=-M}^{M} \gamma_k e^{2\pi i k t/T} \quad \text{for all } t \in [0,T]\,.$$

It is our responsibility (using our experience) to choose M large enough that $f(t) \approx f_{\rm FS}(t)$. Given $\texttt{f} = (f_0, f_1, f_2, \ldots, f_{N-1})^{\rm T}$, the Fourier coefficients are calculated in MATLAB by

```
>>  fc = fft(f)/N
```

where the coefficients of the discrete Fourier transform are contained in `fc` in the order

$$\big(\gamma_0, \gamma_1, \ldots, \gamma_{M-1}, \gamma_M, \gamma_{-M}, \gamma_{-M+1}, \ldots, \gamma_{-2}, \gamma_{-1}\big)^{\rm T}\,.$$

The command `fftshift` changes the order to

$$\big(\gamma_{-M}, \gamma_{-M+1}, \ldots, \gamma_{-2}, \gamma_{-1}, \gamma_0, \gamma_1, \ldots, \gamma_{M-1}, \gamma_M\big)^{\rm T}\,.$$

†Note that t_N is not used because $f(t_N)$ has the same value as $f(t_0)$ and so does not provide us with an independent equation.

The original function, represented by the vector `f`, is recovered by

```
≫  f = N*ifft(fc)
```

and the order is changed by `ifftshift`

It is important to check the Fourier transform to make sure it returns the results we expect. We began with the real trigonometric series (14.1) and derived the complex trigonometric series (14.2) from it. The nonzero Fourier coefficients of $f(x) = \cos x = (e^{ix} + e^{-ix})/2$ are $a_1 = 1$ and $c_{-1} = c_1 = 1/2$, whereas the nonzero Fourier coefficients of $f(x) = \sin x = (e^{ix} - e^{-ix})/(2i)$ are $a_1 = 1$ and $c_{-1} = i/2$ but $c_1 = -i/2$. The code

```
≫  n = 9;
≫  x = linspace(0, 2*pi, n+1);
≫  x(n+1) = [];
≫  c_c = fft(cos(x));
≫  d_c = fftshift(c_c);
≫  c_s = fft(sin(x));
≫  d_s = fftshift(c_s);
≫  ci_c = ifft(cos(x));
≫  di_c = ifftshift(c_c);
≫  ci_s = ifft(sin(x));
≫  di_s = ifftshift(c_s);
```

returns the vectors

$c_c = (0, 4.5, 0, 0, 0, 0, 0, 0, 4.5)$,
$d_c = (0, 0, 0, 4.5, 0, 4.5, 0, 0, 0)$,
$c_s = (0, -4.5i, 0, 0, 0, 0, 0, 0, 4.5i)$,
$d_s = (0, 0, 0, 4.5i, 0, -4.5i, 0, 0, 0)$,
$c_c^{(i)} = (0, 0.5, 0, 0, 0, 0, 0, 0, 0.5)$,
$d_c^{(i)} = (0, 0, 0, 0, 0.5, 0, 0.5, 0, 0)$,
$c_s^{(i)} = (0, 0.5i, 0, 0, 0, 0, 0, 0, -0.5i)$, and
$d_s^{(i)} = (0, 0, 0, 0, -0.5i, 0, 0.5i, 0, 0)$.

Notice that `fft` and `ifft` both return the correct coefficients for $\cos x$ (up to the scaling of n), but only the `fft` returns the correct coefficients for $\sin x$. Thus, the command `fft` is correct, but it multiplies the coefficients by N.

Also, notice that `fftshift` correctly shifts the coefficients, whereas `ifftshift` does not — but `ifftshift` correctly shifts the coefficients back. That is, `ifftshift` is the inverse of `fftshift` so `ifftshift(fftshift(c_s)) = c_s`.

Warning: One of the most common mistakes in using `fft` is forgetting that the input is in the order

$$f_0, f_1, f_2, \ldots, f_{N-1}$$

while the output is in the order

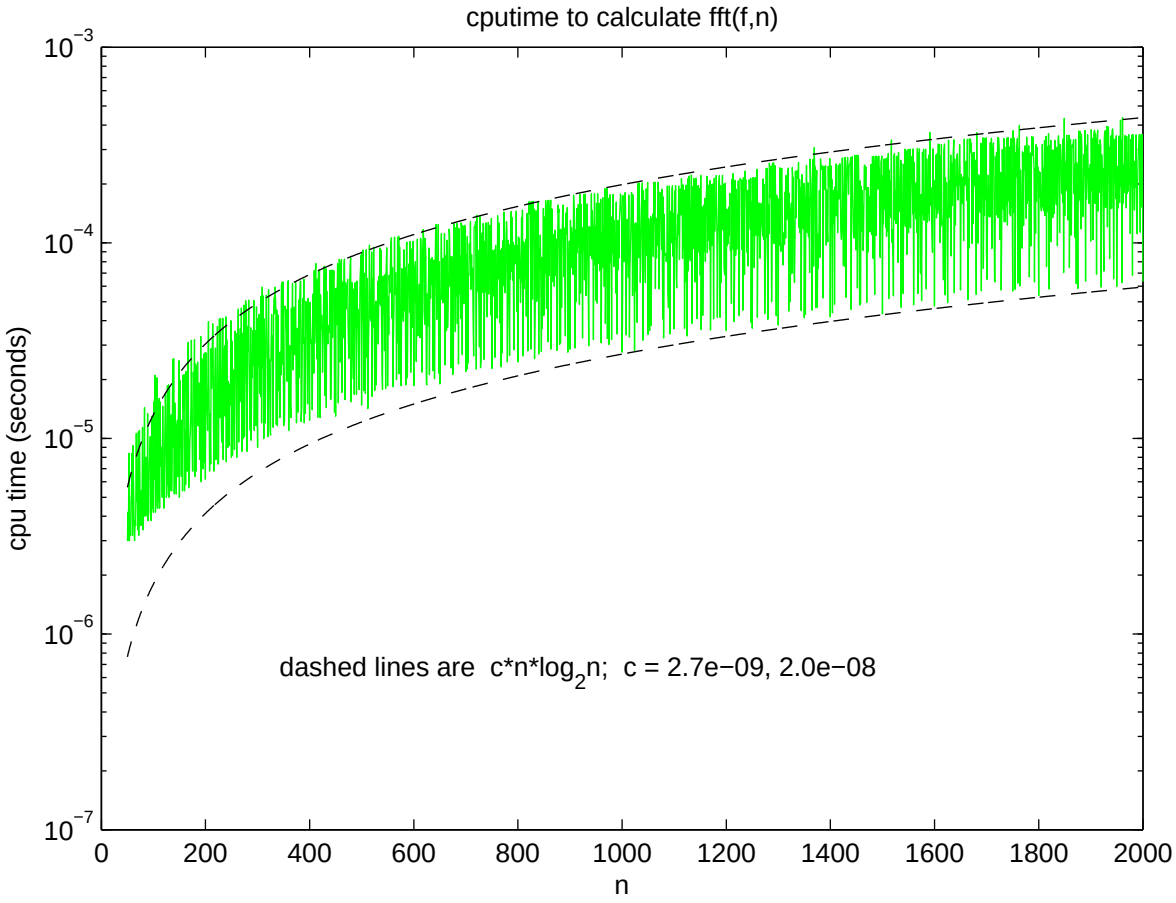
$$\gamma_0, \gamma_1, \ldots, \gamma_{M-1}, \gamma_M, \gamma_{-M}, \gamma_{-M+1}, \ldots, \gamma_{-2}, \gamma_{-1} \ ,$$

not

$$\gamma_{-M}, \gamma_{-M+1}, \ldots, \gamma_{-2}, \gamma_{-1}\gamma_0, \gamma_1, \ldots, \gamma_{M-1}, \gamma_M \ .$$

There is only one difficulty with our presentation. As we have already stated, the vector `f` has $N = 2M + 1$ elements, which is an *odd* number. The Fast Fourier Transform (FFT, for short), which is the method used to calculate the discrete Fourier coefficients by `fft` and also to recover the original function by `ifft`, generally works faster if the number of elements of `f` is even, and is particularly fast if it a power of 2.

The figure below shows the cputime needed to calculate `fft(f)` as a function of N. Since the vertical axis is logarithmic, it is clear that there is a **huge** difference in the time required as we vary N.The dashed lines show the minimum and maximum asymptotic times as $cn\log_2 n$.



For N to be even, we have to drop one coefficient, and the one we drop is γ_M. Now

$$N = 2M$$

is even. The discrete complex Fourier series is

$$f_{\mathrm{FS}}(t) = \sum_{k=-M}^{M-1} \gamma_k e^{2\pi i k t/T} \quad \text{for all } t \in [0,T]$$

and the discrete Fourier coefficients are calculated by

$$\gamma_k = \frac{1}{N}\sum_{j=0}^{N-1} f_j e^{-2\pi i j k/N} \quad \text{for } k = -M, -M+1, \ldots, M-2, M-1\,.$$

As before, given $\mathtt{f} = (f_0, f_1, f_2, \ldots, f_{N-1})^{\mathrm{T}}$, the Fourier coefficients are calculated by

```
≫  fc = fft(f)/N
```

The coefficients of the discrete Fourier transform are now contained in `fc` as

$$\mathtt{fc} = \left(\gamma_0, \gamma_1, \ldots, \gamma_{M-2}, \gamma_{M-1}, \gamma_{-M}, \gamma_{-M+1}, \ldots, \gamma_{-2}, \gamma_{-1}\right)^{\mathrm{T}}.$$

The original function, represented by the vector `f`, is again recovered by

```
≫  f = N*ifft(fc)
```

Note: Since there are now an even number of Fourier coefficients, we can reorder them by using `fftshift`, which switches the first half and the last half of the elements. The result is

$$\texttt{fftshift(fc)} = \left(\gamma_{-M}, \gamma_{-M+1}, \ldots, \gamma_{-2}, \gamma_{-1}, \gamma_0, \gamma_1, \ldots, \gamma_{M-2}, \gamma_{M-1},\right)^{\mathrm{T}}.$$

Also, `ifftshift` is the same as `fftshift` if N is even.

Warning: Remember that if you reorder the elements of `fc` by

```
≫  fc_shift = fftshift(fc)
```

you will have to "unorder" the elements by applying

```
≫  fc = fftshift(fc_shift)
```

again before you use `ifft`.

Note: When N is even we cannot recover γ_M and so we only know one of the two coefficients of the M-th mode. Thus, we cannot determine the M-th mode correctly. Although we cannot give a simple example, it occasionally happens that this causes difficulties. The solution is to set $\gamma_{-M} = 0$ so that the M-th mode is dropped completely.

Here is a simple example of the use of Fourier coefficients from *The Student Edition of MATLAB: User's Guide*. We begin with a signal at 50 and 120 hertz (cycles per unit time), `y0`, and then we perturb it by adding Gaussian noise, `ypert`. We plot the periodic unperturbed signal, and then the perturbed signal, vs. time. (If you enter all these commands into an m-file, put a `pause` command between each of the `plot` commands.)

```
≫  time = .6;
≫  N = 600;
≫  t = linspace(0, time, N);
≫  y0 = sin(2*pi*50*t) + sin(2*pi*120*t);     % unperturbed signal
≫  ypert = y0 + 2*randn(size(t));     % perturbed signal
≫  figure(1)
≫  plot(t, y0, 'r'), axis([0 time -8 8])
≫  hold on
≫  plot(t, ypert, 'g')
```

Clearly, once the random noise has been added, the original signal has been completely lost — or has it.

We now look at the Fourier spectrum of `y0` by plotting the power at each frequency. First, we plot the unperturbed power, `power0`, and then the perturbed power, `powerpert`, vs. the frequency at each mode, `freq`. The two spikes in the plot of the unperturbed power are precisely at 50 and 120 hertz, the signature of the two sine functions in `y0`. (For simplicity in the discussion, we have deleted the power in the M-th mode by `fc(N/2 +1) = []` so that `power0(k)` is the power in the $k-1$-st mode.)

```
≫  fc0 = fft(y0)/N;     % Fourier spectrum of unperturbed signal
≫  figure(2)
≫  fc0(N/2 +1) = [];     % delete k = N/2 +1 mode
≫  power0(1) = abs(fc0(1)).^2;
≫  power0(2:N/2) = abs(fc0(2:N/2)).^2 + abs(fc0(N-1:-1:N/2 +1)).^2;
≫  freq = [1:N]'/time;     % the frequency of each mode
≫  plot(freq(1:N/2), power0, 'r'), axis([0 freq(N/2) 0 .5])
≫  fcpert = fft(ypert)/N;     % Fourier spectrum of perturbed signal
≫  hold on
≫  powerpert(1) = abs(fcpert(1)).^2;
≫  powerpert(2:N/2) = abs(fcpert(2:N/2)).^2 + abs(fcpert(N-1:-1:N/2 +1)).^2;
≫  plot(freq(1:N/2), powerpert, 'g')
```

Clearly, the original spikes are still dominant, but the random noise has excited every mode.

To see how much power is in the unperturbed signal and then the perturbed signal, enter

```
≫  sum(power0)
≫  sum(powerpert)
```

The perturbed signal has about five times as much power as the original signal, which makes clear how large the perturbation is.

Let us see if we can reconstruct the original signal by removing any mode whose magnitude is "small". By looking at the power plots, we see that the power in all the modes, except for those corresponding to the spikes, have an amplitude $\lessapprox 0.1$. Thus, we delete any mode of the perturbed Fourier spectrum, i.e., `fcpert`, whose power is less than this value; we call this new Fourier spectrum `fcchop`. We then construct a new signal `ychop` from this "chopped" Fourier spectrum and compare it with the original unperturbed signal.

```
≫ fcchop = fcpert;     % initialize the chopped Fourier spectrum
≫ ip = zeros(size(fcpert));            % construct a vector with 0's
≫ ip(1:N/2) = ( powerpert > 0.1 );     % where fcchop should be
≫ ip(N:-1:N/2 +2) = ip(2:N/2);         % zeroed out
≫ fcchop( find(~ip) ) = 0;     % zero out "small" modes
≫ ychop = real( N*ifft(fcchop) );      % signal of "chopped" Fourier spectrum
≫ figure(1)
≫ plot(t, ychop, 'b')
```

(`ychop` is the real part of `N*ifft(fcchop)` because, due to round-off errors, the inverse Fourier transform returns a "slightly" complex result.) The result is remarkably good considering the size of the perturbation. If you have trouble comparing `y0` with `ychop`, reenter

```
≫ plot(t, y0, 'r')
```

Discrete Fourier Transform

`fft(f)`	The discrete Fourier transform of `f`.
`ifft(fc)`	The inverse discrete Fourier transform of the Fourier coefficients `fc`.
`fftshift(fc)`	Switches the first half and the second half of the elements of `fc`.
`ifftshift(cf)`	Unswitches the first half and the second half of the elements of `fc`. (`fftshift` and `ifftshift` are the same if the number of elements is even.

15. Mathematical Functions Applied to Matrices

As we briefly mentioned in subsection 2.7, mathematical functions can generally only be applied to square matrices. For example, if $\mathtt{A} \in \mathbb{C}^{n\times n}$ then $e^{\mathtt{A}}$ is defined from the Taylor series expansion of e^a. That is, since

$$e^a = 1 + \frac{a}{1!} + \frac{a^2}{2!} + \frac{a^3}{3!} + \cdots$$

we define $e^{\mathtt{A}}$ to be

$$e^{\mathtt{A}} = 1 + \frac{\mathtt{A}}{1!} + \frac{\mathtt{A}^2}{2!} + \frac{\mathtt{A}^3}{3!} + \cdots .$$

(Thus, if $\mathtt{A} \in \mathbb{C}^{m\times n}$ where $m \neq n$ then $e^{\mathtt{A}}$ does not exist because $\mathtt{A}^k$ does not exist if $\mathtt{A}$ is not a square matrix.)

If $\mathtt{A}$ is a square diagonal matrix $e^{\mathtt{A}}$ is particularly simple to calculate since

$$\mathtt{A}^p = \begin{pmatrix} a_{11} & & & \\ & a_{22} & & \\ & & \ddots & \\ 0 & & & a_{nn} \end{pmatrix}^p = \begin{pmatrix} a_{11}^p & & & \\ & a_{22}^p & & \\ & & \ddots & \\ 0 & & & a_{nn}^p \end{pmatrix}.$$

Thus,

$$e^{\mathtt{A}} = \begin{pmatrix} e^{a_{11}} & & & & \\ & e^{a_{22}} & & & \\ & & \ddots & & \\ & & & e^{a_{n-1,n-1}} & \\ 0 & & & & e^{a_{nn}} \end{pmatrix} .$$

The MATLAB command

```
>> expm(A)
```

calculates $e^{\mathtt{A}}$ if `A` is a square matrix. (Otherwise, it generates an error message.)

A simple example where $e^{\mathtt{A}}$ occurs is in the solution of first-order ode systems with constant coefficients. Recall that the solution of

$$\frac{dy}{dt}(t) = ay(t) \quad \text{for } t \geq 0 \text{ with } y(0) = y_{\text{ic}}$$

is

$$y(t) = y_{\text{ic}} e^{at} .$$

Similarly, the solution of

$$\frac{d}{dt}\begin{pmatrix} y_1(t) \\ y_2(t) \\ \vdots \\ y_n(t) \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} y_1(t) \\ y_2(t) \\ \vdots \\ y_n(t) \end{pmatrix} \quad \text{for } t \geq 0 \text{ with } \mathtt{y}(0) = \mathtt{y}_{\text{ic}}$$

i.e., $\mathtt{y}'(t) = \mathtt{A}\mathtt{y}(t)$, is

$$\mathtt{y}(t) = e^{\mathtt{A}t}\mathtt{y}_{\text{ic}} .$$

To calculate $\mathtt{y}(t)$ for any time t, you only need enter

```
>> expm(A*t) * yic
```

Note: The above statement gives the *exact* solution to the ode system at `t` $= 10$ by

```
>> expm(A*10) * yic
```

You could also use numerical methods, as discussed in section 10, to solve it. However, you would have to solve the ode for all `t` $\in [0, 10]$ in order to obtain a numerical approximation at the final time. This would be much more costly than simply using the analytical solution.

Similarly, $\sqrt{\mathtt{B}}$ is calculated in MATLAB by entering

```
>> sqrtm(A)
```

Finally, $\log \mathtt{B}$ is calculated in MATLAB by entering

```
>> logm(A)
```

These are the only explicit MATLAB commands for applying mathematical functions to matrices. However, there is a general matrix function for the other mathematical functions. The command

```
>> funm(A, <function handle>)
```

evaluates `<function name>(A)` for the MATLAB functions `exp`, `sin`, `cos`, `sinh`, and `cosh` as well as user-defined functions.

Matrix Functions

`expm(A)`	Calculates $e^{\mathtt{A}}$ where `A` must be a square matrix.
`sqrtm(A)`	Calculates $\sqrt{\mathtt{A}}$ where `A` must be a square matrix.
`logm(A)`	Calculates $\log \mathtt{A}$ where `A` must be a square matrix.
`funm(A, <function handle>)`	Calculates `<function name>(A)` where `A` must be a square matrix.

Appendix: Reference Tables

These tables summarize the functions and operations described in this tutorial. The number (or numbers) shown give the page number of the table where this entry is discussed.

Arithmetical Operators

`+`	Addition. (p. 7, 28)
`-`	Subtraction. (p. 7, 28)
`*`	Scalar or matrix multiplication. (p. 7, 28)
`.*`	Elementwise multiplication of matrices. (p. 28)
`/`	Scalar division. (p. 7, 28)
`./`	Elementwise division of matrices. (p. 28)
`\`	Scalar left division, i.e., $b\backslash a = a/b$. (p. 7)
`\`	The solution to `Ax` $= $ `b` for `A` $\in \mathbb{C}^{m\times n}$: when $m = n$ and `A` is nonsingular this is the solution Gaussian elimination; when $m > n$ this is the least-squares approximation of the overdetermined system; when $m < n$ this is a solution of the underdetermined system. (p. 28, 62)
`.\`	Elementwise left division of matrices i.e., `B.\A` $=$ `A./B`. (p. 28)
`^`	Scalar or matrix exponentiation. (p. 7, 28)
`.^`	Elementwise exponentiation of matrices. (p. 28)

Special Characters

`:`	Creates a vector by `a:b` or `a:c:b`; subscripts matrices. (p. 24)
`;`	Ends a statement without printing out the result; also, ends each row when entering a matrix. (p. 9)
`,`	Ends a statement when more than one appear on a line and the result is to be printed out; also, separates the arguments in a function; also, can separate the elements of each row when entering a matrix. (p. 9)
`...`	Continues a MATLAB command on the next line. (p. 14)
`%`	Begins a comment. (p. 14)
↑	The up-arrow key moves backward in the MATLAB workspace, one line at a time. (p. 7)

Getting Help

`demo`	Runs demonstrations of many of the capabilities of MATLAB. (p. 16, 53)
`doc`	On-line reference manual. (p. 16)
`help`	On-line help. (p. 16)
`helpdesk`	Loads the main page of the on-line reference manual. (p. 16)
`load`	Loads back all of the variables which have been saved previously. (p. 16)
`lookfor`	Searches all MATLAB commands for a keyword. (p. 16)
`save`	Saves all of your variables. (p. 16)
`type`	Displays the actual MATLAB code. (p. 14, 16)
`who`	Lists all the current variables. (p. 16)
`whos`	Lists all the current variables in more detail than `who`. (p. 16)
`^C`	Abort the command which is currently executing (i.e., hold down the control key and type "c"). (p. 16)

Predefined Variables

`ans`	The default variable name when one has not been specified. (p. 9)
`pi`	π. (p. 9)
`eps`	Approximately the smallest positive real number on the computer such that `1 + eps` $\neq$ `1`. (p. 9)
`Inf`	∞ (as in $1/0$). (p. 9)
`NaN`	Not-a-Number (as in $0/0$). (p. 9)
`i`	$\sqrt{-1}$. (p. 9)
`j`	$\sqrt{-1}$. (p. 9)
`realmin`	The smallest "usable" positive real number on the computer. (p. 9)
`realmax`	The largest "usable" positive real number on the computer. (p. 9)

Format Options

`format short`	The default setting. (p. 11)
`format long`	Results are printed to approximately the maximum number of digits of accuracy in MATLAB. (p. 11)
`format short e`	Results are printed in scientific notation. (p. 11)
`format long e`	Results are printed in scientific notation to approximately the maximum number of digits of accuracy in MATLAB. (p. 11)

Input-Output Functions

`csvread`	Reads data into MATLAB from the named file, one row per line of input. (p. 47)
`csvwrite`	Writes out the elements of a matrix to the named file using the same format as `csvread`. (p. 47)
`diary`	Saves your input to MATLAB and most of the output from MATLAB to disk. (p. 7)
`fopen`	Opens the file with the permission string determining how the file is to be accessed. (p. 63)
`fclose`	Closes the file. (p. 63)
`fscanf`	Behaves very similarly to the C command in reading data from a file using any desired format. (p. 63)
`fprintf`	Behaves very similarly to the C command in writing data to a file using any desired format. It can also be used to display data on the screen. (p. 63, 63)
`input`	Displays the prompt on the screen and waits for you to enter whatever is desired. (p. 10)
`load`	Reads data into MATLAB from the named file, one row per line of input. (p. 47)
`print`	Prints a plot or saves it in a file using various printer specific formats. (p. 47)

Some Common Mathematical Functions

`abs`	Absolute value (p. 12, 13)
`acos`	Inverse cosine. (p. 12)
`acosd`	Inverse cosine (result in degrees). (p. 12)
`acosh`	Inverse hyperbolic cosine. (p. 12)
`acot`	Inverse cotangent. (p. 12)
`acotd`	Inverse cotangent (result in degrees). (p. 12)
`acoth`	Inverse hyperbolic cosine. (p.)
`acsc`	Inverse cosecant. (p. 12)
`acscd`	Inverse cosecant (result in degrees). (p. 12)
`angle`	Phase angle of a complex number. (p. 13)
`asec`	Inverse secant. (p.)
`asec`	Inverse secant (result in degrees). (p.)
`asin`	Inverse sine. (p. 12)
`asin`	Inverse sine (result in degrees). (p. 12)
`asinh`	Inverse hyperbolic sine. (p. 12)
`atan`	Inverse tangent. (p. 12)
`atand`	Inverse tangent (result in degrees). (p. 12)
`atan2`	Inverse tangent using two arguments. (p. 12)
`atanh`	Inverse hyperbolic tangent. (p. 12)
`ceil`	Round upward to the nearest integer. (p. 12)
`conj`	Complex conjugation. (p. 13)
`cos`	Cosine. (p. 12)
`cosd`	Cosine (argument in degrees). (p. 12)
`cosh`	Hyperbolic cosine. (p. 12)
`cot`	Cotangent. (p. 12)
`cotd`	Cotangent (argument in degrees). (p. 12)
`csc`	Cosecant. (p. 12)
`cscd`	Cosecant (argument in degrees). (p. 12)
`exp`	Exponential function. (p. 12)
`factorial`	Factorial function. (p. 12)
`fix`	Round toward zero to the nearest integer. (p. 12)
`floor`	Round downward to the nearest integer. (p. 12)
`imag`	The imaginary part of a complex number. (p. 13)
`log`	The natural logarithm, i.e., to the base e. (p. 12)
`log10`	The common logarithm, i.e., to the base 10. (p. 12)
`mod`	The modulus after division. (p. 12)
`real`	The real part of a complex number. (p. 13)
`rem`	The remainder after division. (p. 12)
`round`	Round to the closest integer. (p. 12)
`sec`	Secant. (p. 12)
`secd`	Secant (argument in degrees). (p. 12)
`sign`	The sign of the real number. (p. 12)
`sin`	Sine. (p. 12)
`sind`	Sine (argument in degrees). (p. 12)
`sinh`	Hyperbolic sine. (p. 12)
`sqrt`	Square root. (p. 12)
`tan`	Tangent. (p. 12)
`tand`	Tangent (argument in degrees). (p. 12)
`tanh`	Hyperbolic tangent. (p. 12)

Arithmetical Matrix Operations

`A + B`	Matrix addition. (p. 7, 28)
`A - B`	Matrix subtraction. (p. 7, 28)
`A*B`	Matrix multiplication. (p. 7, 28)
`A^n`	Matrix exponentiation. (p. 7, 28)
`A\b`	The solution to `Ax` = `b` by Gaussian elimination when `A` is a square non-singular matrix. (p. 28, 62)
`A\B`	The solution to `AX` = `B` by Gaussian elimination. (p. 28)
`b/A`	The solution to `xA` = `b` **where** `x` **and** `b` **are row vectors**. (p. 7, 28)
`B/A`	The solution to `XA` = `B`. (p. 28)

`A.*B`	Elementwise multiplication. (p. 28)
`A.^p`	Elementwise exponentiation. (p. 28)
`p.^A`	
`A.^B`	
`A./B`	Elementwise division. (p. 28)
`B.\A`	Elementwise left division, i.e., `B.\A` is exactly the same as `A./B`. (p. 28)

Elementary Matrices

`eye`	Generates the identity matrix. (p. 20)
`false`	Generates a logical matrix with all elements having the value false. (p. 74)
`ones`	Generates a matrix with all elements being 1. (p. 20)
`rand`	Generates a matrix whose elements are uniformly distributed random numbers in the interval $(0, 1)$. (p. 20)
`randn`	Generates a matrix whose elements are normally (i.e., Gaussian) distributed random numbers with mean 0 and standard deviation 1. (p. 20)
`randperm(n)`	Generates a random permutation of the integers $1, 2, \ldots, n$. (p.)
`speye`	Generates a Sparse identity matrix. (p. 94)
`sprand`	Sparse uniformly distributed random matrix. (p. 94, 94)
`sprandsym`	Sparse uniformly distributed symmetric random matrix; the matrix can also be positive definite. (p. 94)
`sprandn`	Sparse normally distributed random matrix. (p. 94)
`true`	Generates a logical matrix with all elements having the value true. (p. 74)
`zeros`	Generates a zero matrix. (p. 20)

Specialized Matrices

`hilb`	Generates the hilbert matrix. (Defined on p. 60.)
`vander`	Generates the Vandermonde matrix. (Defined on p. 110.)

Elementary Matrix Operations

`size`	The size of a matrix. (p. 20)
`length`	The number of elements in a vector. (p. 20)
`.'`	The transpose of a matrix. (p. 20)
`'`	The conjugate transpose of a matrix. (p. 20)

Manipulating Matrices

`cat`	Concatenates arrays; this is useful for putting arrays into a higher-dimensional array. (p. 34)
`clear`	Deletes a variable **or all the variables. This is a very dangerous command.** (p. 9)
`diag`	Extracts or creates diagonals of a matrix. (p. 24)
`fliplr`	Flips a matrix left to right. (p. 24)
`flipud`	Flips a matrix up and down. (p. 24)
`spdiags`	Generates a sparse matrix by diagonals. (p. 94)
`repmat`	Tiles a matrix with copies of another matrix. (p. 24)
`reshape`	Reshapes the elements of a matrix. (p. 24)
`rot90`	Rotates a matrix a multiple of $90°$. (p. 24)
`squeeze`	Removes (i.e., squeezes out) dimensions which only have one element. (p. 34)
`triu`	Extracts the upper triangular part of a matrix. (p. 24)
`tril`	Extracts the lower triangular part of a matrix. (p. 24)
`[]`	The null matrix. This is also useful for deleting elements of a vector and rows or columns of a matrix. (p. 24)

Odds and Ends

`path`	Viewing and changing the search path. (p.)
`cputime`	Approximately the CPU time (in seconds) used during this session. (p. 28)
`tic, toc`	Returns the elapsed time between these two commands. (p. 28)
`pause`	Halts execution until you press some key. (p. 87)
`rats`	Converts a floating-point number to a "close" rational number, which is frequently the exact value. (p. 62)

Two-Dimensional Graphics

`plot`	Plots the data points in Cartesian coordinates. (p. 48)
`fill`	Fills one or more polygons. (p. 53)
`semilogx`	The same as `plot` but the x axis is logarithmic. (p. 48)
`semilogy`	The same as `plot` but the y axis is logarithmic. (p. 48)
`loglog`	The same as `plot` but both axes are logarithmic. (p. 48)
`ezplot`	Generates an "easy" plot (similar to `fplot`). It can also plot a parametric function, i.e., $\big(x(t), y(t)\big)$, or an implicit function, i.e., $f(x, y) = 0$. (p. 48)
`polar`	Plots the data points in polar coordinates. (p. 48)
`ezpolar`	Generates an "easy" polar plot. (p. 48)
`linspace`	Generates equally-spaced points, similar to the colon operator. (p. 48)
`xlabel`	Puts a label on the x-axis. (p. 48)
`ylabel`	Puts a label on the y-axis. (p. 48)
`title`	Puts a title on the top of the plot. (p. 48)
`axis`	Controls the scaling and the appearance of the axes. (p. 48)
`hold`	Holds the current plot or release it. (p. 48)
`hist`	Plots a histogram. (p. 48)
`errorbar`	Plots a curve through data points and also the error bar at each data point. (p. 48)
`subplot`	Divides the graphics window into rectangles and moves between them. (p. 48, 50)

Three-Dimensional Graphics

`plot3`	Plots the data points in Cartesian coordinates. (p. 50)
`ezplot3`	Generates an "easy" plot in 3-D. (p. 50)
`fill3`	Fills one or more 3D polygons. (p. 53)
`mesh`	Plots a 3-D surface using a wire mesh. (p. 50)
`ezmesh`	Generates an "easy" 3-D surface using a wire mesh. (p. 50)
`surf`	Plots a 3-D filled-in surface. (p. 50)
`ezsurf`	Generates an "easy" 3-D filled-in surface. (p. 50)
`view`	Changes the viewpoint of a 3-D surface plot. (p. 50)
`meshgrid`	Generates a 2-D grid. (p. 50)
`zlabel`	Puts a label on the z-axis. (p. 50)
`axis`	Controls the scaling and the appearance of the axes. (p. 50)
`contour`	Plots a contour looking down the z axis. (p. 50)
`ezcontour`	Generates an "easy" contour looking down the z axis. (p. 50)
`contour3`	Plots a contour in 3-D. (p. 50)
`ezcontour3`	Generates an "easy" contour in 3-D. (p. 50)
`subplot`	Divides the graphics window into rectangles and moves between them. (p. 48, 50)
`colorbar`	Adds a color bar showing the correspondence between the value and the color. (p. 53)
`colormap`	Determines the current color map or choose a new one. (p. 53)

Advanced Graphics Features

`clf`	Clear a figure (i.e., delete everything in the figure (p. 53)
`demo`	Runs demonstrations of many of the capabilities of MATLAB. (p. 16, 53)
`figure`	Creates a new graphics window and makes it the current target. (p. 53)
`fplot`	Plots the specified function within the limits given. (p. 48)
`gtext`	Places the text at the point given by the mouse. (p. 53)
`image`	Plots a two-dimensional image. (p. 53)
`legend`	Places a legend on the plot. (p. 53)
`text`	Adds the text at a particular location. (p. 53)
`ginput`	Obtains the current cursor position. (p. 53)
`get`	Returns the current value of the property of an object. (p. 55)
`set`	Sets the value of the property, or properties of an object. (p. 55)
`gca`	The current axes handle. (p. 55)
`gcf`	The current figure handle. (p. 55)

String Functions, Cell Arrays, and Structures

`num2str`	Converts a variable to a string. (p. 41)
`sprintf`	Behaves very similarly to the C command in writing data to a text variable using any desired format. (p. 41)
`sscanf`	Behaves very similarly to the C command in reading data from a text variable using any desired format. (p. 41)
`str2num`	Converts a string to a variable. (p. 41)
`strcmp`	Compares strings. (p. 71)
`cell`	Preallocate a cell array of a specific size. (p. 41)
`celldisp`	Display all the contents of a cell array. (p. 41)
`struct`	Create a structure. (p. 41)

Data Manipulation Commands

`errorbar`	Plots a curve through data points and also the error bar at each data point. (p. 48)
`hist`	Plots a histogram of the elements of a vector. (p. 48)
`max`	The maximum element of a vector. (p. 33)
`min`	The minimum element of a vector. (p. 33)
`mean`	The mean, or average, of the elements of a vector. (p. 33)
`norm`	The norm of a vector or a matrix. (p. 33)
`prod`	The product of the elements of a vector. (p. 33)
`sort`	Sorts the elements of a vector in increasing order. (p. 33)
`std`	The standard deviation of the elements of a vector. (p. 33)
`sum`	The sum of the elements of a vector. (p. 33)

Some Useful Functions in Linear Algebra

`chol`	Calculates the Cholesky decomposition of a symmetric, positive definite matrix. (p. 69)
`cond`	Calculates the condition number of a matrix. (p. 69)
`condest`	Calculates a lower bound to the condition number of a square matrix. (p. 69)
`det`	Calculates the determinant of a square matrix. (p. 69)
`eig`	Calculates the eigenvalues, and eigenvectors if desired, of a square matrix. (p. 69)
`eigs`	Calculates some eigenvalues and eigenvectors of a square matrix. (p. 69)
`inv`	Calculates the inverse of a square invertible matrix. (p. 69)
`linsolve`	Solve a square matrix equation where the matrix can have certain properties to increase the CPU time. (p. 59)
`lu`	Calculates the LU decomposition of a square invertible matrix. (p. 69)
`norm`	Calculates the norm of a vector or matrix. (p. 69)
`null`	Calculates an orthonormal basis for the null space of a matrix. (p. 69)
`orth`	Calculates an orthonormal basis for the range of a matrix. (p. 69)
`pinv`	Calculates the pseudoinverse of a matrix. (p. 62)
`qr`	Calculates the QR decomposition of a matrix. (p. 69)
`rank`	Estimates the rank of a matrix. (p. 69)
`rref`	Calculates the reduced row echelon form of a matrix. (p. 59)
`svd`	Calculates the singular value decomposition of a matrix. (p. 69)

Logical and Relational Operators

`&`	Logical AND. (p. 72)	<	Less than. (p. 71)
`|`	Logical OR. (p. 72)	<=	Less than or equal to. (p. 71)
`~`	Logical NOT. (p. 72)	==	Equal. (p. 71)
`xor`	Logical EXCLUSIVE OR. (p. 72)	>	Greater than. (p. 71)
`&&`	A short-circuiting logical AND. (p. 72)	>=	Greater than or equal to. (p. 71)
`||`	A short-circuiting logical OR. (p. 72)	~=	Not equal to. (p. 71)
		`strcmp`	Comparing strings. (p. 71)

Flow Control

`break`	Terminates execution of a `for` or `while` loop. (p. 74)
`case`	Part of the `switch` command. (p. 74)
`continue`	Begins the next iteration of a `for` or `while` loop immediately. (p. 74)
`else`	Used with the `if` statement. (p. 74)
`elseif`	Used with the `if` statement. (p. 74)
`end`	Terminates the scope of the `for`, `if`, `switch`, and `while` statements. (p. 74, 87)
`error`	Displays the error message and terminates all flow control statements. (p. 87)
`for`	Repeat statements a specific number of times. (p. 74)
`if`	Executes statements if certain conditions are met. (p. 74)
`otherwise`	Part of the `switch` command. (p. 74)
`switch`	Selects certain statements based on the value of the `switch` expression. (p. 74)
`while`	Repeats statements as long as an expression is true. (p. 74)

Logical Functions

`all`	True if all the elements of a vector are true; operates on the columns of a matrix. (p. 77)
`any`	True if any of the elements of a vector are true; operates on the columns of a matrix. (p. 77)
`exist`	False if this name is not the name of a variable or a file. (p. 77)
`find`	The indices of a vector or matrix which are nonzero. (p. 77)
`ischar`	True if a vector or array contains character elements. (p. 77)
`isempty`	True if the matrix is empty, i.e., `[]`. (p. 77)
`isfinite`	Generates a matrix with 1 in all the elements which are finite (i.e., not `Inf` or `NaN`) and 0 otherwise. (p. 77)
`isinf`	Generates a matrix with 1 in all the elements which are `Inf` and 0 otherwise. (p. 77)
`islogical`	True for a logical variable or array. (p. 77)
`isnan`	Generates a matrix with 1 in all the elements which are `NaN` and 0 otherwise. (p. 77)
`logical`	Converts a numeric variable to a logical one. (p. 77)

Programming Language Functions

`echo`	Turns echoing of statements in m-files on and off. (p. 87)
`end`	Ends a function. Only required if the function m-file contains a nested function. (p. 74, 87)
`error`	Displays the error message and terminates the function. (p. 87)
`eval`	Executes MATLAB statements contained in a text variable. (p. 89)
`feval`	Executes a function specified by a string. (p. 89)
`function`	Begins a MATLAB function. (p. 87)
`global`	Defines a global variable (i.e., it can be shared between different functions and/or the workspace). (p. 87)
`lasterr`	If `eval` "catches" an error, it is contained here. (p. 89)
`persistent`	Defines a local variable whose value is to be saved between calls to the function. (p. 87)
`keyboard`	Stops execution in an m-file and returns control to the user for debugging purposes. (p. 82, 87)
`nargin`	Number of input arguments supplied by the user. (p. 87)
`nargout`	Number of output arguments supplied by the user. (p. 87)
`return`	Terminates the function immediately. (p. 82, 87)

Debugging Commands

`keyboard`	Turns debugging on. (p. 82, 87)
`dbstep`	Execute one or more lines. (p. 82)
`dbcont`	Continue execution. (p. 82)
`dbstop`	Set a breakpoint. (p. 82)
`dbclear`	Remove a breakpoint. (p. 82)
`dbup`	Change the workspace to the calling function or the base workspace. (p. 82)
`dbdown`	Change the workspace down to the called function. (p. 82)
`dbstack`	Display all the calling functions. (p. 82)
`dbstatus`	List all the breakpoints. (p. 82)
`dbtype`	List the current function, including the line numbers. (p. 82)
`dbquit`	Quit debugging mode and terminate the function. (p. 82)
`return`	Quit debugging mode and continue execution of the function. (p. 82, 87)

Discrete Fourier Transform

`fft`	The discrete Fourier transform. (p. 120)
`fftshift`	Switches the first half and the second half of the elements of a vector. (p. 120)
`ifft`	The inverse discrete Fourier transform. (p. 120)
`fftshift`	Unswitches the first half and the second half of the elements of a vector. (p. 120)

Sparse Matrix Functions

`speye`	Generates a Sparse identity matrix. (p. 94)
`sprand`	Sparse uniformly distributed random matrix. (p. 94, 94)
`sprandn`	Sparse normally distributed random matrix. (p. 94)
`sparse`	Generates a sparse matrix elementwise. (p. 94)
`spdiags`	Generates a sparse matrix by diagonals. (p. 94)
`full`	Converts a sparse matrix to a full matrix. (p. 94)
`find`	Finds the indices of the nonzero elements of a matrix. (p. 94)
`nnz`	Returns the number of nonzero elements in a matrix. (p. 94)
`spfun`	Applies the function to a sparse matrix. (p. 94)
`spy`	Plots the locations of the nonzero elements of a sparse matrix. (p. 94)
`spconvert`	Generates a sparse matrix given the nonzero elements and their indices. (p. 94)

Time Evolution ODE Solvers

`ode45`	Non-stiff ode solver; fourth-order, one-step method for the ode $\mathtt{y}' = \mathtt{f}(t, \mathtt{y})$. (p. 96)
`ode23`	Non-stiff ode solver; second-order, one-step method. (p. 96)
`ode113`	Non-stiff ode solver; variable-order, multi-step method. (p. 96)
`ode15s`	Stiff ode solver; variable-order, multi-step method. (p. 96)
`ode23s`	Stiff ode solver; second-order, one-step method. (p. 96)
`ode23t`	Stiff ode solver; trapezoidal method. (p. 96)
`ode23tb`	Stiff ode solver; second-order, one-step method. (p. 96)
`ode15i`	Stiff ode solver; variable-order, multi-step method for the fully implicit ode $\mathtt{f}(t, \mathtt{y}, \mathtt{y}') = \mathbf{0}$. (p. 105)
`odeset`	Assigns values to properties of the ode solver. (p. 100)

Boundary-Value Solver

`bvp4c`	Numerically solves $\mathtt{y}'(x) = \mathtt{f}(x, \mathtt{y})$ for $x \in [a, b]$ with given boundary conditions and an initial guess for `y`. (p. 109)
`bvpinit`	Calculates the initial guess either by giving `y` directly or by using a function $\mathtt{y} = \mathtt{initial\_guess\_function}(x)$. (p. 109)
`deval`	Interpolate to determine the solution desired points. (p. 109)

Numerical Operations on Functions

`dblquad`	Numerically evaluates a double integral. (p. 114)
`fminbnd`	Numerically calculates a local minimum of a one-dimensional function. (p.)
`fminsearch`	Numerically calculates a local minimum of a multi-dimensional function. (p. 114)
`optimset`	Allows you to modify the parameters used by `fzero`, `fminbnd`, and `fminsearch`. (p. 114)
`fzero`	Numerically calculates a zero of a function. (p. 114)
`quad`	Numerically evaluates an integral using Simpson's method. (p. 114)
`quadl`	Numerically evaluates an integral using the adaptive Gauss-Lobatto method. (p. 114)

Numerical Operations on Polynomials

`interp1`	Does one-dimensional interpolation. (p. 111)
`interp2`	Does two-dimensional interpolation. (p. 111)
`interp3`	Does three-dimensional interpolation. (p. 111)
`interp4`	Does n-dimensional interpolation. (p.)
`pchip`	Cubic Hermite interpolation. (p. 111)
`poly`	Calculates the coefficients of a polynomial given its roots. (p. 111)
`polyder`	Calculates the derivative of a polynomial. (p. 111)
`polyfit`	Calculates the least-squares polynomial of a given order which fits the given data. (p. 111)
`polyval`	Evaluates a polynomial at a point. (p. 111)
`polyvalm`	Evaluates a polynomial with a matrix argument. (p. 111)
`ppval`	interpolates a piecewise polynomial calculated by `pchip` or `spline`. (p. 111)
`roots`	Numerically calculates all the zeroes of a polynomial. (p. 111)
`spline`	Cubic spline interpolation. (p. 111)

Matrix Functions

`expm`	Matrix exponentiation. (p. 121)
`funm`	Evaluate general matrix function. (p. 121)
`logm`	Matrix logarithm. (p. 121)
`sqrtm`	Matrix square root. (p. 121)

Solutions To Exercises

These are the solutions to the exercises given in subsections 1.9, 2.10, and 4.5.

1.9.1a)
```
» a = 3.7; b = 5.7; deg = pi/180; ab = 79*deg;
» c = sqrt(a^2 + b^2 - 2*a*b*cos(ab))
```
answer: `7.3640`

b)
```
» format long
» c
```
answer: `7.36398828251259`

c)
```
» format short e
» asin( (b/c)*sin(ab) ) / deg
```
answer: `4.9448e+01`

d)
```
» diary 'triangle.ans'
```

1.9.2)
```
» (1.2e20 - i*12^20)^(1/3)
```
answer: `1.3637e+07 - 7.6850e+06i`

1.9.3)
```
» th = input('th ='); cos(2*th) - (2*cos(th)^2 - 1)
```

1.9.4) `help fix` or `doc fix`.

2.10.1a)
```
» A = [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]
» A = [1:4; 5:8; 9:12; 13:16]
» A = [ [1:4:13]' [2:4:14]' [3:4:15]' [4:4:16]' ]
```

b)
```
» A(2,:) = (-9/5)*A(2,:) + A(3,:)
```

2.10.2)
```
» A = 4*eye(n) - diag( ones(n-1,1), 1 ) - diag( ones(n-1,1), -1 )
```

2.10.3)
```
» A = diag([1:n].^2) - diag( ones(n-1,1), 1 ) - diag( exp([2:n]), -1 )
```

2.10.4a)
```
» A = [ ones(6,4) zeros(6) ]; A(6,1) = 5; A(1,10) = -5
```

b)
```
» A = A - tril(A,-1)
```

2.10.5)
```
» x = [0:30]'.^2     % or  x = [0:30].^2'
```

2.10.6a)
```
» R = rand(5)
```

b)
```
» [m, im] = max(R')
```

c)
```
» mean(mean(R))     % or  mean(R(:))
```

d)
```
» S = sin(R)
```

e)
```
» r = diag(R)
```

2.10.7a)
```
» A = [1 2 3; 4 5 6; 7 8 10]
» B = A^.5      % or  B = sqrtm(A)
» C = A.^.5     % or  C = sqrt(A)
```

b)
```
» A - B^2
» A - C.^2
```

4.5.1a)
```
» x = linspace(-1, +1, 100);
» y = exp(x);
» plot(x, y)
```

b)
```
» z = 1 + x + x.^2 /2 + x.^6 /6
» hold on
» plot(x, z)
```

c)
```
≫ plot(x, y-z)
```

d)
```
≫ hold off
≫ plot(x, y, 'r', x, z, 'g', x, y-z, 'm')
≫ axis equal
≫ xlabel('x')
≫ ylabel('y')
≫ title('e^i\pi = -1 is profound')
```

e)
```
≫ subplot(2, 1, 1)
≫ hold off
≫ plot(x, y, 'r', x, z, 'g')
≫ axis equal
≫ xlabel('x')
≫ ylabel('y')
≫ title('e^i\pi = -1 is profound')
≫ subplot(2, 1, 2)
≫ plot(x, y-z)
```

4.5.2a)
```
≫ x = linspace(-3, 3, 91);
≫ y = x;
≫ [X, Y] = meshgrid(x, y);     % or just do [X, Y] = meshgrid(x, x);
≫ Z = (X.^2 + 4* Y.^2) .* sin(2*pi*X) .* sin(2*pi*Y);
≫ surf(X, Y, Z);
```

b) One particular choice is
```
≫ view([1 2 5])     % or  view([63 66])
```

ASCII Table

Octal	Decimal	Control Sequence	Description
000	0	^@	Null character
001	1	^A	Start of header
002	2	^B	Start of text
003	3	^C	End of text
004	4	^D	End of transmission
005	5	^E	Enquiry
006	6	^F	Acknowledgment
007	7	^G	Bell
010	8	^H	Backspace
011	9	^I	Horizontal tab
012	10	^J	Line feed
013	11	^K	Vertical tab
014	12	^L	Form feed
015	13	^M	Carriage return
016	14	^N	Shift out
017	15	^O	Shift in
020	16	^P	Data link escape
021	17	^Q	Device control 1 (often XON)
022	18	^R	Device control 2
023	19	^S	Device control 3 (often XOFF)
024	20	^T	Device control 4
025	21	^U	Negative acknowledgement
026	22	^V	Synchronous idle
027	23	^W	End of transmissions block
030	24	^X	Cancel
031	25	^Y	End of medium
032	26	^Z	Substitute
033	27	^[	Escape
034	28	^\	File separator
035	29	^]	Group separator
036	30	^^	Record separator
037	31	^_	Unit separator
040	32		Space
041	33		!
042	34		" Double quote
043	35		# Numer sign
044	36		$ Dollar sign
045	37		% Percent
046	38		& Ampersand
047	39		' Closing single quote (apostrophe)
050	40		(Left parenthesis
051	41		) Right parenthesis
052	42		* Asterisk
053	43		+ Plus sign
054	44		, Comma
055	45		- Minus sign or dash
056	46		. Dot
057	47		/ Forward slash
060	48		0
061	49		1
062	50		2
063	51		3
064	52		4
065	53		5
066	54		6
067	55		7
070	56		8
071	57		9
072	58		: Colon
073	59		; Semicolon
074	60		< Less than sign
075	61		= Equal sign
076	62		> Greather than sign
077	63		? Question mark

Octal	Decimal	Control Sequence	Description
100	64		@ AT symbol
101	65		A
102	66		B
103	67		C
104	68		D
105	69		E
106	70		F
107	71		G
110	72		H
111	73		I
112	74		J
113	75		K
114	76		L
115	77		M
116	78		N
117	79		O
120	80		P
121	81		Q
122	82		R
123	83		S
124	84		T
125	85		U
126	86		V
127	87		W
130	88		X
131	89		Y
132	90		Z
133	91		[Left bracket
134	92		\ Back slash
135	93		] Right bracket
136	94		^ Caret
137	95		_ Underscore
140	96		‘ Opening single quote
141	97		a
142	98		b
143	99		c
144	100		d
145	101		e
146	102		f
147	103		g
150	104		h
151	105		i
152	106		j
153	107		k
154	108		l
155	109		m
156	110		n
157	111		o
160	112		p
161	113		q
162	114		r
163	115		s
164	116		t
165	117		u
166	118		v
167	119		w
170	120		x
171	121		y
172	122		z
173	123		{ Left brace
174	124		| Vertical bar
175	125		} Right brace
176	126		Tilde
177	127	^?	Delete

American Standard Code for Information Interchange (ASCII) specifies a correspondence between bit patterns and character symbols. The octal and decimal representations of the bit patterns are shown along

with a description of the character symbol. The first 32 codes (numbers 0–31 decimal) as well as the last (number 127 decimal) are non-printing characters which were initially intended to control devices or provide meta-information about data streams. For example, decimal 10 ended a line on a line printer and decimal 8 backspaced one character so that the preceding character would be overstruck. The control sequence column shows the traditional key sequences for inputting these non-printing characters where the caret (^) represents the "Control" or "Ctrl" key which must be held down while the following key is depressed.

Index

Note: In this index MATLAB commands come first, followed by symbols, and only then does the index begin with "A".

Note: All words shown in typewriter font are MATLAB commands or predefined variables unless it is specifically stated that they are defined locally (i.e., in this document).

Note: If an item is a primary topic of a section, an appendix, or a subsection, this is indicated as well as the page number (in parentheses).

MATLAB functions

Symbols

A

B

C

G

H

I

J

K

L

M

N

O

S

T

U