(March 1, 2005 3:20 p.m.)

MATLAB Overview

Ed Overman
Department of Mathematics
The Ohio State University

Introduction 3
1 Scalar Calculations 5
1.1 Simple Arithmetical Operations 5
1.2 Variables 6
1.3 Round-off Errors 8
1.4 Formatting Printing 9
1.5 Common Mathematical Functions 10
1.6 Complex Numbers 11
1.7 Help! 12
1.8 Be Able To Do 14
2 Vector and Matrix Calculations 14
2.1 Generating Matrices 15
2.2 The Colon Operator 18
2.3 Manipulating Matrices 19
2.4 Simple Arithmetical Operations 22
2.5 Be Careful! 25
2.6 Common Mathematical Functions 27
2.7 Data Manipulation Commands 27
2.8 Advanced Topic: Multidimensional Arrays 29
2.9 Be Able To Do 30
3 Text Variables and Inline Functions 32
4 Graphics 34
4.1 Two-Dimensional Graphics 34
4.2 Three-Dimensional Graphics 39
4.3 Advanced Graphics Techniques 41
4.4 Be Able To Do 46
5 Solving Linear Systems of Equations 46
5.1 Square Linear Systems 47
5.2 Catastrophic Round-Off Errors 49
5.3 Overdetermined and Underdetermined Linear Systems 50
6 File Input-Output 52
7 Some Useful Linear Algebra Commands 54
8 Programming in MATLAB 60
8.1 Control Flow 60
8.2 Matrix Relational Operators and Logical Operators 63
8.3 Script Files and Function Files 66
8.4 Odds and Ends 74
8.5 Advanced Topic: Vectorizing Code 75
9 Sparse Matrices 78
10 Ordinary Differential Equations 81
10.1 Basic Commands 81
10.2 Advanced Commands 84
11 Polynomials and Polynomial Functions 91
12 Numerical Operations on Functions 93
13 Discrete Fourier Transform 96
14 Mathematical Functions Applied to Matrices 101
Appendix: Reference Tables 103
Solutions To Exercises 113
Index 115

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 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 science and engineering. And this access is available by entering only a small number of commands and operations because its basic data element is a matrix. For an overview of the capabilities of MATLAB, type

```
>> demo
```

click on "Matrices", double-click on "Basic matrix operations", and then click on "Start".

This document 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 *The Student Edition of MATLAB: User's Guide* or *MATLAB: Using MATLAB*. There are numerous documents such as this floating around universities — some are even floating around the internet. These were generally written to cover whatever topics the author felt students needed to know in their coursework or research. This document is no different; it is being used in courses covering linear algebra, mathematical modelling, and numerical analysis.

In this document MATLAB is first introduced as a calculator and then as a plotting package. Only afterwards are more technical topics discussed. We are taking 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. Since it is easy to forget some MATLAB commands or operations, at the end of each section or subsection we provide a table which provides a brief description of each of the MATLAB commands or operations covered. It all too frequently happens that we know there is a command, or sequence of commands, that does exactly what we want — if only we could just remember what it is. We also collect all these tables in the appendix and include additional cross-referencing to show what commands and operations apply to various topics. In addition, the index is designed to help in finding things that are "just on the tip of your tongue". All the MATLAB commands discussed in this document are listed at the beginning of the index, as well as alphabetically throughout the index.

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.

Summary of Contents

Section 1 of this document 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 section 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.)

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 standard linear algebra algorithms.

Section 8 discusses MATLAB as a programming language — really a “baby C”. It also discusses how to create your own commands. Since the basic data element of MATLAB is a matrix, this programming language is *very* simple to learn and to use.

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 two, but we felt that it added too much complexity at too early a point in this document. 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 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 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 12 discusses how to numerically calculate zeroes, extrema, and integrals of functions.

Section 13 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 14 discusses how to apply mathematical functions to matrices.

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
```

A long expression can be continued to a new line by typing three periods followed by the "return" (or "enter") 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 2.7).

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.

While your work is being archived, 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.

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.

`...`	Continue an expression onto the next line.
`%`	Begin a comment

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 $\mathtt{x} = 2^{1/3}$. 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.

Text 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'
```

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 put into 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 text 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 $1 + \texttt{eps} \neq 1$.
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 $1/3$ 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 $1/3$ and the approximation to $1/3$ 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` commands and then type in `1/3` to see how the result is printed out.

Format Commands

`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.

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)
```

This function is actually calculated by generating the vector $(1, 2, \ldots, n)$ and them multiplying all its elements together by `prod([1:n])`.

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

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

```
>> deg = pi/180; sin(40*deg)
```

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

```
>>  sin = 20
```
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

Function	Description	Function	Description
`abs(x)`	The absolute value of `x`.	`floor(x)`	This is the largest integer which is $\leq$ `x`.
`acos(x)`	$\arccos$ `x`.	`log(x)`	The natural log of `x`, i.e., $\log_e$`x`.
`acosh(x)`	arccosh `x`.	`log10(x)`	The common log of `x`, i.e., $\log_{10}$`x`.
`asin(x)`	$\arcsin$ `x`.	`mod(x, y)`	The modulus after division. That is, $x - n * y$ where $n =$ `floor`(y/x).
`asinh(x)`	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`.	`round(x)`	The integer which is closest to `x`.
`atan2(x, y)`	$\arctan$ `y/x` where the angle is in $(-\pi, +\pi]$.	`sign(x)`	If $x > 0$ this returns $+1$, if $x < 0$ this returns -1, and if $x = 0$ this returns 0.
`atanh(x)`	arctanh `x`.	`sin(x)`	$\sin$ `x`.
`ceil(x)`	The smallest integer which is $\geq$ `x`.	`sinh(x)`	$\sinh$ `x`.
`cos(x)`	$\cos$ `x`.	`sqrt(x)`	$\sqrt{x}$.
`cosh(x)`	$\cosh$ `x`.	`tan(x)`	$\tan$ `x`.
`exp(x)`	e^x.	`tanh(x)`	$\tanh$ `x`.
`factorial(n)`	$n!$ for n a non-negative integer.		
`fix(x)`	If $x \geq 0$ this is the largest integer which is $\leq$ `x`. If $x < 0$ this is the smallest integer which is $\geq$ `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 $\texttt{z} = \texttt{x} + i\texttt{y}$.	`conj(z)`	$\texttt{z}^* = \texttt{x} - i\texttt{y}$.
`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. 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 command 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
```

[†]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 $a^* = (2 - 3i)\sin(1 - 3i)/(3 + \sqrt{5}\, i)$.

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").

1.8. Be Able To Do

After reading this section you should be able to do the following exercises. The MATLAB statements are given on page 113.

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 $\angle ab = 79^\circ$.
(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. Vector and Matrix Calculations

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.

Notation: *We will always write matrices using capital letters and vectors using lower case letters.* This is also a good practice for you to use.

Note: 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 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 A is *symmetric* if $\mathtt{A}^{\mathrm{T}} = \mathtt{A}$.

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

Definition

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

Note: 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]'
```

Warning: $\mathtt{x}^{\mathrm{T}} \to$ `x.'` while $\mathtt{x}^{\mathrm{H}} \to$ `x'` **so that you can only calculate** $\mathtt{x}^{\mathrm{T}}$ **by** `x'` **if** x **is real**.

Aside: In fact, x should be entered as

```
>>  x = [1:6]'
```

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

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 $\mathtt{r} = \big(\sqrt{2/3}\,, \sqrt{2}\,, \sqrt{3}\,, \sqrt{6}\,, \sqrt{2/3}\,\big)^{\mathrm{T}}$ 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

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

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`.

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" . 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('seed', <seed number>)
```

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.

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`.
`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 $5{\times}6$ 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] , [1 2 3 4] )
```
extracts the elements in the first, third, and fifth rows and the second, third, fourth, and fifth columns of `E`; thus,

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

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.,

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

† 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.

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 $\underline{5\times5}$ 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 $\underline{6\times6}$ 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 $\underline{5\times5}$ 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.,

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

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.,

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

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)
```

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 $\mathtt{H} \in \mathbb{C}^{p\times q}$ into $\mathtt{K} \in \mathbb{C}^{m\times n}$ 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 rows 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)`	A column vector of the main diagonal of the matrix (i.e., non-vector) `A`.
`diag(A, k)`	A column vector of the k-th diagonal of the matrix (i.e., non-vector) `A`.
`diag(d)`	A square matrix with the main diagonal being the vector `d`.
`diag(d, k)`	A square matrix with the k-th diagonal being the vector `d`.
`triu(A)`	A matrix which is the same size as `A` and consists of the elements on and above the main diagonal of `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`. (`triu(A, 0)` is the same as `triu(A)`.)
`tril(A)` `tril(A, k)` }	The same as the command `triu` except it uses the elements on and *below* the main diagonal or the k-th diagonal.
`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 $\mathtt{A}^p = \underbrace{\mathtt{AA}\cdots\mathtt{A}}_{p \text{ times}}$.

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 $\mathtt{AB}^{-1}$. Instead, use the operation

```
>> A\b
```

to calculate the solution of the linear system $\mathtt{Ax} = \mathtt{b}$ (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 $\mathtt{Ax} = \mathtt{b}$ repeatedly where b is each column of B in turn.

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 $\texttt{A}, \texttt{B} \in \mathbb{C}^{m \times n}$, 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 $\texttt{A} + \texttt{c}$, 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 $\texttt{a}, \texttt{b} \in \mathbb{R}^n$ by $\texttt{a}^{\mathrm{T}}\texttt{b}$. It is calculated by

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

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

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

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

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,

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

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

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 `Ax` = `b` by Gaussian elimination when `A` is a square non-singular matrix.	`A./B`	Elementwise division.
`b/A`	*Does not exist.*	`B.\A`	Elementwise left division, i.e., `B.\A` is exactly the same as `A./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. 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.

2.6. 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 14.)

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.

2.7. 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

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

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

Command	Description
`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.

2.8. 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.

2.9. Be Able To Do

After reading this section you should be able to do the following exercises. The answers are given on page 113.

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 one MATLAB statement.

2. Generate the tridiagonal matrix

$$\texttt{A} = \begin{pmatrix} 4 & -1 & & & & \\ -1 & 4 & -1 & & & \mathbf{0} \\ & -1 & 4 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 4 & -1 \\ \mathbf{0} & & & & -1 & 4 \end{pmatrix} \in \mathbb{R}^{n\times n}$$

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

3. Generate the tridiagonal matrix

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

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

4. Consider the matrix

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

(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. Text Variables and Inline Functions

Text variables 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 character variable, such as

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

is simply a row vector with each character (actually its ASCII representation) 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.

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 command. (This 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 text 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.)

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 the `inline` command. If our function is

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

the MATLAB statement is

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

For example, we can define the function

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

by

```
>> f = inline('t.^5 .* exp(-2*t) .* cos(3*t)', '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 = inline('x.^a .* exp(-b*x) .* cos(c*y)', 'x', 'y', 'a', 'b', 'c')
```

or if we want g to have one vector argument, say $\mathbf{x} = (x, y, a, b, c)^{\mathrm{T}}$ by

```
>> g = inline('x(1).^x(3) .* exp(-x(4)*x(1)) .* cos(x(5)*x(2))', 'x')
```

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.

Text Variable Commands

`inline`	Creates a mathematical function.
`vectorize`	Modifies a mathematical function created by `inline` so that it can evaluate vectors or matrices.
`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.)

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)
```

---

†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 $\mathtt{n} - 1$ 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.

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)	Symbol	Line Style	Marker	Description
`r`	red (1 0 0)	`-`	solid line (default)	`+`	plus sign
`g`	green (0 1 0)	`--`	dashed line	`o`	circle
`b`	blue (0 0 1)	`:`	dotted line	`*`	asterisk
`y`	yellow (1 1 0)	`-.`	dash-dot line	`.`	point
`m`	magenta (1 0 1) (a deep purplish red)			`x`	cross
`c`	cyan (0 1 1) (greenish blue)			`s`	square
`w`	white (1 1 1)			`d`	diamond
`k`	black (0 0 0)			`^`	upward pointing triangle
				`v`	downward pointing triangle
				`>`	right pointing triangle
				`<`	left pointing triangle
				`p`	pentagram
				`h`	hexagram
				`(none)`	no marker

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.

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 ±`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`.

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

```
>> ezpolar(f)
```

plots $r = f(\theta)$ for $\theta \in (0, 2\pi)$. You can also specify the endpoints as with `ezplot`.

Yet another command which can plot a function is `ezplot`. This command has three distinct uses. First, it is "roughly" equivalent to `fplot`. This command is easier to use than `fplot` because it does not even require you to enter the endpoints. If you do not enter them, it plots the curve in the interval $[-2\pi, +2\pi]$. It does not always show quite same curve as `fplot`. For example, in

```
>> s = 'log(x)'
>> fplot(s, [-1 1])
>> ezplot(s, [-1 1])
```

`fplot` generates a spurious plot for $x \in [-1, 0)$ because it plots the real part of $\log x$ while `ezplot` only plots the function for $x \in (0, 1]$. Also, in

```
>> f = inline('x ./ (x.^2 + 0.01)', 'x')
>> fplot(f, [-2*pi +2*pi])
>> ezplot(f)
```

the vertical axis is different and `fplot` shows more of the curve.

For every version of MATLAB there are many functions which are not plotted "correctly" by `ezplot`, especially if there are asymptotes lurking around. For example,

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

the full curve for $x \in (-5, 2)$ is not shown.

Second, `ezplot` can plot $x = x(t)$ and $y = y(t)$ by

```
>> ezplot(x, y)
```

where the plot is for $t \in [0, 2\pi]$ or

```
>> ezplot(x, y, [tmin, tmax])
```

Third, `ezplot` can plot implicit functions, i.e., $f(x, y) = 0$. For example,

```
>> f = inline('(x.^2 + y.^2).^2 - (x.^2 - y.^2)', 'x', 'y')
>> ezplot(f)
```

plots the lemniscate of Bernoulli (basically an "∞" symbol). (In the above `inline` command, you do not actually need the "."'s to indicate a vector operation because `ezplot` uses `vectorize`.) To give the endpoints of the axis, enter

```
>> ezplot(f, [xmin, xmax, ymin, ymax])
```

Again, be careful when using this command because implicit functions can be REALLY NASTY and occasionally MATLAB may not get it "completely correct".

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.

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.

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).

[†]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.

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.
`ezplot('<function>')`	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>')`	Generate an "easy" polar plot of $r = f(\theta)$.
`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)`	Plots the data points `x` vs. `y` with error bars given by `e`. `errorbar(x, y, l, u)` plots error bars which need not be symmetric about `y`.

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)
```

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 $\{ (x_i, y_j) \mid i = 1, 2, \ldots, m \, , \, j = 1, 2, \ldots, n \}$ 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	Generates an “easy” plot in 3-D.
mesh(X, Y, Z)	Plots a 3-D surface using a wire mesh.
ezmesh	Generates an “easy” 3-D surface using a wire mesh.
surf(X, Y, Z)	Plots a 3-D filled-in surface.
ezsurf	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.
ezcountour	Generates an “easy” contour looking down the z axis.
contour3(X, Y, Z)	Plots a contour in 3-D.
ezcontour3	Generates an “easy” contour in 3-D.

4.3. Advanced Graphics Techniques

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. By the way, the demonstration program shows many more of the graphics capabilities of MATLAB. Enter demo and then click on “Visualization” or on “Language/Graphics”.

MATLAB can also plot a function instead of a set of points using two different commands. We are discussing these commands here in the advanced section because, as we will show, they do not always work as you would wish — although they are very easy to use. The first command we discuss is fplot, which can be executing by simply entering

```
>> fplot('<function>', <limits>)
```

You have great flexibility in choosing the function. For example, it can be a MATLAB function, such as

```
>> fplot('tan', [0 2])
```

or a string containing a valid MATLAB expression with the variable being x, such as

```
>> fplot('sin(2*x) + 2*cos(sqrt(x))', [0 10])
```

or even as

```
>> F = 'sin(2*x) + 2*cos(sqrt(x))'
>> fplot(F, [0 10])
```

In addition, you can code your own inline function or MATLAB function (as described in section 8.3) and plot it. 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.

This function generates as many data points as it considers necessary to plot the function accurately. You can also store the data points by

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

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

Warning: This command does not always generate the correct curve.

It is also 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 `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
```

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. There are many properties of the text that can be changed in the `text` command by

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

The name of the desired property is given first, and then its value. As many properties as desired can be entered. Some of the properties are shown in the table below.

Text Properties

Property	Description
`Clipping`	`on` — (default) Any portion of the text that extends outside the axes rectangle is clipped `off` — No clipping is done.
`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.

You can also use the mouse to place text inside the plot using

```
>> gtext(<string>)
```

or

```
>> gtext(<string>, '<Name 1>', <Value 1>, '<Name 2>', <Value 2>, ... )
```

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

You can also change the default properties for `xlabel`, `ylabel`, `zlabel`, and `title`. For example, to add a **large** title, enter

```
>> title('And now for something completely different', 'FontSize', 16)
```

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 frequently important for the title to include important information about the 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 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 create the text variable `t` simply to make the example easier to read. (Alright, this isn't a *great* example, but it's better than nothing.)

You can display m plots horizontally and n plots vertically in one graphics window by

```
>> subplot(m, n, p)
```

This divides the graphics window into mn 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.

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

```
>> figure
```

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. (The number of each window appears at the top.)

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>)
```
or
```
>> text(x, y, z, <string>, '<Name 1>', <Value 1>, '<Name 2>', <Value 2>, ...)
```
The new command
```
>> zlabel(<string>)
```
is needed to label the z-axis.

If the need arises, the user can obtain much finer control over the display of the plot. This is documented in *Using MATLAB Graphics* by The MathWorks, Inc.

As we discussed in the previous section, 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. The value input to this color map is the row representing the desired color. (If this value is not an integer, it is truncated to an integer. If this integer is < 1 the value 1 is used; if it is $> n$ the value n is used.) 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
```
>> help 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 through gray (RGB components [.5 .5 .5]) to white.

MATLAB can also plot a two-dimensional image (i.e., a picture) which is represented by a matrix $\mathtt{X} \in \mathbb{R}^{m \times n}$. 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>`.
`fplot('<function>', <limits>)`	Plots the specified function within the limits given. The limits can be `[xmin xmax]` or `[xmin xmax ymin ymax]`.
`image`	Plots a two-dimensional image.
`subplot(m, n, p)`	Divides the graphics window into $m{\times}n$ rectangles and selects the p-th rectangle for the current plot.

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. Its properties can be changed by text(x, y, <string>, '<Name 1>', <Value 1>, ...).
gtext(<string>)	Places the text at the point given by the mouse. Its properties can be changed by gtext(<string>, '<Name 1>', <Value 1>, ...).
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. Be Able To Do

After reading this section you should be able to do the following exercises. The answers are given on page 113.

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
$$1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!}$$
on the same graph.
(c) Also plot the difference between e^x and this cubic polynomial on the same graph.
(d) Finally, 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.

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

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

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. However, when `A` is singular there are either zero solutions or an infinite number of solutions to this equation and a different approach is needed.

In this case the appropriate MATLAB command system is `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.,

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

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

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

(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 $\mathtt{A}$ 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 $\mathtt{I} - \mathtt{C}$ 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 $\mathtt{I} - \mathtt{C}$ is singular. Clearly there is some value of `b` between 0 and 1 where MATLAB switches between believing that $\mathtt{I} - \mathtt{C}$ is non-singular and singular.[†]

Solving Linear Systems

`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))
```

[†]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`.

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

```
Warning: Matrix is close to singular or badly scaled.
Results may be inaccurate. RCOND = 2.055969e-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

```
>>  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 = 200; 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 20 times as large for this random matrix as for the Hilbert matrix — and even though there are over 7000 times as many floating point operations needed to calculate `x` by Gaussian elimination for this random matrix.

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

$$\mathtt{Ax} - \mathtt{b}$$

is smallest in Euclidean length; that is, `norm(A*x - b)` is minimized. 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 stable 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.

Solving Linear Systems	
`A\b`	When $\mathtt{Ax} = \mathtt{b}$ is an overdetermined system, i.e., $m > n$ where $\mathtt{A} \in \mathbb{C}^{m\times n}$, 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
```

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

Command	Description
`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 $\mathrm{cond}(\mathtt{A})$, 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**.

† $\mathtt{A} \in \mathbb{R}^{n\times n}$ 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 $\mathtt{x} = \mathbf{0}$.)

- 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` 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 $\mathtt{A}^{-1}$.

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)`.

<u>`det`</u>

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.)

<u>`eig`</u>

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

$$\mathtt{Av} = \lambda \mathtt{v} \, ;$$

`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

[†] 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

```
>>  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))`.

<u>`eigs`</u>

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*

$$\mathtt{Ax} = \lambda \mathtt{Bx}$$

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)
```

(If you know C or C++, `disp` is a variable whose value has been set to 0 in the structure `op`.)

<u>`inv`</u>

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 $\mathtt{AA}^{-1} = \mathtt{A}^{-1}\mathtt{A} = \mathtt{I}$, 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.

<u>lu</u>

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`.

<u>norm</u>

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

$$\|\mathtt{x}\|_p = \begin{cases} \left(\displaystyle\sum_{i=1}^{n} |x_i|^p\right)^{1/p} & \text{if } p \in [1,\infty) \\ \displaystyle\max_{1\le i\le n} |x_i| & \text{if } p = \infty\,. \end{cases}$$

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

$$\|\mathtt{A}\|_p = \max_{\mathtt{x}\neq\mathbf{0}} \frac{\|\mathtt{Ax}\|_p}{\|\mathtt{x}\|_p} \qquad \text{for } p \in [1,\infty]$$

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

† 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`.

```
>> norm(A, 'fro')
```

null

Let $A \in \mathbb{C}^{n\times n}$. We can calculate an orthonormal basis for the null space of A by

```
>> null(A)
```

orth

Let $A \in \mathbb{C}^{n\times n}$. We can calculate an orthonormal basis for the columns of A by

```
>> orth(A)
```

qr

Let $A \in \mathbb{R}^{m\times n}$. Then there exists an orthogonal matrix $Q \in \mathbb{R}^{m\times m}$ † and an upper triangular matrix $R \in \mathbb{R}^{m\times n}$ such that

$$A = QR\,.$$

(If $A \in \mathbb{C}^{m\times n}$ then there exists an unitary matrix $Q \in \mathbb{C}^{m\times m}$ and an upper triangular matrix $R \in \mathbb{C}^{m\times n}$ such that $A = QR$.) 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

$$AE = QR$$

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 A^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 A^T. The command orth is preferable if all you want is an orthonormal basis for $R(A)$.

rank

Let $A \in \mathbb{C}^{m\times n}$. 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 $A \in \mathbb{R}^{m\times n}$. A can be decomposed into

$$A = U\Sigma V^T$$

where $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$ are orthogonal matrices and $\Sigma \in \mathbb{R}^{m\times n}$ 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 $A \in \mathbb{C}^{m\times n}$ then $U \in \mathbb{C}^{m\times m}$ and $V \in \mathbb{C}^{n\times n}$ are unitary matrices and $\Sigma \in \mathbb{R}^{m\times n}$ 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 $A^TA \in \mathbb{R}^{n\times n}$ and $AA^T \in \mathbb{R}^{m\times m}$ are square symmetric matrices. (If A is complex, A^HA and

† $Q \in \mathbb{R}^{m\times m}$ is *orthogonal* if $Q^{-1} = Q^T$. ($Q \in \mathbb{C}^{m\times m}$ is *unitary* if $Q^{-1} = Q^H$.)

AA^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 A^TA are the columns of V and those of AA^T are the columns of U. (If A is complex, the eigenvectors of A^HA are the columns of V and those of AA^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

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

has rank 2, use the `svd` command to find that the singular values of A are 25.4368, 1.7226, and 7.1857×10^{-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

$$1.8\times10^{0},\ 4.3\times10^{-1},\ 5.7\times10^{-2},\ 5.6\times10^{-3},\ 4.3\times10^{-4},\ 2.7\times10^{-5},\ 1.3\times10^{-6},\ 5.5\times10^{-8},$$
$$1.8\times10^{-9},\ 4.7\times10^{-11},\ 9.3\times10^{-13},\ 1.4\times10^{-14},\ 1.5\times10^{-16},\ 9.7\times10^{-18},\ \text{and}\ 8.1\times10^{-18}$$

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.

[†] 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 A^TA. The m eigenvalues of AA^T consist of the squares of these n singular values and $m-n$ additional zero eigenvalues.

[§] 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. Control Flow

MATLAB has four control flow and/or branching instructions: `for` loops, `while` loops, `if-else` branching tests, and `switch` branching tests.

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.) 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!*

`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
```

† `<expression>` can be a matrix in which case each column vector is stored one at a time in `i`.

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 evaluated 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.

[†]Compare the results of

```
≫  'Yes'== 'yes'
```

and

```
≫  'Yes'== 'no'
```

When a logical expression such as

```
>> i == 5
```

is evaluated, the result is either "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).

Logical Operators			
`A & B`	AND.	`~A`	NOT.
`A | B`	OR.	`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. This is not normally needed. However, as in C the `break` command does exactly this: it terminates the execution of the innermost `for` loop or `while` loop.

The `continue` statement is related to `break`, but is used even less frequently. 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 scope of 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.

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. 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, simple enter

```
>> sum( sum( F > 0.5 ) )
```

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 `x` $= (0, 4, 0, 1, -1, 0, \pi)^{\mathrm{T}}$ then the resulting vector is $(2, 4, 5, 7)^{\mathrm{T}}$. Suppose we want to add 10 to every nonzero element of `x`. Simply enter

```
>> 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. `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.

Note: 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`.

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 A^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. Script Files and Function Files

Up until now we have always entered MATLAB statements directly into the text window so that they are executed immediately. There are two difficulties with this approach when using flow control commands. First, MATLAB does not execute the statements until it encounters the final end command. Any typographical error will invalidate the entire sequence of statements and so require retyping them all over

again (unless all the statements are typed on one line, since editing can be done later on a single line). Second, this sequence of statements must be retyped every time it is needed.

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 file*: when MATLAB executes the command `<file_name>` the contents of the file "`<file_name>.m`" are executed just as if you had typed them into into the text window.

Function 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`".

Since the results of these statements can "zip by" on the computer screen, there is often a need to slow down the output. The `pause` command stops the M-file until some key is pressed. This can be particularly useful when graphics commands are executed because we can look at each plot before it is overwritten by the next one.

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>`, etc. 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.

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.

[†]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.

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 immediately follow the first line are special: they appear on the screen when you type

```
>> help <function name>
```

Before discussing functions in *great* detail, there is one 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>` is a built-in function (i.e., coded in C).
(2) It checks if `<function>` is a function (the primary function or a subfunction) in the current file.
(3) It checks if the file `<function>.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>.m` exists in this subdirectory.
(5) It searches the directories in the *search path* for the file `<function>.m`.

Note from item three 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(50000, 1)
```

runs over 200 times as fast as

```
>> prealloc(50000, 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 $\mathtt{A} \in \mathbb{C}^{n\times n}$ are desired, enter

```
>> eigval = eig(A)
```

However, if both the eigenvalues and eigenvectors are desired, enter

```
>> [V, D] = eig(A)
```

where $\mathtt{D} \in \mathbb{C}^{n\times n}$ is a diagonal matrix containing the n eigenvalues on its diagonal and the corresponding eigenvectors are found in the same columns of the matrix $\mathtt{V} \in \mathbb{C}^{n\times n}$.

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.

The `echo` command is useful for debugging script and function files. Typing

```
>> echo on
```

turns on the echoing of statements in all script files, and `echo off` turns it back off. However, this does not affect function files. To turn echoing on for a particular function, type

```
>> echo <function> on
```

To turn echoing on for all functions, type

```
>> echo on all
```

Using `echo` you can easily determine if the control flow instructions are correct in an M-file. In addition, by removing a semicolon following a statement you can see exactly what is being computed by the statement. Normally, the result will "fly by" on the terminal, but following the statement with a `pause` will stop the display.

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 them variables. The only way you will recognize this is not a "standard" MATLAB session is that the prompt is

```
K>>
```

for `K`eyboard. 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
```

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 one or more lines.
`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.

The arguments in a MATLAB function are somewhat different than in any other programming language. 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.)

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, but sometimes you have no alternative, as in the function `gravity` which can be found at the end of subsection 10.2.

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> ...
```

Warning: Spaces, not commas, must separate the variables.

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.

Also, in a MATLAB function there is occasionally a need to save the value of a local variable. Normally, local variables come into existence when the function is called and disappear when the function is completed. Once in a while, it is very convenient to be able to "save" the value of a local variable between calls to the function. (It is possible to make the variable global or pass it as a parameter which is modified by the function, but these are not recommended.) 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`.

The final point concerns an important element of programming style in any computer language. It frequently happens that programs and/or functions grow large enough to be unwieldy and inefficient. 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. Any remaining functions are called *subfunctions.* (At the end of subsection 10.2 we code the function `gravity` using one function and then code the function `gravity2` using a number of subfunctions. You can compare the readability of these two functions.)

Note: The primary function or a subfunction begins with the function definition line (i.e., the line which begins with the keyword `function`). Since there is no "endfunction" statement in MATLAB, this also ends the previous function in the file.

Subfunctions 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` command can only access the primary file.

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

Command	Description
`function`	Begins a MATLAB function.
`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.

†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.

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)
```

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.)

Another, much more esoteric, application for this command is that a MATLAB function can create or modify statements during execution. For example, since text variables can be constructed piece by piece, it is possible for a (quite simple) MATLAB function to create almost any imaginable inline function.

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 alternative ways to pass the function in the argument list. The first way is to create a *function handle*, which is the name of the function immediately preceded by the character '`@`'. For example, `'zeros'` is replaced by `@zeros` (without quotes surrounding it) so you can enter

```
>> A = feval(@zeros, 5, 6)
```

The advantage of a function handle is that it contains more information about the function than just simply its name, and so the evaluation can be done faster.

The second way to pass the function is to use an inline function. For example, another way to obtain `zeros(5,6)` is

```
>> silly_function = inline('zeros(x,6)', 'x')
>> A = feval(silly_function, 5)
```

Note that the function `silly_function` is now being passed directly, i.e., without turning it into either a character string or a function handle.

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 68. 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 much 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 control flow instructions with vector operations. We have already discussed this topic at length in subsection 8.2. Here we will focus on some advanced techniques.

†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.

As a simple example of method (1), consider the function `hilb` on page 68. Although the Hilbert matrix can be easily generated by that code, no flow control statements appear in the actual MATLAB function `hilb`:[†]

```
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);
```

You can see this by entering

```
>> type hilb
```

Although this code requires two additional matrices, it is nearly 20 times as fast. However, unless you are a MATLAB "expert" and n is in the hundreds or thousands, use the `for` loops.

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 control flow statements:

[†]This code was written before the command `repmat` was added to MATLAB. Now we can easily generate `J` by `J = repmat([1:n], n, 1)`.

```
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)`.
(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 40. And if `y` has 10^7 elements then the slow code takes a couple of minutes and the fast code a couple of seconds on a "reasonably fast" PC.

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

$$\mathtt{S} = \begin{pmatrix} n & -2 & & & & \\ & n-1 & -4 & & & \mathbf{0} \\ & & n-2 & -6 & & \\ & & & \ddots & \ddots & \\ & & & & 2 & -2n+2 \\ \mathbf{0} & & & & & 1 \end{pmatrix}$$

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>', 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. 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

$$\frac{d}{dt}\mathtt{y} = \mathtt{f}(t, \mathtt{y}) \qquad \text{for } \ t \geq t_0$$

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. For more details, consult *MATLAB — The Language of Technical Computing: Using MATLAB* by The MathWorks, Inc. There is an entire chapter on this topic.

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 $y_1 = y$, $y_2 = y_1'$, $y_3 = y_2'$, $\ldots$, $y_n = y_{n-1}'$.

Then we create a MATLAB function file which calculates the right hand side of this ode system.

```
function deriv = duffing1(t, y)
% duffing1: Duffing's equation, first try
% y'' + alpha*y' - y*(1 - beta*y^2) = Gamma*cos(omega*t)
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) ];
```

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. (In the advanced subsection we will show how to pass the parameters to the M-file in the argument list to the ode solver.)

Warning: Recall that no spaces are allowed in an element of a matrix and so in the statement

```
deriv = [ y(2) ; ... ]
```

there cannot be any spaces in entering the second row (except between parentheses).

To obtain a numerical solution to an ode system, enter

```
>>  [t, Y] = <ode solver>('<function name>', tspan, y0)
```

First, we have to choose which of the ode solvers shown in the table below to use. It is possible for MATLAB itself to decide which numerical method to use. However, there are good reasons (which we will discuss shortly) why the decision should be left in the hand of the user.

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 file 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 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.

The output parameters 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.

Note: If you do not include a left-hand side, then `t` and `Y` cannot be returned. Since MATLAB assumes that it should output the numerical trajectory somehow, it plots the solution as $y_1, y_2, \ldots, y_n$ vs. t. (For more control over the plot, see `OutputFcn` and `OutputSel` in the table "ODE Solver Parameters" in the advanced section.)

[†]The `t` in `[t, Y]` is unrelated to the `t` argument in the function `duffing1`.

[‡]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. It might be helpful to write the function as

```
function deriv = duffing1(tnow, ynow)
.....
deriv = [ ynow(2) ; -ynow(1)*(1-beta*y(1)^2)-alpha*ynow(2)+Gamma*cos(omega*tnow) ];
```

to indicate that `ynow` is `y` at the present time, i.e., `tnow`.

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.

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

$$e_i \le \max\{\, \texttt{RelTol} \cdot |y_i|, \texttt{AbsTol(i)}\,\}$$

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.)

It is up to *you* to decide which ode solver to use. As a general rule, unless you believe that the ode is stiff[†] try `ode45` or `ode113`. For a given level of accuracy, these methods should run "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`. The difficulty with a stiff solver is that you may well have to supply the Jacobian of the ode yourself. The Jacobian of $\texttt{f}(t, \texttt{y})$ is the $n{\times}n$ matrix

$$\texttt{J}(t, \texttt{y}) = \left(\frac{\partial f_i}{\partial y_j}(t, \texttt{y}) \right) ,$$

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. If you discover such an ode, check the reference book; it discusses and contrasts each of the methods in detail.

[†]There is no precise definition of when an ode is *stiff*. Instead, we say it is stiff if the time step required to obtain a stably and accurate solution is "unreasonably" small. For example, the solution to the ode

$$y'' + \left(\frac{1}{\epsilon} - 1\right) y' - \frac{1}{\epsilon} y = 0 \qquad \text{for } t \in [0, 1]$$

with boundary conditions $y(0) = 0$ and $y(1) = 1$ is

$$y(t) = \frac{e^t - e^{-t/\epsilon}}{e - e^{-1/\epsilon}} .$$

If $\epsilon \ll 1$ the trajectory quickly approaches $e^t/(e - e^{-1/\epsilon})$ for $t \gtrapprox \epsilon$ because $e^{-t/\epsilon}$ quickly becomes negligible compared to e^t. We would like to choose the time step based on the e^t term so that we want the time step $\Delta t \ll 1$. However, unless we use a stiff ode solver, we will have to choose $\Delta t \ll \epsilon$ because of numerical instabilities that arise in non-stiff ode solvers. (In chemical reaction models, it is not uncommon for ϵ to be 10^{-5} or even 10^{-10}.) A stiff solver allows us to use a "reasonable" Δt but at a cost: the Jacobian of `f` (which we will discuss shortly) must be calculated repeatedly and a linear system of equations must be solved repeatedly.

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, enter

```
>> [t, Y] = ode45('duffing1', [0 100], [2 1]);
>> figure(1)
>> plot(t, Y)
>> figure(2)
>> plot(Y(:,1), Y(:,2))
```

This results in a plot of y and y' vs. t [†] and a separate plot of y' vs. y. If we only want to plot the trajectory, enter

```
>> ode45('duffing1', [0 100], [2 1])
```

(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.)

Note: The function `duffing1` can also be passed as a function handle, i.e., as `@duffing1`. Although rarely used, the right hand side of the ode can also be created as an inline function.

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('<Name 1>', <Value 1>, '<Name 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 name>', 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
```

[†] The only difficulty is remembering which curve is y and which is y'. The ordering of the colors is: blue, green, red, blue-green, purple.

ODE Solver Parameters

`odeset('<Name 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`	The times and locations of certain events are calculated (value: `'on'`) or they are not (value: `'off'`). (Default value: `'off'`.)
`Jacobian`	Whether the analytical Jacobian is contained in the function file which calculates `f` (value: `'on'`) or not (value: `'off'`). (Default value: `'off'`.)
`JPattern`	The numerically calculated Jacobian is sparse (value: `'on'`) or it is full (value: `'off'`). (Default value: `'off'`.)
`OutputFcn`	How the output, i.e., `[t Y]`, should be handled. For example, a plot of the trajectory can be generated automatically as it is being calculated. The value of this parameter is the name of a user-defined or a MATLAB function. 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 returned (value: `'on'`) 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('duffing1', 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, s] = ode45('duffing1', tspan, y0, params)
```

and the vector `s` will contain:

`s(1)` – the number of successful steps.
`s(2)` – the number of failed attempts.
`s(3)` – the number of times `f`$(t,$`y`$)$ was evaluated.
`s(4)` – (if the ode is stiff) the number of times the Jacobian was evaluated.
`s(5)` – (if the ode is stiff) the number of LU decompositions calculated.
`s(6)` – (if the ode is stiff) the number of times a linear system had to be solved.

This might be useful in "optimizing" the performance of the ode solver if the command seems to be running excessively slowly.

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.

As a specific example, suppose we want to record where and when a specific solution to 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 $+0.5$. The function, which we discuss in detail below, follows.

```
>> params = odeset('RelTol', 1.e-6, 'Events', 'on')
>> [t, Y, tevent, Yevent, indexevent] = ode45('duffing2', tspan, y0, params)
```

where we have to create a new file `duffing2.m` as

```
function [out1, out2, out3] = duffing2(t, y, flag)
% duffing2: Duffing's equation with events set
% y'' + alpha*y' - y*(1 - beta*y^2) = Gamma*cos(omega*t)
if strcmp(flag, '')     % calculate f(t,y)
   alpha = 0.05;
   beta = 1.0;
   Gamma = 0.5;
   omega = 1.0;
   out1 = [ y(2) ; y(1)*(1-beta*y(1)^2)-alpha*y(2)+Gamma*cos(omega*t) ];
elseif strcmp(flag, 'events')     % set up the events
   out1 = [y(1)+0.5; y(1)-0.5];     % check whether y(1) passes through ±0.5
   out2 = [0; 0];                   % do not halt when this occurs
   out3 = [0; 0];                   % an event occurs when y(1) passes through
                                    % zero in either direction
end
```

Note that every time this function is called, either the right-hand side of the ode is calculated *or* the event is described — never both.

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 the same function which calculates the right-hand side. (It would be possible to write a separate function to describe the event or events. However, it is simpler to keep everything in one place, even though this makes the function somewhat more complicated to code and read.) Thus, the function `duffing2` has to serve two purposes: sometimes it calculates the right-hand side of the ode, and other times it describes what an event is. The input argument `flag` is a text variable which determines the purpose of the function: if `flag` is empty, the function returns the value of the right-hand side; if `flag` consists of the word `events`, then the event is described.

Note that there are three output argument for the function `duffing2.m`. If `flag` is empty, only the first argument is used; it is used exactly as in the function `duffing1` above. If `flag` is the string `events`, three vector arguments are output:

- `out1` – 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 will be quite involved.
- `out2` – A vector determining whether the ode solver should terminate when this particular event occurs: 1 means yes and 0 means no.
- `out3` – A vector determining how the values in `out1` 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]
```

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 `out1` passed through zero.

Since the function `duffing2` might appear confusing, we will discuss how an event is actually calculated. At the initial time, t and `y` are known, say $t^{(0)}$ and $\mathtt{y}^{(0)}$. `duffing2` is called with `flag` set to `events` so that the vector

$$\mathtt{e}^{(0)} = \begin{pmatrix} y_1^{(0)} + 0.5 \\ y_1^{(0)} - 0.5 \end{pmatrix},$$

i.e., `out1`, is generated, and so is `out2` and `out3`. Next, `duffing2` is called with `flag` empty and the solution $\mathtt{y}^{(1)}$ is calculated at time $t^{(1)}$. `duffing2` is called again with `flag` set to `events` and $\mathtt{e}^{(1)}$ is calculated and compared elementwise to $\mathtt{e}^{(0)}$. If the values have different signs in some row, then `out3` 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 exactly zero is estimated and `duffing2` is called again with `flag` empty to calculate the solution at this time, say $t^{(z,1)}$ and $\mathtt{y}^{(z,1)}$. Again, `duffing2` is called with `flag` set to `events` and $\mathtt{e}^{(z,1)}$ is generated. It is compared to $\mathtt{e}^{(0)}$ as before and the time at which the element is exactly zero is estimated again, say $t^{(z,2)}$. Again, $\mathtt{y}^{(z,2)}$ is calculated and then $\mathtt{e}^{(z,2)}$ is generated. This procedure continues until the zero is found to the desired accuracy. The calls to `duffing2` always follow this pattern: the function is called to calculate the solution `y` at the time t †; then it is called once to generate `e`, and also `out2` and `out3`.

We want to discuss a few more parameters which are passed to the ode solver. However, there is an important subject that has been left hanging long enough: it is very inconvenient that the parameters in Duffing's equation are determined in the function. 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 ode solver itself as

```
[t, Y] = ode45('duffing3', tspan, y0, [], alpha, beta, Gamma, omega);
```

where the function file `duffing3.m` is

```
function deriv = duffing3(t, y, flag, al, be, Ga, om)
% duffing3: Duffing's equation, with coefficients passed through arguments
% y'' + alpha*y' - y*(1 - beta*y^2) = Gamma*cos(omega*t)
deriv = [ y(2) ; y(1)*(1-be*y(1)^2)-al*y(2)+Ga*cos(om*t) ];
```

where we have changed the parameter names in `duffing3.m` just to show that there is does not need to be any relationship between the names in the calling statement and the names in the function file. Note that we are not passing any parameters to the ode solver, just to the function `duffing3`. We could have added this flexibility to the function `duffing2` rather than returning to `duffing1`, but we feel it is better to keep the functions as simple as possible. However, we waited until now to discuss this topic because we have to leave space for the input argument `params` and `flag`, which we have now discussed. That is, if there are four or more arguments passed to the ode solver, the fourth argument must be `param` (or whatever you want to call it). The fifth, and any following arguments, are passed directly to the function. Similarly, if the function has three or more input arguments, the third must be `flag` (or whatever you want to call it). The fourth, and any following arguments, are passed directly from the ode solver. Whether or not any parameters are passed to the ode solver, arguments must be set aside for them. In the present example, the fourth argument to `ode45` is empty, i.e., `[]`, but it has to be there. Also, the third argument to `duffing3` is unused in the function, but it has to be there.

To see a sampling of the different type of behavior in Duffing's equation, enter

```
>> [t, Y] = ode45('duffing3', [0 200], [0 1], [], 0.15, 1, 0.3, 1) plot(t, Y(:,1))
```

† For some of the ode solvers the function need only be called once. However, for others, such as `ode45` the function has to be called a number of times.

so that $\alpha = 0.15$, $\beta = 1$, $\Gamma = 0.3$ and $\omega = 1$. The initial condition is $\mathtt{y}(0) = (0, 1)^{\mathrm{T}}$. After a short time, the solution looks completely regular: 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 $\mathtt{y} = (1, 0)^{\mathrm{T}}$ by

```
≫  [t, Y] = ode45('duffing3', [0 200], [1 0], [], 0.15, 1, 0.3, 1) plot(t, Y(:,1))
```

the behavior appears to be chaotic. If we increase the time we plot by

```
≫  [t, Y] = ode45('duffing3', [0 1000], [1 0], [], 0.15, 1, 0.3, 1) plot(t, Y(:,1))
```

the behavior appears to still be completely irregular. Here is an example of a ode which has periodic motion for one initial condition and is chaotic for another! If we change α from 0.15 to 0.22 by

```
≫  [t, Y] = ode45('duffing3', [0 200], [0 0.5], [], 0.22, 1, 0.3, 1) plot(t, Y(:,1))
```

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.

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 is coded as

```
function deriv = vdp1(t, y, flag, mu)
% vdp1: van der Pol's equation
% y'' - mu*(1 - y^2)*y'  + y = 0
deriv = [ y(2) ; mu*(1-y(1)^2)*y(2)-y(1) ];
```

and is called by

```
≫  [t, Y] = <ode solver>('vdp1', tspan, y0, [], mu)
```

This is not stiff unless μ is "large".†

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 in `vdp2` by

```
function out = vdp2(t, y, flag, mu)
% vdp2: van der Pol's equation
% y'' - mu*(1 - y^2)*y'  + y = 0
if strcmp(flag, '')
   out = [ y(2) ; mu*(1-y(1)^2)*y(2)-y(1) ];
elseif strcmp(flag, 'jacobian')
   out = [ 0 1; -2*mu*y(1)*y(2)-1 mu*(1-y(1)^2) ];
elseif strcmp(flag, 'jpattern')
   out = sparse([1 2 2], [2 1 2], [1 1 1], 2, 2);
end
```

This function can be called by

```
≫  params = odeset('Jacobian', 'on');
≫  [t, Y] = ode15s('vdp2', [0 1000], [1 0], params, 100)
```

in which case the Jacobian is calculated numerically in `vdp2`.

---

†For example, using $\mu = 1$ 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 $\mu = 1000$ (but you might want to reduce the final time to 10 or even 1 if the time for the calculation seems excessive). Then use `ode15s` and see the difference in the time required.

Warning: Plotting the trajectory by

```
>> plot(t, Y)
```

is not very instructive. Instead, use

```
>> figure(1)
>> plot(t, Y(:,1))
>> figure(2)
>> plot(t, Y(:,2))
```

If we use

```
>> params = odeset('JPattern', 'on');
>> [t, Y] = ode15s('vdp2', [0 1000], [1 0], params, 100)
```

then the Jacobian is approximated numerically, but only for the elements $\mathcal{J}_{1,2}$, $\mathcal{J}_{2,1}$, and $\mathcal{J}_{2,2}$. In `vdp2` if `flag` is empty, then $\mathtt{f}(t, \mathtt{y})$ is calculated and returned in the output variable `out` as before; if `flag` contains the string `'jacobian'`, then the Jacobian matrix is returned; and if `flag` contains the string `'jpattern'` then a sparse matrix is returned with 1 in all the elements where the Jacobian has non-zero elements.

There are occasions when it is inconvenient to continually input `tspan`, `y0`, and/or `params`. This frequently happens when we want to solve a particular ode repeatedly for different values of some parameter. In this case we would like the function itself to define as many of the parameters as possible so that we do not have to continually enter them ourselves.

For example, 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 $g = 9.8\,\text{meters/second}$ 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 two "events" in the ball's flight: the highest point of the trajectory of the ball and the distance it travels.

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 function which does all of this is the following.

```
function [out1, out2, out3] = gravity(t, y, flag, speed, angle)
% gravity: the trajectory of a ball thrown from (0,0) with initial
%          speed and angle (in degrees) given
g = 9.8;
if strcmp(flag, 'init')     % set initial parameters
   out1 = [0 10*speed/g];
   out2 = [ 0 ; speed*cos(angle*pi/180) ; 0 ; speed*sin(angle*pi/180) ];
   out3 = odeset('RelTol', 1.e-6, ...
            'Events', 'on', ...
            'Refine', 20, ...
            'OutputFcn', 'odephas2', ...
            'OutputSel', [1 3]);
elseif strcmp(flag, '')     % calculate f(t,y)
   out1 = [ y(2) ; 0 ; y(4) ; -g ];
elseif strcmp(flag, 'events')     % set up the events
   out1 = [y(3) y(4)];      % check whether y(3) or y(4) pass through zero
   out2 = [1 0];            % halt only when y(3) passes through 0
   out3 = [-1 0];           % an event occurs when y(3) decreases through zero or
                            % y(4) passes through zero in either direction
end
```

(At the end of this subsection we rewrite this function using subfunctions for easier readability.) When `flag = 'init'` the function returns the initial and final time as a vector in `out1`, the initial condition as a column vector in `out2`, and the ode solver parameters in `out3`. In `odeset` the relative error is set to 10^{-6}. Next, the location of certain events is to be calculated. Third, many more data points are calculated on the numerical trajectory than needed so that the solution looks smooth. Also, the output is to be plotted using the graphics function `odephas2` which calculates the phase plane. Finally, the first and third components of `y`, i.e., x and z, are to be plotted (rather than the first and second, which is the default). (We have included each parameter on a separate line simply for readability.)

When `flag = ''` the function returns the right-hand side of the ode. Finally, when `flag = 'events'` the function sets up the event handler. The variables which are to be watched to determine when they pass through 0 are returned as a vector in `out1`. `out2` contains the vector which determines if the program is to stop when an event occurs. `out3` contains the vector which specifies in which direction the variables are to pass through 0 in order for an event to occur.

The calling sequence for solving this ode system is

```
>>  [t, Y, tevent, Yevent, index_event] = ode45('gravity', [], [], [], 10, 45)
```

If any of the parameters `tspan`, `y0`, or `params` are empty, the function is called with `flag = 'init'`. On output from this function, the first output argument contains `tspan`, the second contains `y0`, and the third contains `params`. These output arguments are used if the corresponding argument in the calling statement is empty (i.e., `[]`).

Note: Even if only one of the parameters is to be obtained from the function, all three of the output arguments, i.e., `out1`, `out2`, and `out3`, must be defined (if only as empty matrices).

Note: The number and types of the output arguments depend on the contents of `flag`. Be careful!

Since functions such as `gravity` can grow rather long and unwieldy, we split this function up into the primary function `gravity2` and a number of subfunctions. Consider doing this if have trouble debugging your code.

```
function [out1, out2, out3] = gravity2(t, y, flag, speed, angle)
% gravity: the trajectory of a ball thrown from (0,0) with initial
%          speed and angle (in degrees) given
g = 9.8;
if strcmp(flag, 'init')     % set initial parameters
   [out1, out2, out3] = initialize(g, speed, angle);
elseif strcmp(flag, '')     % calculate f(t,y)
   out1 = fnctn(g, y);
elseif strcmp(flag, 'events')     % set up the events
   [out1, out2, out3] = events(y);
end

function [times, y0, params] = initialize(g, speed, angle)
% initializes all the parameters and settings
times = [0 10*speed/g];
y0 = [ 0 ; speed*cos(angle*pi/180) ; 0 ; speed*sin(angle*pi/180) ];
params = odeset('RelTol', 1.e-6, ...
                'Events', 'on', ...
                'Refine', 20, ...
                'OutputFcn', 'odephas2', ...
                'OutputSel', [1 3]);

function deriv = fnctn(g, y)
% calculates the right hand side
out1 = [ y(2) ; 0 ; y(4) ; -g ];

function [values, halt, direction] = events(y)
% sets up the events
values = [y(3) y(4)];     % check whether y(3) or y(4) pass through zero
halt = [1 0];             % do not halt when this occurs
direction = [-1 0];       % an event occurs when y(1) passes through zero in
                          % either direction
```

11. 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 `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 14 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

$$\{\ (-3,-2),(-1.2,-1),(0,-0.5),(1,1),(1.8,2)\ \}$$

and we want to find the “best” fit by a straight line. Defining the data points more abstractly as $\{\ (x_i,y_i)\ \big|\ i=1,2,\ldots,n\ \}$ and the desired straight line by $y=c_1x+c_2$, 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

$$\mathtt{Vc}=\mathtt{y}$$

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).

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 $\{\ (x_i,y_i)\ \}$ 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 is
```
>> interp1(x, y, xvalues, <method>)
```
where xvalues is a vector of the values to be interpolated 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. Four 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.

'cubic': Cubic polynomials connect each pair of adjacent data points. Each polynomial is calculated by using the two nearest data points (if possible) at each end of the interval in which each desired value is contained. (The technical name for this is cubic Hermite interpolation.)

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 is specific to cubic spline interpolation. With it you can specify precisely the boundary conditions to use.

Polynomial Functions

Command	Description
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.
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.
roots(q)	Numerically calculates all the zeroes of a polynomial given the vector of the coefficients of the polynomial.
spline	Cubic spline interpolation.

12. Numerical Operations on Functions

MATLAB can also find a zero of a function by

```
>> fzero('<function>', x0)
```

or

```
>> fzero(@<function>, x0)
```

```
>> fzero(<function>, x0)
```

if <function> is an inline function. x0 is a guess as to the location of the zero. Alternately,

```
>> fzero('<function>', [xmin xmax])
```

finds a zero in the interval $x \in$ (xmin, xmax) where the signs of the function must differ in sign 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>', xstart, options, <arg 1>, <arg 2>, ... )
```

where `xstart` is either `x0` or `[xmin xmax]`, as we discussed previously. We can "tune" the zero finding algorithm by using the variable `options`, which is defined by

```
>> options = optimset('<Name 1>', <Value 1>, '<Name 2>', <Value 2>, ...)
```

Enter

```
>> help optimset
```

to see the various options. If desired, we can also pass arguments to `<function>`. The initial line of `<function>` is then

```
function y = <function>(x, <arg 1>, <arg 2>, ... )
```

For example, we can find a zero of the function $f(x) = \cos ax + bx$ by

```
>> yzero = fzero('fcos', xstart, [], a, b)
```

where the function is defined as

```
function y = fcos(x, a, b)
% fcos: f(x) = cos(a*x) + b*x
y = a*cos(x) + b*x;
```

We can find a zero of a particular function, such as $f(x) = \cos 2x + 3x$ without creating a function file by

```
>> fzero('cos(2*x) + 3*x', 0)
```

or

```
>> F = 'cos(2*x) + 3*x'
>> fzero(F, 0)
```

or

```
>> F_inline = inline('cos(2*x) + 3*x', 'x')
>> fzero(F_inline, 0)
```

MATLAB can also find a local minimum of a function of a single variable in an interval by

```
>> fmin('<function>', xmin, xmax)
```

or

```
>> fminbnd('<function>', xmin, xmax)
```

(the latter is preferred). As with `fzero`, the full argument list is

```
>> fminbnd('<function>', xmin, xmax, options, <arg 1>, <arg 2>, ... )
```

and `<function>` can be a string containing the function rather than the name of a function file.

MATLAB can also find a local minimum of a function of several variables by

```
>> fminsearch('<function>', iterate0)
```

where `iterate0` is a vector specifying where to begin searching for a local minimum. For example, if we enter

```
>> fminsearch('fnctn', [0 0]')
```

or

```
>> fminsearch(@fnctn, [0 0]')
```

on the function defined in a function M-file by

```
function y = fnctn(x)
% fnctn: y = (x1 - 1)^2 + (x2 + 2)^4
y = (x(1) - 1)^2 + (x(2) + 2)^4;
```

we obtain $(1.0000 - 2.0003)^{\mathrm{T}}$ (actually $(1.00000004979773, -2.00029751371046)^{\mathrm{T}}$). Alternately, if the function is defined inline by

```
>> fnctn = vectorize(inline('(x(1) - 1)^2 + (x(2) + 2)^4', 'x'))
```

we enter

```
>> fminsearch(fnctn, [0 0]')
```

(i.e., no quote marks and no "@"). 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

$$\mathtt{f}(\mathtt{x}) = \begin{pmatrix} x_1 + x_2 + \sin(x_1 - x_2) \\ x_1 - x_2 + 2\cos(x_1 + x_2) \end{pmatrix} .$$

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, the result of

```
>> xmin = fminsearch(@f, [0 0])
```

is $\mathtt{x}_{\mathrm{min}} = (-.1324\ldots, 1.0627\ldots)$. We are not done since we still have to calculate $g(\mathtt{x}_{\mathrm{min}})$. 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 $g(\mathtt{x}_{\mathrm{min}}) = 2.3\times10^{-17}$ we can assume that we have found a zero of `f`.

MATLAB can also calculate definite integrals by the functions `quad`, which uses the adaptive Simpson's method, or `quad8`, which uses the adaptive Newton-Cotes 8 panel method. To evaluate $\int_a^b f(x)\,dx$ by Simpson's method enter

```
>> quad('<function>', a, b)
```

The full argument list is

```
>> quad('<function>', a, b, tol, trace, <arg 1>, <arg 2>, ...)
```

where `tol` sets the relative tolerance for the convergence test and information about each iterate is printed if `trace` is non-zero. The arguments following are passed to the function file which calculates $f(\mathtt{x})$.

MATLAB can also calculate the double integral

$$\int_a^b \int_c^d f(x,y)\,dx\,dy$$

by

```
>> dblquad('<function>', a, b, c, d)
```

Numerical Operations on Functions

dblquad('<function>', a, b, c, d)	Numerically evaluates a double integral.
fmin('<function>', xmin, xmax) fminbnd('<function>', xmin, xmax)	Numerically calculates a local minimum of a one-dimensional function given the endpoints of the interval in which to search
fmins('<function>', iterate0) fminsearch('<function>', iterate0)	Numerically calculates a local minimum of a multi-dimensional function given the the initial iterate vector.
fzero('<function>', 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>', a, b)	Numerically evaluates an integral using Simpson's method.
quad8('<function>', a, b)	Numerically evaluates an integral using a Newton-Cotes method.

13. 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

$$f(t) = a_0 + \sum_{k=1}^{\infty} \left(a_k \cos \frac{2\pi kt}{T} + b_k \sin \frac{2\pi kt}{T} \right) \quad \text{for all } \ t \in [0, T]$$

where

$$\begin{aligned} a_0 &= \frac{1}{T} \int_0^T f(t)\, dt \\ \left.\begin{aligned} a_k &= \frac{2}{T} \int_0^T f(t) \cos kt\, dt \\ b_k &= \frac{2}{T} \int_0^T f(t) \sin kt\, dt \end{aligned}\right\} & \quad \text{for } \ k \in \mathbb{N}[1, \infty) \,. \end{aligned}$$

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†

$$P = \frac{1}{T} \int_0^T \bigl| f(t) \bigr|^2 dt$$

†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
$$f(t) = \sum_{k=-\infty}^{\infty} c_k e^{2\pi ikt/T} \quad \text{for all } t \in [0,T] \tag{13.1}$$

where
$$\begin{aligned} c_0 &= a_0 \\ \left.\begin{aligned} c_k &= \tfrac{1}{2}(a_k - ib_k) \\ c_{-k} &= \tfrac{1}{2}(a_k + ib_k) \end{aligned}\right\} &\quad \text{for } k > 0\ . \end{aligned} \tag{13.2}$$

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

$$f_j = \sum_{k=-M}^{M} \gamma_k e^{2\pi i j k/N} \quad \text{for } j = 0,1,2,\ldots,N-1 \tag{13.3}$$

where $f_j \equiv f(t_j)$. This linear system can be solved to obtain

$$\gamma_k = \frac{1}{N}\sum_{j=0}^{N-1} f_j e^{-2\pi i j k/N} \quad \text{for } k = -M, -M+1, \ldots, M\,. \tag{13.4}$$

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. (13.1), and are calculated by (13.2). The γ's are the coefficients in the discrete complex Fourier series, eq. (13.3), and are calculated by (13.4).

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

$$f_{\text{FS}}(t) = \sum_{k=-M}^{M} \gamma_k e^{2\pi i k t/T} \quad \text{for all } t \in [0,T]\,.$$

It is our responsibility (using our experience) to choose M large enough that $f(t) \approx f_{\text{FS}}(t)$. Given $\mathtt{f} = (f_0, f_1, f_2, \ldots, f_{N-1})^{\mathrm{T}}$, 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

$$\big(\gamma_0, \gamma_1, \ldots, \gamma_{M-1}, \gamma_M, \gamma_{-M}, \gamma_{-M+1}, \ldots, \gamma_{-2}, \gamma_{-1}\big)^{\mathrm{T}}\,.$$

The original function, represented by the vector `f`, is recovered by

```
>> f = N*ifft(fc)
```

†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.

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

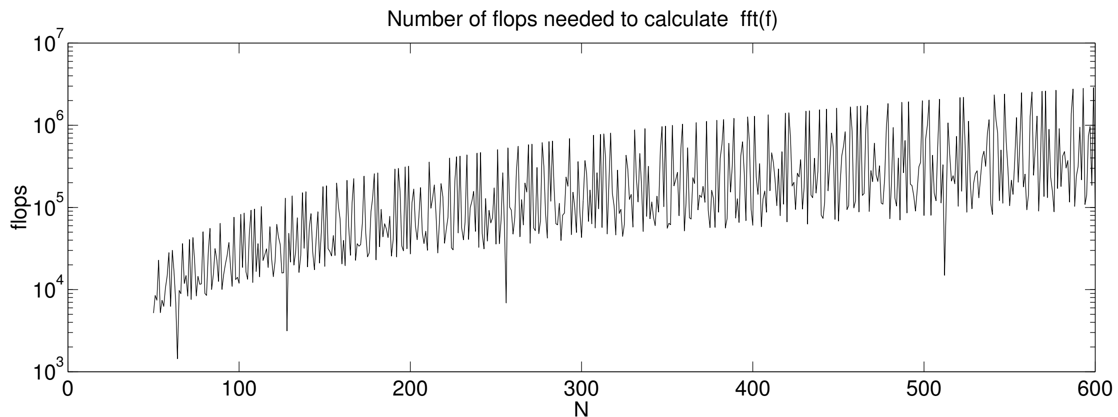
$$\gamma_0, \gamma_1, \ldots, \gamma_{M-1}, \gamma_M, \gamma_{-M}, \gamma_{-M+1}, \ldots, \gamma_{-2}, \gamma_{-1} \,,$$

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 number of flops 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 number of flops required as we vary N.† (The dips are at $N = 2^6$, 2^7, 2^8, and 2^9.)



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

$$f_{\mathrm{FS}}(t) = \sum_{k=-M}^{M-1} \gamma_k e^{2\pi ikt/T} \quad \text{for all } t \in [0, T]$$

and the discrete Fourier coefficients are calculated by

$$\gamma_k = \frac{1}{N} \sum_{j=0}^{N-1} f_j e^{-2\pi ijk/N} \quad \text{for } k = -M, -M+1, \ldots, M-2, M-1 \,.$$

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
```

†Unless N is **very large** or you have to do many runs, the CPU time used is only a tiny fraction of a second. There is often no need to manipulate the data to obtain a "good" value of N. (This was calculated using version 5 of MATLAB.)

The coefficients of the discrete Fourier transform are now contained in `fc` as

$$\texttt{fc} = \big(\gamma_0, \gamma_1, \ldots, \gamma_{M-2}, \gamma_{M-1}, \gamma_{-M}, \gamma_{-M+1}, \ldots, \gamma_{-2}, \gamma_{-1}\big)^{\mathrm{T}} .$$

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

$$\texttt{fftshift(fc)} = \big(\gamma_{-M}, \gamma_{-M+1}, \ldots, \gamma_{-2}, \gamma_{-1}, \gamma_0, \gamma_1, \ldots, \gamma_{M-2}, \gamma_{M-1},\big)^{\mathrm{T}} .$$

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`.

14. Mathematical Functions Applied to Matrices

As we briefly mentioned in subsection 2.6, 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 `A` is not a square matrix.)

If `A` is a square diagonal matrix $e^{\mathtt{A}}$ is particularly simple to calculate since

$$\mathtt{A}^p = \begin{pmatrix} a_{11} & & & & 0 \\ & a_{22} & & & \\ & & \ddots & & \\ & & & a_{n-1,n-1} & \\ 0 & & & & a_{nn} \end{pmatrix}^p = \begin{pmatrix} a_{11}^p & & & & 0 \\ & a_{22}^p & & & \\ & & \ddots & & \\ & & & a_{n-1,n-1}^p & \\ 0 & & & & a_{nn}^p \end{pmatrix} .$$

Thus,

$$e^{\mathtt{A}} = \begin{pmatrix} e^{a_{11}} & & & & 0 \\ & e^{a_{22}} & & & \\ & & \ddots & & \\ & & & e^{a_{n-1,n-1}} & \\ 0 & & & & e^{a_{nn}} \end{pmatrix} .$$

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 $\mathtt{t} = 10$ 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 $\mathtt{t} \in [0, 10]$ 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>')
```

evaluates `<function>(A)`.

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>')`	Calculates `<function>(A)` where `A` must be a square matrix.

Appendix: Reference Tables

These tables summarize the functions and operations described in this document. The number (or numbers) shown give the page number of the table where this entry is discussed.

Arithmetical Operators

`+`	Addition. (p. 6, 25)
`-`	Subtraction. (p. 6, 25)
`*`	Scalar or matrix multiplication. (p. 6, 25)
`.*`	Elementwise multiplication of matrices. (p. 25)
`/`	Scalar division. (p. 6, 25)
`./`	Elementwise division of matrices. (p. 25)
`\`	Scalar left division, i.e., $b\backslash a = a/b$. (p. 6)
`\`	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. 25, 52)
`.\`	Elementwise left division of matrices i.e., `B.\A` $=$ `A./B`. (p. 25)
`^`	Scalar or matrix exponentiation. (p. 6, 25)
`.^`	Elementwise exponentiation of matrices. (p. 25)

Special Characters

`:`	Creates a vector by `a:b` or `a:c:b`; subscripts matrices. (p. 22)
`;`	Ends a statement without printing out the result; also, ends each row when entering a matrix. (p. 8)
`,`	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. 8)
`...`	Continues a MATLAB command on the next line. (p. 6)
`%`	Begins a comment. (p. 6)

Getting Help

`demo`	Runs demonstrations of many of the capabilities of MATLAB. (p. 14, 45)
`doc`	On-line reference manual. (p. 14)
`help`	On-line help. (p. 14)
`helpdesk`	Loads the main page of the on-line reference manual. (p. 14)
`load`	Loads back all of the variables which have been saved previously. (p. 14)
`lookfor`	Searches all MATLAB commands for a keyword. (p. 14)
`save`	Saves all of your variables. (p. 14)
`type`	Displays the actual MATLAB code. (p. 14)
`who`	Lists all the current variables. (p. 14)
`whos`	Lists all the current variables in more detail than `who`. (p. 14)
`^C`	Abort the command which is currently executing (i.e., hold down the control key and type "c"). (p. 14)

Predefined Variables

`ans`	The default variable name when one has not been specified. (p. 8)
`pi`	π. (p. 8)
`eps`	Approximately the smallest positive real number on the computer such that $1 + \texttt{eps} \neq 1$. (p. 8)
`Inf`	∞ (as in $1/0$). (p. 8)
`NaN`	Not-a-Number (as in $0/0$). (p. 8)
`i`	$\sqrt{-1}$. (p. 8)
`j`	$\sqrt{-1}$. (p. 8)
`realmin`	The smallest "usable" positive real number on the computer. (p. 8)
`realmax`	The largest "usable" positive real number on the computer. (p. 8)

Format Commands

`format short`	The default setting. (p. 10)
`format long`	Results are printed to approximately the maximum number of digits of accuracy in MATLAB. (p. 10)
`format short e`	Results are printed in scientific notation. (p. 10)
`format long e`	Results are printed in scientific notation to approximately the maximum number of digits of accuracy in MATLAB. (p. 10)

Input-Output Functions

Function	Description
`csvread`	Reads data into MATLAB from the named file, one row per line of input. (p. 38)
`csvwrite`	Writes out the elements of a matrix to the named file using the same format as `csvread`. (p. 38)
`diary`	Saves your input to MATLAB and most of the output from MATLAB to disk. (p. 6)
`fopen`	Opens the file with the permission string determining how the file is to be accessed. (p. 53)
`fclose`	Closes the file. (p. 53)
`fscanf`	Behaves very similarly to the C command in reading data from a file using any desired format. (p. 53)
`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. 53, 53)
`input`	Displays the prompt on the screen and waits for you to enter whatever is desired. (p. 9)
`load`	Reads data into MATLAB from the named file, one row per line of input. (p. 38)
`print`	Prints a plot or saves it in a file using various printer specific formats. (p. 38)

Some Common Mathematical Functions

Function	Description	Function	Description
`abs`	Absolute value (p. 11, 12)	`floor`	Round downward to the nearest integer. (p. 11)
`acos`	Inverse cosine. (p. 11)	`imag`	The imaginary part of a complex number. (p. 12)
`acosh`	Inverse hyperbolic cosine. (p. 11)	`log`	The natural logarithm, i.e., to the base e. (p. 11)
`angle`	Phase angle of a complex number. (p. 12)	`log10`	The common logarithm, i.e., to the base 10. (p. 11)
`asin`	Inverse sine. (p. 11)	`mod`	The modulus after division. (p. 11)
`asinh`	Inverse hyperbolic sine. (p. 11)	`real`	The real part of a complex number. (p. 12)
`atan`	Inverse tangent. (p. 11)	`rem`	The remainder after division. (p. 11)
`atan2`	Inverse tangent using two arguments. (p. 11)	`round`	Round to the closest integer. (p. 11)
`atanh`	Inverse hyperbolic tangent. (p. 11)	`sign`	The sign of the real number. (p. 11)
`ceil`	Round upward to the nearest integer. (p. 11)	`sin`	Sine. (p. 11)
`conj`	Complex conjugation. (p. 12)	`sinh`	Hyperbolic sine. (p. 11)
`cos`	Cosine. (p. 11)	`sqrt`	Square root. (p. 11)
`cosh`	Hyperbolic cosine. (p. 11)	`tan`	Tangent. (p. 11)
`exp`	Exponential function. (p. 11)	`tanh`	Hyperbolic tangent. (p. 11)
`factorial`	Factorial function. (p. 11)		
`fix`	Round toward zero to the nearest integer. (p. 11)		

Elementary Matrices

`eye`	Generates the identity matrix. (p. 18)
`ones`	Generates a matrix with all elements being 1. (p. 18)
`rand`	Generates a matrix whose elements are uniformly distributed random numbers in the interval $(0, 1)$. (p. 18)
`randn`	Generates a matrix whose elements are normally (i.e., Gaussian) distributed random numbers with mean 0 and standard deviation 1. (p. 18)
`speye`	Generates a Sparse identity matrix. (p. 81)
`sprand`	Sparse uniformly distributed random matrix. (p. 81, 81)
`sprandsym`	Sparse uniformly distributed symmetric random matrix; the matrix can also be positive definite. (p. 81)
`sprandn`	Sparse normally distributed random matrix. (p. 81)
`zeros`	Generates a zero matrix. (p. 18)

Specialized Matrices

`hilb`	Generates the hilbert matrix. (Defined on p. 50.)
`vander`	Generates the Vandermonde matrix. (Defined on p. 92.)

Elementary Matrix Operations

`size`	The size of a matrix. (p. 18)
`length`	The number of elements in a vector. (p. 18)
`.'`	The transpose of a matrix. (p. 18)
`'`	The conjugate transpose of a matrix. (p. 18)

Manipulating Matrices

`cat`	Concatenates arrays; this is useful for putting arrays into a higher-dimensional array. (p. 30)
`clear`	Deletes a variable **or all the variables. This is a very dangerous command.** (p. 8)
`diag`	Extracts or creates diagonals of a matrix. (p. 22)
`spdiags`	Generates a sparse matrix by diagonals. (p. 81)
`repmat`	Tiles a matrix with copies of another matrix. (p. 22)
`reshape`	Reshapes the elements of a matrix. (p. 22)
`squeeze`	Removes (i.e., squeezes out) dimensions which only have one element. (p. 30)
`triu`	Extracts the upper triangular part of a matrix. (p. 22)
`tril`	Extracts the lower triangular part of a matrix. (p. 22)
`[]`	The null matrix. This is also useful for deleting elements of a vector and rows or columns of a matrix. (p. 22)

Odds and Ends

`path`	Viewing and changing the search path. (p.)
`cputime`	Approximately the CPU time (in seconds) used during this session. (p. 25)
`tic, toc`	Returns the elapsed time between these two commands. (p. 25)
`pause`	Halts execution until you press some key. (p. 73)
`rats`	Converts a floating-point number to a “close” rational number, which is frequently the exact value. (p. 52)

Two-Dimensional Graphics

`plot`	Plots the data points in Cartesian coordinates. (p. 39)
`fill`	Fills one or more polygons. (p. 45)
`semilogx`	The same as `plot` but the x axis is logarithmic. (p. 39)
`semilogy`	The same as `plot` but the y axis is logarithmic. (p. 39)
`loglog`	The same as `plot` but both axes are logarithmic. (p. 39)
`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. 39)
`polar`	Plots the data points in polar coordinates. (p. 39)
`ezpolar`	Generates an “easy” polar plot. (p. 39)
`linspace`	Generates equally-spaced points, similar to the colon operator. (p. 39)
`xlabel`	Puts a label on the x-axis. (p. 39)
`ylabel`	Puts a label on the y-axis. (p. 39)
`title`	Puts a title on the top of the plot. (p. 39)
`axis`	Controls the scaling and the appearance of the axes. (p. 39)
`hold`	Holds the current plot or release it. (p. 39)
`hist`	Plots a histogram. (p. 39)
`errorbar`	Plots a curve through data points and also the error bar at each data point. (p. 39)

Three-Dimensional Graphics

`plot3`	Plots the data points in Cartesian coordinates. (p. 41)
`ezplot3`	Generates an “easy” plot in 3-D. (p. 41)
`fill3`	Fills one or more 3D polygons. (p. 45)
`mesh`	Plots a 3-D surface using a wire mesh. (p. 41)
`ezmesh`	Generates an “easy” 3-D surface using a wire mesh. (p. 41)
`surf`	Plots a 3-D filled-in surface. (p. 41)
`ezsurf`	Generates an “easy” 3-D filled-in surface. (p. 41)
`view`	Changes the viewpoint of a 3-D surface plot. (p. 41)
`meshgrid`	Generates a 2-D grid. (p. 41)
`zlabel`	Puts a label on the z-axis. (p. 41)
`axis`	Controls the scaling and the appearance of the axes. (p. 41)
`contour`	Plots a contour looking down the z axis. (p. 41)
`ezcontour`	Generates an “easy” contour looking down the z axis. (p.)
`contour3`	Plots a contour in 3-D. (p. 41)
`ezcontour3`	Generates an “easy” contour in 3-D. (p. 41)
`colorbar`	Adds a color bar showing the correspondence between the value and the color. (p. 45)
`colormap`	Determines the current color map or choose a new one. (p. 45)

Advanced Graphics Features

clf	Clear a figure (i.e., delete everything in the figure (p. 45)
demo	Runs demonstrations of many of the capabilities of MATLAB. (p. 14, 45)
figure	Creates a new graphics window and makes it the current target. (p. 45)
fplot	Plots the specified function within the limits given. (p. 45)
gtext	Places the text at the point given by the mouse. (p. 46)
image	Plots a two-dimensional image. (p. 45)
legend	Places a legend on the plot. (p. 46)
subplot	Divides the graphics window into rectangles and moves between them. (p. 45)
text	Adds the text at a particular location. (p. 46)
ginput	Obtains the current cursor position. (p. 46)

String Functions

inline	Creates a mathematical function. (p. 33)
vectorize	Modifies a mathematical function created by inline so that it can evaluate vectors or matrices. (p. 33)
num2str	Converts a variable to a string. (p. 33)
sprintf	Behaves very similarly to the C command in writing data to a text variable using any desired format. (p. 33)
sscanf	Behaves very similarly to the C command in reading data from a text variable using any desired format. (p. 33)
str2num	Converts a string to a variable. (p. 33)
strcmp	Compares strings. (p. 61)

Data Manipulation Commands

errorbar	Plots a curve through data points and also the error bar at each data point. (p. 39)
hist	Plots a histogram of the elements of a vector. (p. 39)
max	The maximum element of a vector. (p. 29)
min	The minimum element of a vector. (p. 29)
mean	The mean, or average, of the elements of a vector. (p. 29)
norm	The norm of a vector or a matrix. (p. 29)
prod	The product of the elements of a vector. (p. 29)
sort	Sorts the elements of a vector in increasing order. (p. 29)
std	The standard deviation of the elements of a vector. (p. 29)
sum	The sum of the elements of a vector. (p. 29)

Some Useful Functions in Linear Algebra

`chol`	Calculates the Cholesky decomposition of a symmetric, positive definite matrix. (p. 59)
`cond`	Calculates the condition number of a matrix. (p. 59)
`condest`	Calculates a lower bound to the condition number of a square matrix. (p. 59)
`det`	Calculates the determinant of a square matrix. (p. 59)
`eig`	Calculates the eigenvalues, and eigenvectors if desired, of a square matrix. (p. 59)
`eigs`	Calculates some eigenvalues and eigenvectors of a square matrix. (p. 59)
`inv`	Calculates the inverse of a square invertible matrix. (p. 59)
`lu`	Calculates the LU decomposition of a square invertible matrix. (p. 59)
`norm`	Calculates the norm of a vector or matrix. (p. 59)
`null`	Calculates an orthonormal basis for the null space of a matrix. (p. 59)
`orth`	Calculates an orthonormal basis for the range of a matrix. (p. 59)
`pinv`	Calculates the pseudoinverse of a matrix. (p. 52)
`qr`	Calculates the QR decomposition of a matrix. (p. 59)
`rank`	Estimates the rank of a matrix. (p. 59)
`rref`	Calculates the reduced row echelon form of a matrix. (p. 49)
`svd`	Calculates the singular value decomposition of a matrix. (p. 59)

Logical and Relational Operators

`&`	Logical AND. (p. 62)	<	Less than. (p. 61)
`|`	Logical OR. (p. 62)	<=	Less than or equal to. (p. 61)
`~`	Logical NOT. (p. 62)	==	Equal. (p. 61)
`xor`	Logical EXCLUSIVE OR. (p. 62)	>	Greater than. (p. 61)
		>=	Greater than or equal to. (p. 61)
		~=	Not equal to. (p. 61)
		`strcmp`	Comparing strings. (p. 61)

Flow Control

`break`	Terminates execution of a `for` or `while` loop. (p. 63)
`case`	Part of the `switch` command. (p. 63)
`continue`	Begins the next iteration of a `for` or `while` loop immediately. (p. 63)
`else`	Used with the `if` statement. (p. 63)
`elseif`	Used with the `if` statement. (p. 63)
`end`	Terminates the scope of the `for`, `if`, `switch`, and `while` statements. (p. 63)
`error`	Displays the error message and terminates all flow control statements. (p. 73)
`for`	Repeat statements a specific number of times. (p. 63)
`if`	Executes statements if certain conditions are met. (p. 63)
`otherwise`	Part of the `switch` command. (p. 63)
`switch`	Selects certain statements based on the value of the `switch` expression. (p. 63)
`while`	Repeats statements as long as an expression is true. (p. 63)

Logical Functions

all	True if all the elements of a vector are true; operates on the columns of a matrix. (p. 66)
any	True if any of the elements of a vector are true; operates on the columns of a matrix. (p. 66)
exist	False if this name is not the name of a variable or a file. (p. 66)
find	The indices of a vector or matrix which are nonzero. (p. 66)
logical	Converts a numeric variable to a logical one. (p. 66)
ischar	True if a vector or array contains character elements. (p. 66)
isempty	True if the matrix is empty, i.e., []. (p. 66)
isfinite	Generates a matrix with 1 in all the elements which are finite (i.e., not Inf or NaN) and 0 otherwise. (p. 66)
isinf	Generates a matrix with 1 in all the elements which are Inf and 0 otherwise. (p. 66)
islogical	True for a logical variable or array. (p. 66)
isnan	Generates a matrix with 1 in all the elements which are NaN and 0 otherwise. (p. 66)

Programming Language Functions

echo	Turns echoing of statements in M-files on and off. (p. 73)
error	Displays the error message and terminates the function. (p. 73)
eval	Executes MATLAB statements contained in a text variable. (p. 75)
feval	Executes a function specified by a string. (p. 75)
function	Begins a MATLAB function. (p. 73)
global	Defines a global variable (i.e., it can be shared between different functions and/or the workspace). (p. 73)
lasterr	If eval "catches" an error, it is contained here. (p. 75)
persistent	Defines a local variable whose value is to be saved between calls to the function. (p. 73)
keyboard	Stops execution in an M-file and returns control to the user for debugging purposes. (p. 71, 73)
nargin	Number of input arguments supplied by the user. (p. 73)
nargout	Number of output arguments supplied by the user. (p. 73)
return	Terminates the function immediately. (p. 71, 73)

Debugging Commands

`keyboard`	Turns debugging on. (p. 71, 73)
`dbstep`	Execute one or more lines. (p. 71)
`dbcont`	Continue execution. (p. 71)
`dbstop`	Set a breakpoint. (p. 71)
`dbclear`	Remove a breakpoint. (p. 71)
`dbup`	Change the workspace to the calling function or the base workspace. (p. 71)
`dbdown`	Change the workspace down to the called function. (p. 71)
`dbstack`	Display all the calling functions. (p. 71)
`dbstatus`	List all the breakpoints. (p. 71)
`dbtype`	List the current function, including the line numbers. (p. 71)
`dbquit`	Quit debugging mode and terminate the function. (p. 71)
`return`	Quit debugging mode and continue execution of the function. (p. 71, 73)

Discrete Fourier Transform

`fft`	The discrete Fourier transform. (p. 101)
`fftshift`	Switches the first half and the second half of the elements of a vector. (p. 101)
`ifft`	The inverse discrete Fourier transform. (p. 101)

Sparse Matrix Functions

`speye`	Generates a Sparse identity matrix. (p. 81)
`sprand`	Sparse uniformly distributed random matrix. (p. 81, 81)
`sprandn`	Sparse normally distributed random matrix. (p. 81)
`sparse`	Generates a sparse matrix elementwise. (p. 81)
`spdiags`	Generates a sparse matrix by diagonals. (p. 81)
`full`	Converts a sparse matrix to a full matrix. (p. 81)
`find`	Finds the indices of the nonzero elements of a matrix. (p. 81)
`nnz`	Returns the number of nonzero elements in a matrix. (p. 81)
`spfun`	Applies the function to a sparse matrix. (p. 81)
`spy`	Plots the locations of the nonzero elements of a sparse matrix. (p. 81)
`spconvert`	Generates a sparse matrix given the nonzero elements and their indices. (p. 81)

ODE Solvers

`ode45`	Non-stiff ode solver; fourth-order, one-step method. (p. 83)
`ode23`	Non-stiff ode solver; second-order, one-step method. (p. 83)
`ode113`	Non-stiff ode solver; variable-order, multi-step method. (p. 83)
`ode15s`	Stiff ode solver; variable-order, multi-step method. (p. 83)
`ode23s`	Stiff ode solver; second-order, one-step method. (p. 83)
`ode23t`	Stiff ode solver; trapezoidal method. (p. 83)
`ode23tb`	Stiff ode solver; second-order, one-step method. (p. 83)
`odeset`	Assigns values to properties of the ode solver. (p. 85)

Numerical Operations on Functions

`dblquad`	Numerically evaluates a double integral. (p. 96)
`fmin`, `fminbnd`	Numerically calculates a local minimum of a one-dimensional function. (p. 96)
`fmins`, `fminsearch`	Numerically calculates a local minimum of a multi-dimensional function. (p. 96)
`optimset`	Allows you to modify the parameters used by `fzero`, `fminbnd`, and `fminsearch`. (p. 96)
`fzero`	Numerically calculates a zero of a function. (p. 96)
`quad`	Numerically evaluates an integral using Simpson's method. (p. 96)
`quad8`	Numerically evaluates an integral using a Newton-Cotes method. (p. 96)

Numerical Operations on Polynomials

`interp1`	Does one-dimensional interpolation. (p. 93)
`interp2`	Does two-dimensional interpolation. (p. 93)
`interp3`	Does three-dimensional interpolation. (p. 93)
`interp4`	Does n-dimensional interpolation. (p.)
`poly`	Calculates the coefficients of a polynomial given its roots. (p. 93)
`polyder`	Calculates the derivative of a polynomial. (p. 93)
`polyfit`	Calculates the least-squares polynomial of a given order which fits the given data. (p. 93)
`polyval`	Evaluates a polynomial at a point. (p. 93)
`polyvalm`	Evaluates a polynomial with a matrix argument. (p. 93)
`roots`	Numerically calculates all the zeroes of a polynomial. (p. 93)
`spline`	Cubic spline interpolation. (p. 93)

Matrix Functions

`expm`	Matrix exponentiation. (p. 102)
`funm`	Evaluate general matrix function. (p. 102)
`logm`	Matrix logarithm. (p. 102)
`sqrtm`	Matrix square root. (p. 102)

Solutions To Exercises

These are the solutions to the exercises given in subsections 1.8, 2.9, and 4.4.

1.8.1a)
```
>> 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'
```

1.8.2)
```
>> (1.2e20 - i*12^20)^(1/3)
```
answer: `1.3637e+07 - 7.6850e+06i`

1.8.3)
```
>> th = input('th ='); cos(2*th) - (2*cos(th)^2 - 1)
```

1.8.4) `help fix` or `doc fix`.

2.9.1a)
```
>> 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,:)
```

2.9.2)
```
>> A = 4*eye(n) - diag( ones(n-1,1), 1 ) - diag( ones(n-1,1), -1 )
```

2.9.3)
```
>> A = diag([1:n].^2) - diag( ones(n-1,1), 1 ) - diag( exp([2:n]), -1 )
```

2.9.4a)
```
>> A = [ ones(6,4) zeros(6) ]; A(6,1) = 5; A(1,10) = -5
```

b)
```
>> A = A - tril(A,-1)
```

2.9.5)
```
>> x = [0:30]'.^2     % or  x = [0:30].^2'
```

2.9.6a)
```
>> R = rand(5)
```

b)
```
>> [m, im] = max(R')
```

c)
```
>> mean(mean(R))     % or  mean(R(:))
```

d)
```
>> S = sin(R)
```

e)
```
>> r = diag(R)
```

2.9.7a)
```
>> 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
```

4.4.1a)
```
>> 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, x, z, x, y-z)
>> axis equal
>> xlabel('x')
>> ylabel('y')
>> title('e^i\pi = -1 is profound')
```

4.4.2a)
```
>> 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])
```

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 Commands

`abs`, 11, 12, 105
`acos`, 11, 105
`acosh`, 11, 105
`all`, 66, 110
`angle`, 12, 105
`any`, 65, 66, 110
`asin`, 11, 105
`asinh`, 11, 105
`atan`, 11, 105
`atanh`, 11, 105
`atan2`, 11, 105
`axis`, 35, 39, 41, 44, 107
`ballode`, 86
`break`, 62, 63, 109
`case`, 63, 109
  different than in C, 63
`cat`, 30, 106
`ceil`, 11, 105
`chol`, 54, 59, 109
`clear`, 7, 8, 11, 106
  danger in using, 7
`clf`, 43, 45, 108
`colorbar`, 44, 45, 107
`colormap`, 44, 45, 107
`cond`, 50, 54, 59, 109
`condest`, 54, 59, 109
`conj`, 12, 105
`continue`, 62, 63, 109
`contour`, 40, 41, 107
`contour3`, 40, 41, 107
`cos`, 11, 105
`cosh`, 11, 105
`cputime`, 24, 25, 107
`csvread`, 37, 38, 52, 105
`csvwrite`, 37, 38, 52, 105
`cumsum`, 28, 29
`dblquad`, 95, 96, 112
`dbclear`, 71, 111
`dbcont`, 71, 111
`dbdown`, 71, 111
`dbquit`, 71, 111
`dbstack`, 71, 111
`dbstatus`, 71, 111
`dbstep`, 71, 111
`dbstop`, 71, 111
`dbtype`, 71, 111
`dbup`, 71, 111
`demo`, 3, 13, 14, 34, 41, 45, 104, 108
`det`, 55, 59, 109
`diag`, 20, 22, 106
`diary`, 5, 6, 105
`diff`, 28, 29
`disp`, 7, 8, 32, 53
`doc`, 3, 13, 14, 104
`echo`, 70, 73, 110
`eig`, 25, 55, 59, 69, 109
`eigs`, 56, 59, 109
`else`, 61, 63, 109
`elseif`, 61, 63, 109
`end`, 60, 62, 63, 109
`error`, 69, 73, 109, 110
`errorbar`, 37, 39, 107, 108
`eval`, 74, 75, 110
`exist`, 66, 110
`exp`, 11, 12, 105
`expm`, 102, 112
`eye`, 17, 18, 106
`ezcontour`, 40, 41, 107
`ezcontour3`, 40, 41, 107
`ezmesh`, 40, 41, 107
`ezplot`, 36, 39, 107
`ezplot3`, 39, 41, 107
`ezpolar`, 36, 39, 107
`ezsurf`, 40, 41, 107
`factorial`, 10, 11, 105
`fclose`, 52, 53, 105
`feval`, 74, 75, 110
`fft`, 98, 101, 111
`fftshift`, 100, 101, 111
`figure`, 43, 45, 108
`fill`, 44, 45, 107
`fill3`, 44, 45, 107
`find`, 64, 65, 66, 80, 81, 110, 111
`fix`, 11, 105
`floor`, 11, 105
`fmin`, 94, 96, 112
`fminbnd`, 94, 96, 112
`fmins`, 96, 112
`fminsearch`, 94, 96, 112
`fopen`, 52, 53, 105
`for`, 60, 63, 109
`format`, 9, Subsect. 2.5 (25), 104

fplot, 41, 45, 108
fprintf, 7, 37, 52, 53, 105
fscanf, 37, 52, 53, 105
full, 79, 81, 111
function, 67, 72, 73, 110
funm, 102, 112
fzero, 93, 94, 96, 112
ginput, 42, 46, 108
global, 72, 73, 110
gtext, 42, 43, 46, 108
help, 3, 12, 14, 67, 73, 104
helpdesk, 13, 104
hilb, 25, 26, 55, 76, 106
hist, 36, 39, 107, 108
hold, 35, 39, 107
if, 60, 63, 109
ifft, 98, 101, 111
imag, 12, 105
image, 44, 45, 108
inline, 33, 108
input, 9, 105
interp1, 93, 112
interp2, 93, 112
interp3, 93, 112
interpn, 93, 112
inv, 23, 56, 59, 109
ischar, 66, 110
isempty, 66, 72, 110
isfinite, 66, 110
isinf, 66, 110
islogical, 65, 66, 110
isnan, 66, 110
keyboard, 70, 71, 73, 110, 111
lasterr, 74, 75, 110
legend, 42, 46, 108
length (number of elements in), 17, 18, 29, 65, 106
linspace, 34, 37, 39, 107
load, 13, 14, 37, 38, 44, 104, 105
  be careful, 38
log, 11, 105
logical, 65, 66, 110
loglog, 36, 39, 107
logm, 102, 112
log10, 11, 105
lookfor, 12, 14, 67, 104
lu, 57, 59, 109
max, 27, 28, 29, 108
mean, 28, 29, 108
mesh, 40, 41, 44, 107
meshgrid, 40, 41, 107
min, 29, 108
mod, 11, 105
nargin, 69, 73, 110
nargout, 69, 73, 110
nnz, 80, 81, 111
norm, 29, 57, 59, 69, 108, 109
null, 58, 59, 109
num2str, 32, 33, 53, 108
odeset, 84, 85, 90, 91, 111
ode113, 82, 111
ode15s, 82, 111
ode23, 82, 111
ode23s, 82, 111
ode23t, 82, 111
ode23tb, 82, 111
ode45, 82, 111
ones, 17, 18, 106
optimset, 94, 96, 112
orth, 58, 59, 109
otherwise, 63, 109
path, 68, 107
pause, 67, 70, 73, 107
persistent, 72, 73, 110
pinv, 51, 52, 109
plot, 34, 35, 39, 42, 92, 107
plot3, 39, 41, 107
polar, 36, 39, 107
poly, 91, 93, 112
polyder, 92, 93, 112
polyfit, 92, 93
polyval, 91, 92, 93, 112
polyvalm, 92, 93, 112
print, 38, 105
prod, 29, 108
qr, 58, 59, 109
quad, 95, 96, 112
quad8, 95, 96, 112
rand, 17, 18, 36, 50, 106
randn, 17, 18, 37, 106
rank, 58, 59, 109
rats, 52, 107
real, 12, 105
rem, 11, 105
repmat, 21, 22, 106
reshape, 20, 21, 22, 106
return, 69, 71, 73, 110, 111
roots, 91, 93, 112
round, 11, 105
rref, Sect. 5 (46), 73, 109
save, 13, 14, 104
semilogx, 36, 39, 107
semilogy, 36, 39, 107
sign, 11, 105
sin, 11, 105
sinh, 11, 105
size, 17, 18, 106
sort, 28, 29, 108
sparse, 78, 79, 81, 111
spconvert, 80, 81, 111
spdiags, 79, 81, 111
  differences from diag, 79
speye, 81, 106, 111
spfun, 81, 111
spline, 93, 112
sprand, 80, 81, 106, 111
sprandn, 80, 81, 106, 111
sprandsym, 80, 81, 106
sprintf, 32, 33, 108

spy, 81, 111
sqrt, 11, 27, 105
sqrtm, 23, 102, 112
squeeze, 30, 106
sscanf, 33, 108
std, 28, 29, 108
strcmp, 61, 108, 109
str2num, 33, 108
subplot, 43, 45, 108
sum, 29, 64, 108
surf, 40, 41, 44, 107
svd, 58, 59, 109
switch, 62, 63, 109
different than in C, 63
tan, 11, 105
tanh, 11, 105
text, 42, 43, 46, 108
tic, 24, 25, 107
title, 36, 39, 42, 107
toc, 24, 25, 107
tril, 21, 22, 106
triu, 20, 22, 106
type, 13, 14, 67, 104
vander, 92, 106
vectorize, 33, 108
view, 40, 41, 107
while, 62, 63, 109
who, 13, 14, 104
whos, 13, 14, 104
xlabel, 36, 39, 42, 43, 107
xor, 62, 64, 109
ylabel, 36, 39, 42, 43, 107
zeros, 17, 18, 106
zlabel, 41, 42, 44, 107

Symbols

+, 6, 22, 25, 103
exception to, 24
-, 6, 22, 25, 103
*, 6, 22, 25, 103
.*, 23, 25, 103
/, 6, 23, 25, 103
warning about matrix division, 23
./, 23, 25, 103
\, 6, 25, 47, 48, 51, 52, 103
.\, 23, 25
^, 5, 6, 23, 25, 103
.^, 23, 25
', 6, 16, 18, 106
.', 16, 18, 106
..., 5, 6, 103
%, 5, 6, 103
,, 6, 8, 15, 22, 103
;, 6, 8, 15, 22, 103
:, 16, Subsect. 2.2 (18), Subsect. 2.3 (19), 22, 103
<, 61, 109
<=, 61, 109
>, 61, 109
>=, 61, 109
==, 61, 109
~=, 61, 109
&, 62, 64, 109
|, 62, 64, 109
~, 62, 64, 109
!, *See* factorial
[], 21, 22, 106

A

A^H, *See* Conjugate transpose
A^T, *See* Transpose
A^+, *See* Pseudoinverse
Abort command, 12
abs, 11, 12, 105
Accuracy, 9
principle, 10
acos, 11, 105
acosh, 11, 105
all, 66, 110
AND (logical operator), 62, 64, 109
angle, 12, 105
ans, 7, 8, 104
any, 65, 66, 110
Arithmetic progression, 18
Arithmetical operations, Subsect. 1.1 (5), Subsect. 2.4 (22), 103
+, 6, 22, 25, 103
exception to, 24
-, 6, 22, 25, 103
/, 6, 23, 25, 103
warning about matrix division, 23
./, 23, 25, 103
*, 6, 22, 25, 103
.*, 23, 25, 103
\, 6, 25, 47, 48, 51, 52, 103
.\, 23, 25
^, 6, 23, 25, 103
.^, 23, 25
elementwise, 23
Array, *See* Matrix, Multidimensional array, or Vector
ASCII character representation, 32, 42
asin, 11, 105
asinh, 11, 105
atan, 11, 105
atanh, 11, 105
atan2, 11, 105
Augmented matrix form, 46–49
See also Matrix
Average value, 28
axis, 35, 39, 41, 44, 107

B

Ball, 89
ballode, 86
Binary format, 13, 38
break, 62, 63, 109

C

^C, 12, 14, 104
C (programming language), 6, 19, 33, 37, 46, 52, 53, 62, 63, 71, 72, 105, 108
C++ (programming language), 74
Calculator, Subsect. 1.1 (5)
case, 63, 109
  different than in C, 63
Case sensitive, 8
cat, 30, 106
Catching errors, 74
ceil, 11, 105
chol, 54, 59, 109
Cholesky decomposition, 54
clear, 7, 8, 11, 106
  danger in using, 7
clf, 43, 45, 108
Clown, 44
Colon operator, 16, Subsect. 2.2 (18), Subsect. 2.3 (19), 22, 103
  possible floating-point errors in, 19, 34
    See also linspace
colorbar, 44, 45, 107
Color map, 44
colormap, 44, 45, 107
Comment character, 5, 6, 103
Complex conjugate, 12
Complex numbers, 5, Subsect. 1.6 (11)
cond, 50, 54, 59, 109
condest, 54, 59, 109
Condition number, *See* Matrix
conj, 12, 105
Conjugate transpose
    See also Transpose
Continuation (of a line), 5, 6, 103
continue, 62, 63, 109
contour, 40, 41, 107
Contour plot, 40
contour3, 40, 41, 107
Control flow, *See* Programming language
cos, 11, 105
$\cos z$, 12
cosh, 11, 105
CPU, 24
cputime, 24, 25, 107
csvread, 37, 38, 52, 105
csvwrite, 37, 38, 52, 105
Cubic splines, *See* Interpolation
cumsum, 28, 29
Cursor
  entering current position, 42

D

Data
  best polynomial fit to, 92
  closing files, 52
  manipulation, Subsect. 2.7 (27), 108
Data (cont.)
  opening files, 52
  reading into MATLAB, 37, 38, 52, 80, 105
  writing from MATLAB, 37, 38, 52, 105
dblquad, 95, 96, 112
dbclear, 71, 111
dbcont, 71, 111
dbdown, 71, 111
dbquit, 71, 111
dbstack, 71, 111
dbstatus, 71, 111
dbstep, 71, 111
dbstop, 71, 111
dbtype, 71, 111
dbup, 71, 111
Debugging M-files, *See* Function files *and* Script files
demo, 3, 13, 14, 34, 41, 45, 104, 108
Demonstration program, 3, 13, 41, 44
det, 55, 59, 109
Determinant, 55
diag, 20, 22, 106
Diagonals, *See* Matrix
diary, 5, 6, 105
diff, 28, 29
Digits of accuracy, 9
disp, 7, 8, 32, 53
Display
  formatting the, Subsect. 1.4 (9)
  misinterpreting, Subsect. 2.5 (25)
  suppressing, 6, 8, 15, 22, 103
  variable, 7, 8, 53
    See also disp*See* fprintf
doc, 3, 13, 14, 104
Documentation (MATLAB), 13
Dot product, 24
Duffing's equation, *See* Ordinary differential equations
duffing1 (locally defined), 82
duffing2 (locally defined), 86, 87
duffing3 (locally defined), 87

E

e^z, 12
Earth, 44
echo, 70, 73, 110
eig, 25, 55, 59, 69, 109
Eigenvalues, 25, 54, 55, 56, 59, 69
  definition of, 55
Eigenvectors, 55, 56, 59, 69
eigs, 56, 59, 109
else, 61, 63, 109
elseif, 61, 63, 109
end, 60, 62, 63, 109
eps, 8, 9, 62, 104
    See also Machine epsilon
error, 69, 73, 109, 110
Error bars, 36, 37
errorbar, 37, 39, 107, 108
Euclidean length, *See* Length of a vector

eval, 74, 75, 110
EXCLUSIVE OR (logical operator), 62, 64, 109
exist, 66, 110
exp, 11, 12, 105
expm, 102, 112
Exponentiation, 5, 6, 23
Extrapolation, 92
    See also Interpolation
eye, 17, 18, 106
ezcontour, 40, 41, 107
ezcontour3, 40, 41, 107
ezmesh, 40, 41, 107
ezplot, 36, 39, 107
ezplot3, 39, 41, 107
ezpolar, 36, 39, 107
ezsurf, 40, 41, 107

F

factorial, 10, 11, 105
Factorial function, 10
FALSE (result of logical expression), 62
Fast Fourier transform, *See* Fourier transform
fclose, 52, 53, 105
feval, 74, 75, 110
fft, 98, 101, 111
fftshift, 100, 101, 111
figure, 43, 45, 108
fill, 44, 45, 107
fill3, 44, 45, 107
find, 64, 65, 66, 80, 81, 110, 111
Finite differences, 28
fix, 11, 105
Floating-point numbers, 8, 19
Floating-point operations, *See* Flops
floor, 11, 105
Flops (*fl*oating-point *op*eration*s*), 24
Flow control, *See* Programming language
fmin, 94, 96, 112
fminbnd, 94, 96, 112
fmins, 96, 112
fminsearch, 94, 96, 112
fopen, 52, 53, 105
for, 60, 63, 109
format, 9, Subsect. 2.5 (25), 104
Format commands, 9, 104
Fourier series, Sect. 13 (96)
  complex, 97
  real, 96
Fourier transform, Sect. 13 (96)
  discrete, Sect. 13 (96), 101, 111
  fast (FFT), 99
fplot, 41, 45, 108
fprintf, 7, 37, 52, 53, 105
  specifications, 53
Frequency, *See* Power
fscanf, 37, 52, 53, 105
  specifications, 53
full, 79, 81, 111
function, 67, 72, 73, 110
Function handle, 75
Function files, Subsect. 8.3 (66)
  commands in, 71, 73, 110, 111
  comments in, 67, 68
  conflict between function and variable name, 10
  debugging, 70
  error, 69, 73, 109, 110
  definition line, 67
  example using multiple input and output arguments, 69
  function (required word), 67, 72, 73, 110
  input and output arguments, 67, 71–72
    variable number of, 69
  names of, 67
  passing function names to, 74, 75
  primary function in, 72
  return, 69, 71, 73, 110, 111
  subfunctions in, 72
Functions (mathematical)
    See also Polynomials
  built-in, 10, 13
  common mathematical, Subsect. 1.5 (10)
  definite integrals of, 95
  "hijacked", 73
  inline, 33
    passing as an argument, 75
  local minimum of, 94
  numerical operations on, 93, 96, 112
  order in which MATLAB searches for functions, 68, 73
  primary, 72
  private, 73
  subfunctions
  zeroes of, 93, 95
funm, 102, 112
fzero, 93, 94, 96, 112

G

Gaussian elimination, 47, 50
Generalized eigenvalue problem, 56
get_intervals_fast (locally defined), 78
get_intervals_slowly (locally defined), 77
ginput, 42, 46, 108
global, 72, 73, 110
Graphics, Sect. 4 (34)
  advanced techniques, Subsect. 4.3 (41), 108
  changing endpoints, 35
  customizing lines and markers, 35
  demonstration, 34
  holding the current plot, 35
  labelling, 42–46
    text properties, 42
    using TEX commands, 43
  multiple plots, 43
  multiple windows, 43
  printing, 38, 105
  saving to a file, 38, 105

Graphics (cont.)
  two-dimensional, Subsect. 4.1 (34), 107
  three-dimensional, Subsect. 4.2 (39)
  window, 34
Gravity, 89
gravity (locally defined), 90
gravity2 (locally defined), 90, 91
gtext, 42, 43, 46, 108

H

^{H}, *See* Conjugate transpose
Handle, *See* Function handle
Helix, 39
help, 3, 12, 14, 67, 73, 104
Help facility, Subsect. 1.7 (12)
  keyword, 12
  getting help, 14, 104
helpdesk, 13, 104
Hermite polynomials, *See* Interpolation
hilb, 25, 26, 55, 76, 106
hilb_local (locally defined), 68
Hilbert matrix, 25, 26, 50, 55, 59, 68
  function file for, 68, 75
hist, 36, 39, 107, 108
Histogram, 36
hold, 35, 39, 107

I

I, *See* Identity matrix
i, 5, 8, 104
Identity matrix, 17
    See also eye
if, 60, 63, 109
ifft, 98, 101, 111
imag, 12, 105
image, 44, 45, 108
Imaginary numbers, 5, 8, 104
Inf, 8, 35, 104
Inline, 33, 108
Inline functions, *See* Functions (mathematical)
Inner product, 24
input, 9, 105
Integration, numerical, 95
Interpolation, 92, 93
  cubic, 93
  cubic splines, 93
  how to do extrapolation, 93
  linear splines, 93
interp1, 93, 112
interp2, 93, 112
interp3, 93, 112
interpn, 93, 112
inv, 23, 56, 59, 109
ischar, 66, 110
isempty, 66, 72, 110
isfinite, 66, 110
isinf, 66, 110
islogical, 65, 66, 110
isnan, 66, 110

J

j, 5, 8, 104
Java (programming language), 74

K

keyboard, 70, 71, 73, 110, 111
Keyword, 12

L

lasterr, 74, 75, 110
Left division, *See* \
legend, 42, 46, 108
Lemniscate of Bernoulli, 36
length (number of elements in), 17, 18, 29, 65, 106
Length of a vector (i.e., Euclidean length), 29
    See also norm
Linear splines, *See* Interpolation
Linear system of equations, Sect. 5 (46), Subsect. 5.3 (51)
  least-squares solution, 51, 92
  overdetermined, Subsect. 5.3 (51), 92
  solving by rref, Sect. 5 (46)
  underdetermined, Subsect. 5.3 (51)
linspace, 34, 37, 39, 107
load, 13, 14, 37, 38, 44, 104, 105
  be careful, 38
log, 11, 105
logical, 65, 66, 110
Logical expression, 60
  result of, 61
Logical functions, 66, 110
Logical operators, 62, 109
  AND (&), 62, 64, 109
  applied to matrices, Subsect. 8.2 (63)
    result of, 64
  EXCLUSIVE OR (xor), 62, 109
  NOT (~), 62, 109
  OR (|), 62, 109
loglog, 36, 39, 107
logm, 102, 112
log10, 11, 105
lookfor, 12, 14, 67, 104
lu, 57, 59, 109
LU decomposition, 57

M

Machine epsilon (eps), 8, 104
  calculation of, 62
Mathematical functions, Subsect. 1.5 (10), 12, Subsect. 2.6 (27), 105
Matrix
  augmented, 46–49

Matrix, augmented (cont.)
    is not a matrix, 47
  Cholesky decomposition, 54
  condition number, 54
    approximation to, 54
  defective, 55
  deleting rows or columns, 21
  determinant of, *See* Determinant
  diagonals of, 20, 79, 80
  elementary, 18, 106
  elementary operations, 106
  extracting submatrices, 19
  full, 78
  generating, Subsect. 2.1 (15), Subsect. 2.3 (19)
    individual elements, 16
    by submatrices, 17
  Hermitian, 16
  Hilbert, *See* Hilbert matrix
  identity, 17
  inverse of, 56
  Jacobian, 83, 88
  logical, 65
  lower triangular part of, 20, 57
    unit, 57
  LU decomposition, 57
  manipulation, Subsect. 2.3 (19), 106
  "masking" elements of, 65
  maximum value, 27, 28
  minimum value, 28
  multidimensional, Subsect. 2.8 (29)
  null, 21, 22, 106
  orthogonal, 58
  QR decomposition, 58
  positive definite, 80
  preallocation of, 68
  pseudoinverse of, 51
  replicating, 21
  reshaping, 20, 21
  singular
    warning of, 56
  singular value decomposition, 58
  sparse, Sect. 9 (78), 111
  specialized, 106
  sum of elements, 28
  SVD, *See* Singular value decomposition (above)
  symmetric, 16, 80
  tridiagonal, 55, 78
  unitary, 58
  upper triangular part of, 20
  Vandermonde, *See* Vandermonde matrix
`max`, 27, 28, 29, 108
Maximum value, 27
`mean`, 28, 29, 108
Mean value, 28
`mesh`, 40, 41, 44, 107
`meshgrid`, 40, 41, 107
M-file, 67
`min`, 29, 108
Minimum value, 28
`mod`, 11, 105
Monotonicity, test for, 28
Monty Python, 32
Moore-Penrose conditions, 51
Mouse location, *See* `ginput`
Multidimensional arrays, Subsect. 2.8 (29)

N

`NaN`, 8, 104
`nargin`, 69, 73, 110
`nargout`, 69, 73, 110
Newton-Cotes method (of numerical integration), 95
Newton's laws, 89
`nnz`, 80, 81, 111
`norm`, 29, 57, 59, 69, 108, 109
Norm
  matrix, 57
    Frobenius, 57
    p-norm, 57
  vector, 57
NOT (logical operator), 62, 64, 109
`null`, 58, 59, 109
Null matrix, 21, 22, 106
Null space, 58
`num2str`, 32, 33, 53, 108

O

Ode, *See* Ordinary differential equations
`odeset`, 84, 85, 90, 91, 111
`ode113`, 82, 111
`ode15s`, 82, 111
`ode23`, 82, 111
`ode23s`, 82, 111
`ode23t`, 82, 111
`ode23tb`, 82, 111
`ode45`, 82, 111
`ones`, 17, 18, 106
`optimset`, 94, 96, 112
OR (logical operator), 62, 64, 109
Ordinary differential equations, Sect. 10 (81)
  Duffing's equation, 81–88
  first-order system, 81
    with constant coefficients, 102
  projectile equation
  solvers, 82, 83, 111
    `ode113`, 82, 111
    `ode15s`, 82, 111
    `ode23`, 82, 111
    `ode23s`, 82, 111
    `ode23t`, 82, 111
    `ode23tb`, 82, 111
    `ode45`, 82, 111
    absolute error, 83
    adaptive step size, 83
    events, 86–87
    passing parameters to, 87
    properties of, 85

Ordinary differential equations, solvers (cont.)
relative error, 83
statistics for, 85
stiff, 83, 88
Van der Pol's equation, 88–89
`orth`, 58, 59, 109
Orthonormal basis, 58
`otherwise`, 63, 109
Outer product, 24
Overdetermined system, *See* Linear system of equations

P

Parentheses, 8
`path`, 68, 107
Path, *See* Search path
`pause`, 67, 70, 73, 107
`persistent`, 72, 73, 110
`pi`, 7, 8, 104
Piecewise polynomials, *See* Interpolation
`pinv`, 51, 52, 109
`plot`, 34, 35, 39, 42, 92, 107
Plot, generating a, *See* Graphics
Plotting
a curve, 34, 39
a function, 36, 41
a parametric function, 36
an implicit function, 36
in polar coordinates, 36
`plot3`, 39, 41, 107
`polar`, 36, 39, 107
Polar coordinates, 36
`poly`, 91, 93, 112
`polyder`, 92, 93, 112
`polyfit`, 92, 93
Polynomials, Sect. 11 (91), 112
differentiating, 92
evaluating, 91
finding minimum and maximum of, 92
order of, 92
representing by vector
roots of, 91
`polyval`, 91, 92, 93, 112
`polyvalm`, 92, 93, 112
Positive definite matrix, *See* Matrix
Power, 96, 97
average, 97
definition of, 96
frequency of, 97
in each mode, 97
instantaneous, 97
spectrum, 97
`prealloc` (locally defined), 69
Predefined variables, *See* Variables
Principles about computer arithmetic, 8, 10
`print`, 38, 105
Printing, *See* Display
`prod`, 29, 108
Product
Product (cont.)
dot, *See* Dot product
inner, *See* Inner product
outer, *See* Outer product
Programming language (MATLAB), Sect. 8 (60)
flow control, Subsect. 8.1 (60), 63, 109
break out of, 62
continue loop, 62
`for` loops, 60
`if` statement, 60
`switch` statement
different than in C, 63
`while` loops, 62
needed less frequently, 63
Pseudoinverse, *See* Matrix
Pseudorandom numbers, *See* Random numbers
Pythagorean theorem, 10

Q

QR decomposition, 58
`qr`, 58, 59, 109
`quad`, 95, 96, 112
Quadratic polynomial, roots of, 11
`quad8`, 95, 96, 112
Quote mark, 6

R

`rand`, 17, 18, 36, 50, 106
`randn`, 17, 18, 37, 106
Random matrix, 17, 21, 50, 81, 111
Random numbers, 17
Gaussian distribution, 17, 37
normal distribution, 17
pseudorandom numbers, 17
seed, 17
uniform distribution, 17, 36
`rank`, 58, 59, 109
Rank of matrix, 58
`rats`, 52, 107
Rational approximation to floating-point number, 52, 107
RCOND, 50, 55, 56
`real`, 12, 105
`realmax`, 8, 104
`realmin`, 8, 9, 104
Reduced row echelon form, 47
round-off errors in, 49
Relational operators, 61, 109
`<`, 61, 109
`<=`, 61, 109
`>`, 61, 109
`>=`, 61, 109
`==`, 61, 109
`~=`, 61, 109
matrix, Subsect. 8.2 (63)
result of, 64
`rem`, 11, 105

Remainder, 11, 105
Request input, 9
`repmat`, 21, 22, 106
`reshape`, 20, 21, 22, 106
`return`, 69, 71, 73, 110, 111
RGB components (of a color), 44
`roots`, 91, 93, 112
`round`, 11, 105
Round-off errors, Subsect. 1.3 (8), 10, 19, 21, 23, 26, 27, 34, 49, Subsect. 5.2 (49), 50, 56
`rref`, Sect. 5 (46), 73, 109

S

`save`, 13, 14, 104
Save terminal commands, 5
Save work, 5
Scientific notation, 5
Script files, 66, 67, 68, 70
  debugging, 70
  names of, 67
Search path, 68, 73
`semilogx`, 36, 39, 107
`semilogy`, 36, 39, 107
`sign`, 11, 105
Simpson's method (of numerical integration), 95
`sin`, 11, 105
$\sin z$, 12
Singular value decomposition, 58
`sinh`, 11, 105
`size`, 17, 18, 106
`sort`, 28, 29, 108
Sort numbers, 28
`sparse`, 78, 79, 81, 111
`spconvert`, 80, 81, 111
`spdiags`, 79, 81, 111
  differences from `diag`, 79
`speye`, 81, 106, 111
`spfun`, 81, 111
`spline`, 93, 112
Splines, *See* Interpolation
`sprand`, 80, 81, 106, 111
`sprandn`, 80, 81, 106, 111
`sprandsym`, 80, 81, 106
`sprintf`, 32, 33, 108
`spruce` (locally defined function), 70
`spy`, 81, 111
`sqrt`, 11, 27, 105
`sqrtm`, 23, 102, 112
`squeeze`, 30, 106
`sscanf`, 33, 108
Standard deviation, 28
Statements
  executing in text variables, 74
  rerunning previous, 9
  separating on a line, 6, 8, 15, 22, 103
`std`, 28, 29, 108
Stiff ode, 83, 88
`strcmp`, 61, 108, 109
String, *See* Text string
`str2num`, 33, 108
Subfunctions, *See* Functions
`subplot`, 43, 45, 108
`sum`, 29, 64, 108
`surf`, 40, 41, 44, 107
Surface plot, 40
  changing view, 40
  filled-in, 40
  wire-frame, 40
`svd`, 58, 59, 109
SVD, *See* Singular value decomposition
`switch`, 62, 63, 109

T

$^{\mathrm{T}}$, *See* Transpose
`tan`, 11, 105
`tanh`, 11, 105
Taylor series expansion, 101
TEX, *See* Text string
`text`, 42, 43, 46, 108
Text string, 6, Sect. 3 (32), 108
  appending to, 32
  concatenating, 32
  converting to, 32
  comparing strings, 61
  executing, 74
  TEX commands in, 43
Text window, 34
`tic`, 24, 25, 107
Time, *See* `cputime`, `tic`, `toc`
`title`, 36, 39, 42, 107
`toc`, 24, 25, 107
Transpose, 16, 18, 106
  conjugate, 16, 18, 106
Trigonometric functions, Subsect. 1.5 (10), Subsect. 2.6 (27)
`tril`, 21, 22, 106
`triu`, 20, 22, 106
TRUE (result of logical expression), 62
`type`, 13, 14, 67, 104

U

Underdetermined system, *See* Linear system of equations

V

Van der Pol's equation, *See* Ordinary differential equations
`vander`, 92, 106
Vandermonde matrix, 92
Variables, Subsect. 1.2 (6)
  about, 8
  case sensitive, 8
  conflict between variable and function name, 10
  deleting, 8

Variables (cont.)
global, 72
inputting, 9
list of, 13
loading, 13
local, 67, 71
logical, 65
modifying, 72
overwriting, 6
persistent, 72
predefined, 7, 8, 104
ans, 7, 8, 104
eps, 8, 9, 62, 104
i, 5, 8, 104
Inf, 8, 35, 104
j, 5, 8, 104
NaN, 8, 104
overwriting, 7, 60
pi, 7, 8, 104
realmax, 8, 104
realmin, 8, 9, 104
saving, 13
saving local variables in functions, 72
special cases of vectors or matrices, 6
static, 72
text, 6, Sect. 3 (32)
See also Text string
typeless, 7, 72
vdp1 (locally defined), 88
vdp2 (locally defined), 88
Vector
average value of elements, 28
column vs. row, 15
deleting elements, 21
generating, Subsect. 2.1 (15)
individual elements, 17
logical, 65
"masking" elements of, 65
maximum value, 27
mean value of elements, 28
minimum value, 28
preallocation of, 68
repeated elements, testing for, 28
sort elements, 28
standard deviation of elements, 28
sum of elements, 28
vectorize, 33, 108
Vectorizing code, Subsect. 8.5 (75)
view, 40, 41, 107

W

while, 62, 63, 109
who, 13, 14, 104
whos, 13, 14, 104
Workspace, 5, 70

X

xlabel, 36, 39, 42, 43, 107
xor, 62, 64, 109

Y

ylabel, 36, 39, 42, 43, 107

Z

zeros, 17, 18, 106
zlabel, 41, 42, 44, 107