MATLAB® Programming for Engineers

Second Edition

Updated Series

Stephen J. Chapman

Bookware Companion Series

Contents

Section 1

Engineering Problem Solving

Engineering often involves applying a consistent, structured approach to the solving of problems.

A general problem-solving approach and method can be defined, although variations will be required for specific problems.

Problems must be approached methodically, applying an *algorithm*, or step-by-step procedure by which one arrives at a solution.

1.1 Problem-Solving Process

The **problem-solving process** for a computational problem can be outlined as follows:

1. Define the problem.
2. Create a mathematical model.
3. Develop a computational method for solving the problem.
4. Implement the computational method.
5. Test and assess the solution.

The boundaries between these steps can be blurred and for specific problems one or two of the steps may be more important than others. Nonetheless, having this approach and strategy in mind will help to focus our efforts as we solve problems

1. Problem Definition:

The first steps in problem solving include:

- Recognize and define the problem precisely by exploring it thoroughly (may be the most difficult step).

- Determine what question is to be answered and what output or results are to be produced.
- Determine what theoretical and experimental knowledge can be applied.
- Determine what input information or data is available

Many academic problems that you will be asked to solve have this step completed by the instructor. For example, if your instructor ask you to solve a quadratic algebraic equation and provides you with all of the coefficients, the problem has been completely defined before it is given to you and little doubt remains about what the problem is.

If the problem is not well defined, considerable effort must be expended at the beginning in studying the problem, eliminating the things that are unimportant, and focusing on the root problem. Effort at this step pays great dividends by eliminating or reducing false trials, thereby shortening the time taken to complete later steps.

After defining the problem:

- Collect all data and information about the problem.
- Verify the accuracy of this data and information.
- Determine what information you must find: intermediate results or data may need to be found before the required answer or results can be found.

2. Mathematical Model:

To create a mathematical model of the problem to be solved:

- Determine what fundamental principles are applicable.
- Draw sketches or block diagrams to better understand the problem.
- Define necessary variables and assign notation.
- Reduce the problem as originally stated into one expressed in purely mathematical terms.
- Apply mathematical expertise to extract the essentials from the underlying physical description of the problem.
- Simplify the problem only enough to allow the required information and results to be obtained.
- Identify and justify the assumptions and constraints inherent in this model.

3. Computational Method:

A computational method for solving the problem is to be developed, based on the mathematical model.

- Derive a set of equations that allow the calculation of the desired parameters and variables.

- Develop an algorithm, or step-by-step method of evaluating the equations involved in the solution.
- Describe the algorithm in mathematical terms and then implement as a computer program.
- Carefully review the proposed solution, with thought given to alternative approaches.

4. Implementation of Computational Method:

Once a computational method has been identified, the next step is to carry out the method with a computer, whether human or silicon.

Some things to consider in this implementation:

- Assess the computational power needed, as an acceptable implementation may be hand calculation with a pocket calculator.
- If a computer program is required, a variety of programming languages, each with different properties, are available.
- A variety of computers, ranging from the most basic home computers to the fastest parallel supercomputers, are available.
- The ability to choose the proper combination of programming language and computer, and use them to create and execute a correct and efficient implementation of the method, requires both knowledge and experience. In your engineering degree program, you will be exposed to several programming languages and computers, providing you with some exposure to this issue.

The mathematical algorithm developed in the previous step must be translated into a computational algorithm and then implemented as a computer program.

The steps in the algorithm should first be outlined and then decomposed into smaller steps that can be translated into programming commands.

One of the strengths of Matlab is that its commands match very closely to the steps that are used to solve engineering problems; thus the process of determining the steps to solve the problem also determines the Matlab commands. Furthermore, Matlab includes an extensive toolbox of numerical analysis algorithms, so the programming effort often involves implementing the mathematical model, characterizing the input data, and applying the available numerical algorithms.

5. Test and Assess the Solution:

The final step is to test and assess the solution. In many aspects, assessment is the most open-ended and difficult of the five steps involved in solving computational problems.

The numerical solution must be checked carefully:

- A simple version of the problem should be hand checked.

- The program should be executed on obtained or computed test data for which the answer or solution is either known or which can be obtained by independent means, such as hand or calculator computation.
- Intermediate values should be compared with expected results and estimated variations. When values deviate from expected results more than was estimated, the source of the deviation should be determined and the program modified as needed.
- A "reality check" should be performed on the solution to determine if it makes sense.
- The assumptions made in creating the mathematical model of the problem should be checked against the solution.

1.2 Problem Solving Example

As an example of the problem solving process, consider the following problem:

> A small object is launched into flight from the ground at a speed of 50 miles/hour at 30 degrees above the horizontal over level ground. Determine the time of flight and the distance traveled when the ball returns to the ground.

1. **Problem Definition:**

   As stated above, this problem is well defined. The following items could be considered in further defining the problem.

   - Additional information needed:
     - Properties of the object and the flight medium could affect the flight trajectory. For example, if the object is light in weight, has a relatively large surface area, and travels in air, the air resistance would affect its flight. If the air were moving (by wind, for example), the flight of the object would also be affected. If this information is not available, it will be necessary to assume that the medium has no affect on the trajectory of flight.
     - Acceleration of gravity also affects the flight. If no further information is available, it would be reasonable to assume that the gravitational acceleration at the surface of the Earth would apply.
     - The accuracy of the initial speed and angle of the object is needed to determine the necessary accuracy of the quantities to be computed. The problem is one for which we have some physical insight, as it could represent the throwing of a baseball, although it is thrown from the ground instead of shoulder level. From this interpretation, the initial speed and angle seem reasonable.
     - Unit conversions needed:
       1 mile = 5280 feet
       1 hour = 60 minutes = 3600 seconds
       360 degrees = 2π radians

- Output or results to be produced: It is clear from the problem statement that the results to be produced are the time of flight and the distance traveled. The units for these quantities haven't been specified, but from our knowledge of throwing a baseball, computing time in seconds and distance in feet would be reasonable.
- Theoretical and experimental knowledge to be applied: The theory to be applied is that of ballistic motion in two dimensions.
- Input information or data: This includes the object initial velocity of 50 miles per hour at an angle 30 degree above horizontal.

2. **Mathematical Model:**

To pose this problem in terms of a mathematical model, we first need to define the notation:

- Time: t (s), with $t = 0$ when the object is launched.
- Initial velocity magnitude: $v = 50$ miles/hour.
- Initial angle: $\theta = 30°$.
- Horizontal position of ball: $x(t)$ (ft).
- Vertical position of ball: $y(t)$ (ft).
- Acceleration of gravity: $g = 32.2$ ft/s^2, directed in negative y direction.

The key step in developing a mathematical model is to divide the trajectory into its horizontal and vertical components. The initial velocity can be divided in this way, as shown in Figure 1.1. From basic trigonometry, we know that

$$v_h = v \cos\theta$$

$$v_v = v \sin\theta$$

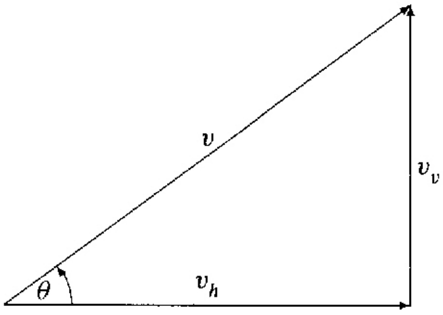


Figure 1.1: Initial velocity (v) divided into horizonal (v_h) and vertical (v_v) components.

Given the horizontal and vertical components of the initial velocity, the horizontal and vertical positions can be determined as functions of time. Since there is no external force acting to retard the horizontal motion, the object will move at a constant speed of v_h in the horizontal direction

$$x(t) = vt \cos\theta$$

In the vertical direction, the object motion is retarded by gravity and its position is

$$y(t) = vt \sin\theta - \frac{1}{2}gt^2$$

3. **Computational Method:**

   Using the model developed above, expressions for the desired results can be obtained. The object will hit the ground when its vertical position is zero

   $$y(t) = vt\sin\theta - \frac{1}{2}gt^2 = 0$$

   which can be solved to yield two values of time

   $$t = 0, \quad \frac{2v\sin\theta}{g}$$

   The second of the two solutions indicates that the object will return to the ground at the time

   $$t_g = \frac{2v\sin\theta}{g}$$

   The horizontal position (distance of travel) at this time is

   $$x(t_g) = vt_g\cos\theta$$

4. **Computational Implementation:**

   The equations defined in the computational method can be readily implemented using Matlab. The commands used in the following solution will be discussed in detail later in the course, but observe that the Matlab steps match closely to the solution steps from the computational method.

```
% Flight trajectory computation
%
% Initial values
g = 32.2;                        % gravity, ft/s^2
v = 50 * 5280/3600;              % launch velocity, ft/s
theta = 30 * pi/180;             % launch angle, radians

% Compute and display results
disp(’time of flight (s):’)      % label for time of flight
tg = 2 * v * sin(theta)/g        % time to return to ground, s
disp(’distance traveled (ft):’)  % label for distance
xg = v * cos(theta) * tg        % distance traveled

% Compute and plot flight trajectory
t = linspace(0,tg,256);
x = v * cos(theta) * t;
y = v * sin(theta) * t - g/2 * t.^2;
plot(x,y), axis equal, axis([ 0 150 0 30 ]), grid, ...
   xlabel(’Distance (ft)’), ylabel(’Height (ft)’), title(’Flight Trajectory’)
```

   Comments:

- Words following percent signs (%) are comments to help in reading the Matlab statements.
- A word on the left of an equals sign is known as a **variable name** and it will be assigned to the value or values on the right of the equals sign. Commands having this form are known as **assignment statements**.
- If a Matlab command assigns or computes a value, it will display the value on the screen if the statement does not end with a semicolon (;). Thus, the values of `v`, `g`, and `theta` will not be displayed. The values of `tg` and `xg` will be computed and displayed, because the statements that computes these values does not end with a semicolon.
- The three dots (...) at the end of a line mean that the statement continues on the next line.
- More than one statement can be entered on the same line if the statements are separated by commas.
- The statement `t = linspace(0,tg,256);` creates a vector of length 256.
- The expression `t.^2` squares *each element* in `t`, making another vector.
- In addition to the required results, the flight trajectory has also been computed and plotted. This will be used below in assessing the solution.
- The `plot` statement generates the plot of height against distance, complete with a title and labels on the x and y axes.

5. **Testing and Assessing the Solution:**

This problem is sufficiently simple that the results can be computed by hand with the use of a calculator. There is no need to generate test data to check the results. One could even question the need to use the power of Matlab to solve this problem, since it is readily solved using a calculator. Of course, our objective here is to demonstrate the application of Matlab to a problem with which you are familiar.

Executing the statements above provides the following displayed results:

```
time of flight (s):
tg =
    2.2774
distance traveled (ft):
xg =
  144.6364
```

Computing these quantities with a calculator can be shown to produce the same results.

The flight trajectory plot that is generated is shown in Figure 1.2. This demonstrates another strength of Matlab, as it has taken care of the details of the plot generation, including axis scaling, label placement, etc. This result can be used to assess the solution. Our physical intuition tell us that this result appears to be correct, as we know that we could throw a baseball moderately hard and have it travel a distance of nearly 150 feet and that maximum height of about 20 feet also seems reasonable. The shape of the trajectory also fits our observations.

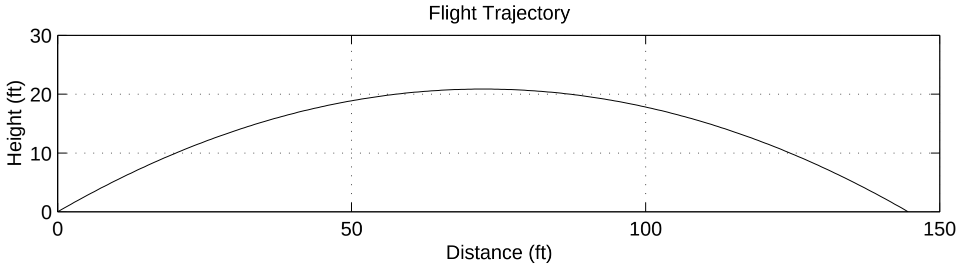


Figure 1.2: Matlab generated plot of flight trajectory

1.2.1 Programming Style

Programs that are not documented internally, while they may do what is request, can be difficult to understand when read a few months later, in order to correct or update them. Thus, it is extremely important to develop the art of writing programs that are well structured, with all of the logic clearly described. This is known as *programming style*. Elements of good programming style include:

- Use comments liberally, both at the beginning of a script, to describe briefly what it does and any special methods that may have been used, and also throughout the script to introduce different logical sections.
- Describe each variable briefly in a comment when it is initialized.
- Separate sections of code with blank lines.
- Indent multiple line structures to make them stand out as a logical unit.
- Use spaces in expressions to make them more readable (for example, on either side of operators and equal signs).

1.3 Computing Software

Before discussing Matlab in more detail, a brief discussion on computing software is useful.

Computer software contains the instructions or commands that the computer is to execute. Categories include:

- System software
- Languages
- Tools

System Software

Computer system software provides:

- An interface between the user and the hardware
- An environment for developing, selecting, and executing software.

The **operating system** is the most important part of the systems software, managing the computer resources, including:

- The individual user sessions on *multiuser* computer systems.
- The division of CPU time across various tasks.
- The *random access memory*, or RAM, where both instructions and data are stored.
- The secondary storage systems, such as disk drives, CD-ROM drives, tape drives, or any other device in which information is stored in binary form on some medium. Information in secondary storage is organized into units called *files*.

When a computer is turned on, it loads the operating system into RAM (usually from secondary storage) before a user can execute a program.

The system software may also include one or more **command shells**:

- Programs that direct the interaction with users. In most cases, when a you type a command, you are interfacing with the shell.
- Early shells provided a “text” only interface, but modern computers (particularly personal computers) have *graphical user interfaces* (GUIs) that allow users to express what they want to do by using a pointing device and selecting among choices displayed on the screen.

System software also includes:

- Language compilers
- Text editors
- Utilities for file management
- Utilities for printing

Computer Languages

Computer languages are used to develop programs to be executed. The can be described in terms of levels.

Machine language:

- Lowest-level language
- Binary-encoded instructions that are directly executed by the hardware.

- Language is closely tied to the hardware design.
- Machine specific: machine language for one computer is different from that of a computer having a different hardware design.

Assembly language:

- Also unique to a specific computer design.
- Commands or instructions are written in text statements instead of numbers.
- Assembly language programming requires good knowledge of the specific computer hardware. This is a tedious process, but it results in programs that are very fast, because they take advantage of the specific computer hardware.
- System software called an **assembler** translates the assembly language program to a machine language program for execution on the hardware.

High-level languages:

- Have English-like command syntax.
- Include languages such as Basic, Fortran, COBOL, Pascal, Ada, C, C++, and Java.
- Supported on many computer designs and knowledge of the specific hardware is not as critical in writing a program.
- System software called a **compiler** translates the program into machine language.

Compilation Process:

- Original program is called the **source program**
- Translated machine language program is called the **object program**.
- Errors detected by the compiler (called **compile errors** or **syntax errors**) must be corrected and compilation performed again.
- The process of compiling, correcting errors (or **debugging**) must often be repeated several times before the program compiles without compile errors.

Execution Process:

- To prepare the object program for **execution**, system software is applied to **link** other machine language statements to the object program and then **load** the program into memory.
- The program is executed and new errors, called execution errors, run-time errors or **logic errors** (also called **bugs**) may be identified.

- These program errors are caused when the programmer errs in determining the correct steps to be taken in the problem solution, such as an attempt to divide by zero.
- These errors must be corrected in the source program, which must again be compiled and link-loaded for execution.
- The process of program compilation, linking, and execution is shown in Figure 1.3.

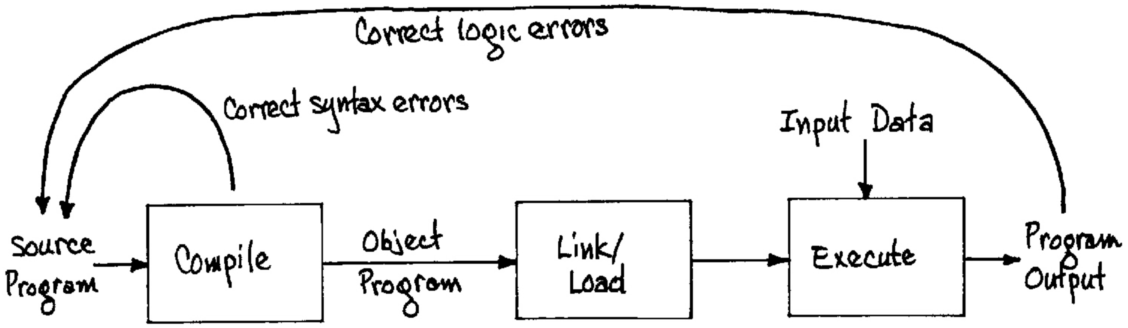


Figure 1.3: High-level language program development

Even when a program executes without an error message, the results must be checked carefully to be sure that they are correct. The computer performs the steps precisely as specified in the source; if the wrong steps are specified, the computer will execute these wrong (but syntactically correct) steps and produce a result that is incorrect.

Matlab **Mathematical Computation Tool**

Matlab and other mathematical computation tools are computer programs that combine computation and visualization power that make them particularly useful tools for engineers. Matlab is both a computer programming language and a software environment for using that language effectively.

The name Matlab stands for **Mat**rix **lab**oratory, because the system was designed to make matrix computations particularly easy. Don't worry if don't know what a matrix is, as this will be explained later. The Matlab environment allows the user to manage variables, import and export data, perform calculations, generate plots, and develop and manage files for use with Matlab.

The Matlab environment is an **interactive** environment:

- Single-line commands can be entered and executed, the results displayed and observed, and then a second command can be executed that interacts with results from the first command that remain in memory. This means that you can type commands at the Matlab prompt and get answers immediately, which is very useful for simple problems.
- Matlab is a executable program, developed in a high-level language, which *interprets* user commands.
- Portions of the Matlab program execute in response to the user input, results are displayed, and the program waits for additional user input.

- When a command is entered that doesn't meet the command rules, an error message is displayed. The corrected command can then be entered.
- Use of this environment doesn't require the compile-link/load-execution process described above for high-level languages.

While this interactive, line-by-line execution of Matlab commands is convenient for simple computational tasks, a process of preparation and execution of programs called **scripts** is employed for more complicated computational tasks:

- A **script** is list of Matlab commands, prepared with a text editor.
- Matlab executes a script by reading a command from the script file, executing it, and then repeating the process on the next command in the script file.
- Errors in the syntax of a command are detected when Matlab attempts to execute the command. A syntax error message is displayed and execution of the script is halted.
- When syntax errors are encountered, the user must edit the script file to correct the error and then direct Matlab to execute the script again.
- The script may execute without syntax errors, but produce incorrect results when a logic error has been made in writing the script, which also requires that the script be editted and execution re-initiated.
- Script preparation and debugging is thus similar to the compile-link/load-execution process required for in the development of programs in a high-level language.

1.4 Computing Terminology

Definitions of computing terms:

Command: A user-written statement in a computer language that provides instructions to the computer.

Variable: The name given to a quantity that can assume a value,.

Default: The action taken or value chosen if none has been specified.

Toggle: To change the value of a variable that can have one of two states or values. For example, if a variable may be "on" or "off" and the current value is "on," to toggle would change the value to "off."

Arguments: The values provided as inputs to a command.

Returns: The results provided by the computer in response to a command.

Execute: To run a program or carry out the instructions specified in a command.

Display: Provide a listing of text information on the computer monitor or screen.

Echo: To display commands or other input typed by the user.

Print: To output information on a computer printer (often confused with “display” in the textbook).

Section 2

Matlab Technical Computing Environment

Matlab provides a technical computing environment designed to support the implementation of computational tasks.

> Briefly, Matlab is an interactive computing environment that enables numerical computation and data visualization.

Matlab has hundreds of built-in functions and can be used to solve problems ranging from the very simple to the sophisticated and complex. Whether you want to do some simple numerical or statistical calculations, some complex statistics, solve simultaneous equations, make a graph, or run and entire simulation program, Matlab can be an effective tool.

Matlab has proven to be extraordinarily versatile and capable in its ability to help solve problems in applied math, physics, chemistry, engineering, finance – almost any application area that deals with complex numerical calculations.

2.1 Workspace, Windows, and Help

Running Matlab

- Unix: From a terminal window, type `matlab`, followed by the `Enter` key
- Win95: double-click on the Matlab icon or select Matlab from Start/Programs

Display Windows

- Command window
  Enter commands and data, display results
  Prompt >> or EDU>>

- Graphics (Figure) window
  Display plots and graphs
  Created in response to graphics commands
- M-file editor/debugger window
  Create and edit scripts of commands called M-files

When you begin MATLAB, the command window will be the active window. As commands are executed, appropriate windows will automatically appear; you can activate a window by clicking the mouse in it.

Getting Help

- `help` – On-line help, display text at command line
  `help`, by itself, lists all help topics
  `help` *topic* provides help for the specified topic
  `help` *command* provides help for the specified command
  `help help` provides information on use of the `help` command
- `helpwin` – On-line help, separate window for navigation.
- `helpdesk` – Comprehensive hypertext documentation and troubleshooting
- `demo` – Run demonstrations
- `intro` – Interactive introduction to MATLAB

Interrupting and Terminating MATLAB

- `Ctrl-C` (pressing the Ctrl and c keys simultaneously): Interrupts (aborts) processing, but does not terminate MATLAB. You may want to interrupt MATLAB if you mistakenly command it to display thousands of results and you wish to stop the time-consuming display.
- `quit`: Terminates MATLAB
- `exit`: Terminates MATLAB
- Select **Exit** under **File** menu: Terminates MATLAB (MS Windows)

2.2 Scalar Mathematics

Scalar mathematics involves operations on **single-valued** variables. MATLAB provides support for scalar mathematics similar to that provided by a calculator.

For more information, type `help ops`.

The most basic MATLAB command is the mathematical **expression**, which has the following properties:

- Mathematical construct that has a value or set of values.
- Constructed from numbers, operators, and variables.
- Value of an expression found by typing the expression and pressing $\boxed{\text{Enter}}$

2.2.1 Numbers

Matlab represents numbers in two form, fixed point and floating point.

Fixed point: Decimal form, with an optional decimal point. For example:

```
2.6349     -381       0.00023
```

Floating point: Scientific notation, representing $m \times 10^e$

For example: 2.6349×10^5 is represented as `2.6349e5`

It is called *floating point* because the decimal point is allowed to move. The number has two parts:

- *mantissa* m: fixed point number (signed or unsigned), with an optional decimal point (`2.6349` in the example above)
- *exponent* e: an integer exponent (signed or unsigned) (`5` in the example).
- Mantissa and exponent must be separated by the letter `e` (or `E`).

Scientific notation is used when the numbers are very small or very large. For example, it is easier to represent 0.000000001 as `1e-9`.

2.2.2 Operators

The evaluation of expressions is achieved with arithmetic *operators*, shown in the table below. Operators operate on *operands* (`a` and `b` in the table).

Operation	Algebraic form	Matlab	Example
Addition	$a + b$	`a + b`	`5+3`
Subtraction	$a - b$	`a - b`	`23-12`
Multiplication	$a \times b$	`a * b`	`3.14*0.85`
Right division	$a \div b$	`a / b`	`56/8`
Left division	$b \div a$	`a \ b`	`8\56`
Exponentiation	a^b	`a ^ b`	`5^2`

Examples of expressions constructed from numbers and operators, processed by Matlab:

```
>> 3 + 4
```

```
ans =
     7
>> 3 - 3
ans =
     0
>> 4/5
ans =
    0.8000
```

Prompt >> is supplied by Matlab, indicates beginning of the command.

`ans =` following the completion of the command with the `Enter` key marks the beginning of the answer.

Operations may be chained together. For example:

```
>> 3 + 5 + 2
ans =
    10
>> 4*22 + 6*48 + 2*82
ans =
   540
>> 4 * 4 * 4
ans =
    64
```

Instead of repeating the multiplication, the Matlab exponentiation or power operator can be used to write the last expression above as

```
>> 4^3
ans =
    64
```

Left division may seem strange: divide the right operand by the *left* operand. For *scalar* operands the expressions `56/8` and `8\56` produce the same numerical result.

```
>> 56/8
ans =
     7
>> 8\56
ans =
     7
```

We will later learn that *matrix* left division has an entirely different meaning.

Syntax

Matlab cannot make sense out of just any command; commands must be written using the correct syntax (rules for forming commands). Compare the interaction above with

```
>> 4 + 6 +
??? 4 + 6 +
          |
Missing operator, comma, or semi-colon.
```

Here, Matlab is indicating that we have made a syntax error, which is comparable to a misspelled word or a grammatical mistake in English. Instead of answering our question, Matlab tells us that we've made a mistake and tries its best to tell us what the error is.

Precedence of operations (order of evaluation)

Since several operations can be combined in one expression, there are rules about the order in which these operations are performed:

1. Parentheses, innermost first
2. Exponentiation (ˆ), left to right
3. Multiplication (*) and division (/ or \) with equal precedence, left to right
4. Addition (+) and subtraction (−) with equal precedence, left to right

When operators in an expression have the same precedence the operations are carried out from left to right. Thus `3 / 4 * 5` is evaluated as `( 3 / 4 ) * 5` and *not* as `3 / ( 4 * 5 )` .

2.2.3 Variables and Assignment Statements

Variable names can be assigned to represent numerical values in Matlab. The rules for these variable names are:

- Must start with a letter
- May consist only of the letters `a-z`, digits `0-9`, and the underscore character (_)
- May be as long as you would like, but Matlab only recognizes the first 31 characters
- Is case sensitive: `items, Items, itEms, and ITEMS` are all different variable names.

Assignment statement: Matlab command of the form:

- *variable* = *number*
- *variable* = *expression*

When a command of this form is executed, the expression is evaluated, producing a number that is assigned to the variable. The variable name and its value are displayed.

If a variable name is not specified, Matlab will assign the result to the default variable, `ans`, as shown in previous examples.

Example 2.1 *Expressions with variables*

```
>> screws = 32
screws =
    32
>> bolts = 18
bolts =
    18
>> rivets = 40;
>> items = screws + bolts + rivets
items =
    90
>> cost = screws * 0.12 + bolts * 0.18 + rivets * 0.08
cost =
   10.2800
```

- Variables: `screws, bolts, rivets, items, cost`
- Results displayed and stored by variable name
- Semicolon at the end of a line (as in the line `>> rivets=40;`) tells Matlab to evaluate the line but not to display the results

■

Matlab workspace: Variables created in the *Command* window are said to reside in the Matlab workspace or Base workspace. The workspace retains the values of these variables, allowing them to be used in subsequent expressions.

Thus, for the example above, we are able to compute the average cost per item by the following:

```
>> average_cost = cost/items
average_cost =
    0.1142
```

Because `average cost` is two words and Matlab variable names must be one word, an underscore was used to create the single Matlab variable `average_cost`.

If an expression contains variables, the value of the expression can be computed only if the values of the variables have been previously computed and still reside in the workspace. To compute the value of $y = x + 5$, you must first supply a value of x:

```
>> x=2
x =
     2
>> y=x+5
y =
     7
```

If you have not supplied a value for x, Matlab will return a syntax error:

```
>> y=x+5
??? Undefined function or variable 'x'.
```

Precedence of operations involving variables

Consider the computation of the area of a trapezoid whose parallel sides are of length b_1 and b_2 and whose height is h:

$$A = \frac{(b_1 + b_2)h}{2}$$

Matlab statement:

```
area = 0.5*(base_1 + base_2)*height;
```

Deleting the parentheses:

```
area = 0.5*base_1 + base_2*height;
```

This would compute

$$A = \frac{b_1}{2} + b_2 h$$

The incorrect result would be computed, but there would be no error messages, as the command has been written in correct Matlab syntax.

Style: writing arithmetic expressions

- Use parentheses often
  Consider the equation to convert from temperature in Fahrenheit (T_F) to temperature in Celsius (T_C):

  $$T_C = \frac{5}{9}(T_F - 32)$$

  This is computed correctly by the Matlab command:

  ```
  TC = 5/9*(TF-32)
  ```

MATLAB would first compute `TF-32`, replacing it with a value. Then `5/9` would be computed and this value would multiply the previously computed value of `TF-32`.

Easier to understand:

```
 TC = (5/9)*(TF-32)
```

- Use multiple statements
  Consider the equation:

  $$H(s) = \frac{s^2 + 4s + 13}{s^3 - 2s^2 + 4s + 5}$$

  MATLAB commands:

  ```
  numerator = s^2 + 4*s + 13;
  denominator = s^3 - 2*s^2 + 4*s + 5;
  H = numerator/denominator;
  ```

Special variables:

`ans`: default variable name
`pi`: ratio of circle circumference to its diameter, $\pi = 3.1415926...$
`eps`: smallest amount by which two numbers can differ
`inf` or `Inf` : infinity, e.g. 1/0
`nan` or `NaN` : not-a-number, e.g. 0/0
`date`: current date in a character string format, such as `19-Mar-1998`.
`flops`: count of floating-point operations.

Commands involving variables:

`who`: lists the names of defined variables
`whos`: lists the names and sizes of defined variables
`clear`: clears all variables, resets default values of special variables
`clear` *var*: clears variable *var*
`clc`: clears the command window, homes the cursor (moves the prompt to the top line), but does not affect variables.
`clf`: clears the current figure and thus clears the graph window.
`more on`: enables paging of the output in the command window.
`more off`: disables paging of the output in the command window.
When `more` is enabled and output is being paged, advance to the next line of output by pressing $\boxed{\text{Enter}}$; get the next page of output by pressing the spacebar. Press $\boxed{\text{q}}$ to exit out of displaying the current item.

Example 2.2 *Listing and clearing variables*

```
>> who
Your variables are:
```

```
average_cost   cost           rivets
bolts          items          screws
>> whos
  Name                Size         Bytes  Class

  average_cost        1x1              8  double array
  bolts               1x1              8  double array
  cost                1x1              8  double array
  items               1x1              8  double array
  rivets              1x1              8  double array
  screws              1x1              8  double array

Grand total is 6 elements using 48 bytes
```

Each variable occupies 8 *bytes* of storage. These variables each have a size `1x1` because they are *scalars*, as opposed to vectors or matrices.

■

Redefining variables

A variable may be redefined simply by executing a new assignment statement involving the variable. Note that previously issued commands involving the redefined variable won't be automatically reevaluated.

Example 2.3 *Variable redefinition*

```
>> screws = 32;
>> bolts = 18;
>> rivets = 40;
>> items = screws + bolts + rivets
items =
    90
>> screws = 36
screws =
    36
>> items
items =
    90
```

After computing `items`, `screws` was changed to 36, overwriting its previous value of 32. Note that the value of `items` has not changed. Unlike a spreadsheet, MATLAB does not recalculate the number of items based on the new value of `screws`. When MATLAB performs a calculation, it does so using the values it knows at the time the requested command is evaluated. In this example, to recalculate `items`, the `items` assignment statement must be re-issued:

```
>> items = screws + bolts + rivets
items =
    94
```

■

Command reuse and editing

Commands can be reused and editted using the following operations:

- Press the up arrow cursor key (↑) to scrolls backward through previous commands. Press Enter to execute the selected command.
- The down arrow cursor key (↓) scrolls forward through commands
- The left (←) and right arrow (→) cursor keys move within a command at the Matlab prompt, allowing the command to be edited.
- The mouse can also be used to reposition the command cursor, by positioning the mouse cursor and pressing the left mouse button.
- Other standard editing keys, such as delete Del, backspace BkSp, home Home, and end End, perform their commonly assigned tasks.
- Once a scrolled or edited command is acceptable, pressing Enter with the cursor anywhere in the command tells Matlab to process it.
- Escape key Esc erases the current command at the prompt.

Windows copy and paste operations:

- Copy: *Highlight* a command to be copied by placing the mouse cursor at the beginning of the text to be highlighted, press the left mouse button, and drag the mouse cursor through the text, releasing the mouse button when you reach the end of the text to be copied. The selected, or highlighted, text will appear as white text on black background instead of the reverse. In Unix, the highlighted text is automatically copied (stored internally). In MS Windows, the highlighted text is copied by pulling down the **Edit** menu and selecting **Copy** (or typing Ctrl+C).
- Paste: To paste the copied text in Unix, move the mouse cursor to the desired location, press the middle mouse button, and the text will be pasted into the window. In MS Windows, reposition the cursor, pull down the **Edit** menu and select **Paste** (or type Ctrl+V).

Punctuation and Comments

- Semicolon (`;`) at the end of a command suppresses the display of the result

- Commas and semicolons can be used to place multiple commands on one line, with commas producing display of results, semicolons supressing
- Percent sign (%) begins a comment, with all text up to the next line ignored by Matlab
- Three periods (...) at the end of a command indicates that the command continues on the next line. A continuation cannot be given in the middle of a variable name.

Example 2.4 *Use of punctuation*

```
>> screws = 36
screws =
    36
>> items
items =
    90
>> items = screws + bolts + rivets
items =
    94
>> screws=32, bolts=18; rivets=40;    % multiple commands
screws =
    32
>> items = screws + bolts + rivets;
>> cost = screws*0.12 + bolts*0.18 + rivets*0.08;
>> average_cost = cost/...            % command continuation
items
average_cost =
    0.1142
```

∎

2.3 Basic Mathematical Functions

Matlab supports many mathematical functions, most of which are used in the same way you write them mathematically.

Elementary math functions (enter `help elfun` for a more complete list):

`abs(x)`	Absolute value $\|x\|$
`sign(x)`	Sign, returns -1 if $x < 0$, 0 if $x = 0$, 1 if $x > 0$
`exp(x)`	Exponential e^x
`log(x)`	Natural logarithm $\ln x$
`log10(x)`	Common (base 10) logarithm $\log_{10} x$
`sqrt(x)`	Square root $\sqrt{x}$
`rem(x,y)`	Remainder of x/y. For example, `rem(100,21)` is 16. Also called the **modulus** function.

Information about these functions is displayed by the command `help` *function.* For example:

```
>> help sqrt

 SQRT   Square root.
    SQRT(X) is the square root of the elements of X. Complex
    results are produced if X is not positive.

    See also SQRTM.
```

Example 2.5 *Use of math functions*

$$x = \frac{\sqrt{2}}{2}$$

$$y = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

$$z = 20 \log_{10} y$$

```
>> x = sqrt(2)/2
x =
    0.7071
>> y = exp(-(x^2)/2)/sqrt(2*pi)
y =
    0.3107
>> z = 20*log10(y)
z =
  -10.1533
```

■

Example 2.6 *Solving for quadratic roots*

Problem: Solve for s: $2s^2 + 10s + 12 = 0$

Analysis: Derive and apply the quadratic equation by first expressing the quadratic polynomial in parametric form

$$as^2 + bs + c = 0$$

Assuming $a \neq 0$, rewrite the equation as

$$s^2 + \frac{b}{a}s + \frac{c}{a} = 0$$

To solve, "complete the square" in s

$$\left(s+\frac{b}{2a}\right)^2-\left(\frac{b}{2a}\right)^2+\frac{c}{a}=0$$

Rewrite to leave only the term involving s on the left hand side

$$\left(s+\frac{b}{2a}\right)^2=\left(\frac{1}{2a}\right)^2\left(b^2-4ac\right)$$

Take the square root of each side to find the solutions

$$s_{1,2}=-\frac{b}{2a}\pm\frac{1}{2a}\sqrt{b^2-4ac}$$

There are two solutions, so it is necessary to rewrite the equation in a different form to develop the Matlab commands.

$$\begin{aligned}
x &= -\frac{b}{2a}\\
y &= \frac{\sqrt{b^2-4ac}}{2a}\\
s_1 &= x+y\\
s_2 &= x-y
\end{aligned}$$

Matlab session:

```
>> a=2;
>> b=10;
>> c=12;
>> x = -b/(2*a);
>> y = sqrt(b^2-4*a*c)/(2*a);
>> s1 = x+y
s1 =
    -2
>> s2 = x-y
s2 =
    -3
```

■

2.4 Computational Limitations

To understand the computational limitations of Matlab and other computer software, it is necessary to consider the methods for representing numbers in digital computers.

Fixed-Point Numbers

All computer instructions and data are represented and stored in binary (base 2) form. In the representation of **fixed-point** numbers, the value of each digit in the number depends on its position relative to the fixed decimal point. For example the integer value 53 in base 10 is represented by the binary or base 2 value 00110101, denoted by

$$53_{10} = 00110101_2$$

In base 10, this means

$$53_{10} = 5 \times 10^1 + 3 \times 10^0$$

Similarly,

$$\begin{aligned} 00110101_2 &= 0 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 32_{10} + 16_{10} + 4_{10} + 1_{10} \end{aligned}$$

Thus, each bit position represents an increasing power of 2, beginning with 2^0 on the right.

Floating-Point Numbers

To represent non-integer numbers, a number representation known as **floating-point** must be defined. Floating-point numbers are used on computers to approximate a subset of the real numbers. While there are many possible floating-point number systems, all such systems consist of zero, a set of positive numbers, and the corresponding set of negative numbers.

The set of base 10 floating-point numbers consists of every number that can be written in the form $\pm m \times 10^e$, where

- m (the *mantissa*) is in the range $0 \leq m < 10$ and contains p digits (p is called the *precision*).
- e (the *exponent*) is an integer that lies between e_{min} (the minimum exponent) and e_{max} (the maximum exponent).

Our definition is for *base 10* floating-point numbers, since you either are, or will become, accustomed to base 10. However, computers use *base 2* numbers of the form $\pm m \times 2^e$. In MATLAB, when you enter a number in base 10, it is converted to base 2. Similarly, a base 2 result will converted to base 10 before being displayed. The essential properties of base 10 and base 2 number systems are the same, so our discussion of floating-point numbers will focus on base 10 represntations.

To represent a real number x with a floating-point number, we first round x to the closest real number y that has a p-digit mantissa. There are then two possibilities:

- If y is a member of the set of floating-point numbers, we say that x is *represented* by y in the floating-point number system. The absolute difference $|x - y|$ is called the *roundoff error.*

- If y is not a member of the set of floating-point numbers, we say that x is *not representable* in the floating-point number system. This can happen if x is too large, which is called overflow error; or if the absolute value of x is too small, which is called an underflow error.

In MATLAB, numbers are typically represented in a floating-point representation conforming to a standard established by the Institute of Electrical and Electronics Engineers (IEEE) in 1985. In the IEEE *double-precision* standard used by MATLAB, there are 53 bits in the mantissa and 11 bits in the exponent, for a total of 64 bits to represent a scalar number. This provides a range of values extending from 10^{-308} to 10^{308}. A *single-precision* representation is usually considered to be a 32-bit representation, but this is not supported by MATLAB.

Two MATLAB functions, `realmax` and `realmin`, display the largest and the smallest numbers, respectively, that can be represented:

```
>> realmax
ans =
  1.7977e+308
>> realmin
ans =
  2.2251e-308
```

If two numbers differ by one in the least-significant of the p digits of the mantissa, this difference is given by the value of the special variable `eps`:

```
>> eps
ans =
  2.2204e-016
```

Note that `e` means *exponent*, not the base of the natural logarithms, so that `ans` above represents 2.2204×10^{-16}.

For an example of **exponent overflow**, consider

```
>> x = 2.5e200;
>> y = 1.0e200;
>> z = x*y
z =
   Inf
```

The result for `z` should have been `2.5e400`, but since this overflowed the exponent, the result was displayed as infinity, represented by the special variable `Inf`.

For an example of **exponent underflow**, consider

```
>> x = 2.5e-200;
>> y = 1.0e-200;
>> z = x*y
z =
     0
```

Here `z` should have been `2.5e-400`, but due to exponent underflow, the result is displayed as 0.

Due to the finite accuracy of the representation of numbers in a computer, errors can be made in computations. For example, we know that $1 - 5 \times 0.2 = 0$; however, MATLAB produces the following:

```
>> 1 -0.2 -0.2 -0.2 -0.2 -0.2
ans =
  5.5511e-017
```

The result is a very small number, but it is not exactly zero. The reason is that the binary number corresponding to 0.2 is

$$0.001100110011001100\ldots$$

This representation requires an infinite number of digits. The consequence is that the computer works with an approximate value of 0.2. Subtracting the approximate value of 0.2 from 1 five times does not yield exactly 0.

2.5 Display Options

There are several ways to display the value of a variable. The simplest way is to enter the name of the variable. The name of the variable will be repeated, and the value of the variable will be printed starting with the next line. There are several commands that can be used to display variables with more control over the form of the display.

Number Display Options

MATLAB follows several rules in displaying numerical results.

- **Integers:** Displayed as integers, as long as they contain 9 digits or less. For 10 digits or more, they are displayed in scientific notation, described below. For example:

  ```
  >> x = 5
  x =
       5
  ```

- **Short fixed point:** The default is to display as a decimal number with four digits to the right of the decimal point, and up to three digits to the left of the decimal point, if this is possible. This is known as the `short` format. For example:

```
>> 610./12.
ans =
   50.8333
```

- **Short floating point:** When the result has more than three digits to the left of the decimal point, it is displayed in scientific notation (called the `short e` format). Scientific notation expresses a value as a number between 1 and 10 multiplied by a power of 10. An example of the `short e` format, which shows four digits to the right of the decimal point:

  ```
  >> 61000./12.
  ans =
    5.0833e+003
  ```

  In this format, 5.08333 is called the mantissa, `e` means exponent, which has the value +003. Written in the more common scientific notation, the result is 5.0833×10^3.

The default behavior can be changed by specifying a different numerical format within the Preferences menu item in the File menu, or by typing a FORMAT command at the command prompt. The table below indicates the affect of changing the display format for the variable `ratio`, computed as:

```
>> ratio = 610./12
ratio =
   50.8333
```

Matlab Command	`ratio`	Comments
`format short`	50.8333	4 decimal digits
`format long`	50.83333333333334	14 decimal digits
`format short e`	5.0833e+001	4 decimal digits plus exponent
`format long e`	5.083333333333334e+001	14 decimal digits plus exponent
`format short g`	50.8333	better of `format short` or `format short e` (default), switching for `ans` > 1000
`format long g`	5.083333333333334e+001	better of format long or format long e
`format bank`	50.83	2 decimal digits
`format +`	+	positive, negative, or zero

The bank format is a fixed format for dollars and cents. The + format displays the symbols +, −, and blank for positive, negative, and zero results, respectively.

Note:

- The display formats do not change the internal representation of a number; only the display changes.
- The internal representation is rounded when the display is shortened.

For example:

```
>> format long
>> 5.78/57.13
ans =
   0.10117276387187
>> format short
>> ans
ans =
    0.1012
```

Thus, the short result is displayed as 0.1012 instead of 0.1011, which would have been the result if the long display had been truncated to four decimal digits.

`format compact` suppresses many of the line feeds that appear between displays and allows more lines of information to be seen together on the screen. For example:

```
>> temp = 78

temp =

    78
>> format compact
>> temp=78
temp =
    78
```

In the examples shown in these notes, it is be asseumed that this command has been executed. The command `format loose` can be used to return to the less compact display mode.

Displaying Values and Text

There are three ways to display values and text in Matlab, to be described in this section:

1. By entering the variable name at the Matlab prompt, without a semicolon.
2. By use of the command `disp`.
3. By use of the command `fprintf`.

- **From the prompt:**

  As demonstrated in previous examples, by entering a variable name, an assignment statement, or an expression at the Matlab prompt, without a semicolon, the result will be displayed, proceeded by the variable name (or by `ans` if only an expression was entered). For example:

```
>> temp = 78
temp =
    78
```

- `disp`:

  There are two general forms of the command `disp` that are useful in displaying results and annotating them with units or other information:

  1. `disp(`*variable*`)`: Displays value of *variable* without displaying the variable name.
  2. `disp(`*string*`)`: Displays *string* by stripping off the single quotes and echoing the characters between the quotes.

  String: A group of keyboard characters enclosed in single quote marks (`'`). The quote marks indicate that the enclosed characters are to represent ASCII text.

  ```
  >> temp=78;
  >> disp(temp); disp('degrees F')
      78
  degrees F
  ```

  Note that the two `disp` commands were entered on the same line so that they would be executed together.

- `fprintf`

  One of the weaknesses of Matlab is its lack of good facilities for formatting output for display or printing. A function providing some of the needed capability is `fprintf`. This function is similar to the function of the same name in the ANSI C language, so if you are familiar with C, you will be familiar with this command. The `fprintf` function provides more control over the display than is provided with `disp`. In addition to providing the display of variable values and text, it can be used to control the format to be used in the display, including the specification to skip to a new line. The general form of this command is:

  `fprintf(`*`'format string', list of variables`*`)`

  The format string contains the text to be displayed (in the form of a character string enclosed in single quotes) and it may also contain format specifiers to control how the variables listed are embedded in the format string. The format specifiers include:

  w.d`%f` Display as fixed point or decimal notation (defaults to `short`), with a width of *w* characters (including the decimal point and possible minus sign, with *d* decimal places. Spaces are filled in from the left if necessary. Set *d* to 0 if you don't want any decimal places, for example `%5.0f`. Include leading zeros if you want leading zeroes in the display, for example `%06.0f`.

  w.d`%e` Display using scientific notation (defaults to `short e`), with a width of *w* characters (including the decimal point, a possible minus sign, and five for the exponent), with *d* digits in the mantissa after the decimal point. The mantissa is always adjusted to be less than 1.

  w.d`%g` Display using the shorter of tt short or `short e` format, with width *w* and *d* decimal places.

  `\n` Newline (skip to beginning of next line)

The *w.d* width specifiers are optional. If they are left out, default values are used.

Examples:

```
>> fprintf('The temperature is %f degrees F \n', temp)
The temperature is 78.000000 degrees F

>> fprintf('The temperature is %4.1f degrees F \n', temp)
The temperature is 78.0 degrees F
```

It should be noted that `fprintf` only displays the real part of a complex number, which is an important data type to be discussed later in the course.

2.6 Accuracy and Precision

Physical measurements cannot be assumed to be exact. Errors are likely to be present regardless of the precautions used when making the measurement. Quantities determined by analytical means are not always exact either. Often assumptions are made to arrive at an analytical expression that is then used to calculate a numerical value. The use of significant digits provides a method of expressing results and measurements that conveys how "good" these numbers are.

Significant Digits

A **significant digit** is defined as:

Any digit used in writing a number, *except*:

- Zeros used only for location of the decimal point
- Zeros that do not have any nonzero digit on their left.

Numbers larger than 10 that are not written in scientific notation can cause difficulties in interpretation when zeros are present. For example, 2000 could contain one, two, three, or four significant digits. If the number is written in scientific notation as 2.000×10^3, then clearly four significant digits are intended. Table 2.1 gives several examples.

When reading instruments, such as a measuring tape, thermometer, or fuel gauge, the last digit will normally be an estimate. That is, the instrument is read by estimating between the smallest graduations on the scale to get the final digit. It is standard practice to count the doubtful digit as significant.

In performing arithmetic operations, it is important to not lose the significance of the measurements, or, conversely, to imply precision that does not exist. The following are rules for determining the number of significant digits that should be reported following computations.

Multiplication and division: The product or quotient should contain the same number of significant digits as are contained in the number with the fewest significant digits.

Quantity	Signficant Digits
4782	4
600	1, 2, or 3
6.0×10^2	2
6.00×10^2	3
31.72	4
30.02	4
46.0	3
0.02	1
0.020	2
600.00	5

Table 2.1: Significant digits

Examples:

- $(2.43)(17.675) = 42.95025 \quad \Longrightarrow \quad 43.0$
- $(2.279\text{h})(60 \text{ min/h}) = 148.74 \text{ min} \Longrightarrow 148.7 \text{ min}$
  The conversion factor is exact (a definition).
- $(4.00 \times 10^2\text{kg})(2.2046\text{lbm/kg}) = 881.84\text{lbm} \quad \Longrightarrow \quad 882.\text{lbm}$
  The conversion factor is not exact, but it should not dictate the precision of the result if this can be avoided. A conversion factor should be used that has one or two more significant figures than will be reported in the result.
- $589.62/1.246 = 473.21027 \quad \Longrightarrow \quad 473.2$

Addition and subtraction: The result should show significant digits only as far to the right as is seen in the least precise number in the calculation.

Examples:

- $1725.463 + 189.2 + 16.73 = 1931.393 \quad \Longrightarrow \quad 1931.4$
- $897.0 - 0.0922 = 896.9078 \quad \Longrightarrow \quad 896.9$

Combined operations: While it would be good practice to convert the result of each operation to the proper number of signficant digits before applying another operation, it is normal practice to perform the entire calculation and then report a reasonable number of significant figures.

Rounding: In rounding a value to the proper number of significant figures, *increase the last digit retained by 1 if the first figure dropped is 5 or greater.* For example, consider the rounding performed by Matlab in displaying results in `short` format:

```
>> 2/9, 2.222222222222/4, 2/3, -2/3
```

```
ans =
    0.2222
ans =
    0.5556
ans =
    0.6667
ans =
   -0.6667
```

Accuracy, Precision, and Errors

Accuracy is a measure of the nearness of a value to the correct or true value.

Precision refers to the repeatability of a measurement, this how close successive measurements are to each other.

Errors: All measurements will have some degree of error. Identifiable and correctable errors are called **systematic**. Accidental or other nonidentifiable errors are called **random**.

Systematic Errors: Some errors will always have the same sign (+ or −) and are said to be systematic. Consider measure the distance between two points with a 25m steel tape. First, if the tape is not exactly 25.000m long, compared with the standard at the U.S. Bureau of Standards, there will be a systematic error. However, this error could be removed by applying a correction. A second source of error can be due to the temperature at the time of use and at the time when the tape was compared with the standard. A mathematical correction can be applied if this temperature is measured and the coefficient of thermal expansion of the tape is known. Another source of systematic is due to the difference in the tension applied to the tape while in use and the tension employed during standardization. Knowing the weight of the tape, the tension applied, and the length of the suspended tape, a correction can be calculated.

Random Errors: Even if all of the systematic errors have been eliminated, a measurement will still not be exact, due to random errors. For example, in reading a thermometer, the reading must be estimated when the indicator falls between graduations. Furthermore, variations in the ambient air temperature could cause the measurement to fluctuate. These errors can thus ssproduce measurements that are either too large or too small.

Repeated measurements of the same quantity will vary due to random error. However, it is impossible to predict the magnitude and sign of the accidental error present in any one measurement. Repeating measurements and averaging the results will reduce random error in the average. However, repeating measurements will not reduce the systematic error in the average result.

Roundoff Errors: Rounding off a number to n decimal places, following the rule described in the preceding section, produces an error whose absolute value is not larger than 0.5×10^{-n}. For example, assuming that the MATLAB value of π is correct, the error associated with rounding to 4 decimal places is:

```
>> E = pi - 3.1416
E =
 -7.3464e-006
```

This error is indeed smaller than the bound shown above. We have used the definition:

error = true value - calculated value

This is also called the **absolute error**, although the qualifier **absolute** may be used to indicate the absolute value of the error defined above.

The error can be compare to the true value by calculating the **relative error**

relative error = error/(true value)

For our example:

```
>> E/pi
ans =
 -2.3384e-006
```

This relative error can also be expressed in per cent:

```
>> 100*E/pi
ans =
 -2.3384e-004
```

Section 3

Files and File Management

In using MATLAB, you will read and write several types of *files*, or collections of computer data. In this section, you will learn

- File management definitions and commands
- Saving and restoring information
- Script M-files
- Errors and debugging
- Search path, path management, and startup

3.1 File Management Definitions and Commands

To describe file management, several concepts and definitions are introduced, then a few Unix commands are concisely described, followed by a description of MATLAB commands in greater detail.

File: A collection of computer data, either ASCII text or binary, that is stored on an external memory device, such as a disk or tape drive. The format of the file, or the order, size, and location of the data items stored in the file, is determined by the program that writes the file on the external memory device.

File System: The combination of hardware and software that allows files to be stored (written), retrieved (read), and managed.

Structure of file system: The file system in Unix and MS Windows has a tree structure, with the base called the root, dividing sequentially into branches, called directories or folders. Files can be thought of as leaves along the branches.

File types: For our purposes, there are two types of files: **binary** and **text** (also called ASCII text or plain text). Binary files contain both the machine instructions of executable programs and data stored in machine-readable form. Text files contain keyboard characters represented by one byte (8 bits) per character. Text, but not binary, files can be created and modified with text editors and printed on printers. Binary files can only be read an operated upon by programs designed to handle the internal formats of these files.

File management: The process, implemented with a set of user commands, to manage the files in a file system. This involves defining the tree structure by creating or deleting directories and managing files by creating, moving, renaming, or removing them and listing their names and attributes. In a Unix terminal window, a MS-DOS window, or a MATLAB Command window, file management is by typed commands (called command-line driven). In MS-Windows, file management is handled by a graphical user interface (GUI).

Directory: Branch of a file tree, containing files and subdirectories; also called a folder.

Root directory: Base of file system, denoted by "/" in Unix and "C:\" in MS-DOS and MS Windows.

Present working directory: The branch of the file system in which you are currently located. Also called "current directory." You have direct access to the files in this directory. Using "change directory" commands, you can move to another directory.

Home directory: When you log into your Unix account, your present working directory, is called your "home directory."

User files: The user is given permission to manage all files in directories (branches) said to be below the home directory.

Path: Sequence of directories (branches) leading to a specific directory or file.

Path separator: Character used to separate the directories (folders) in the path – "/" in Unix, "\" in MS-DOS and MS Windows.

Absolute path: A path beginning at the file system root, such as "/home/ford/teach/e6/" – also called the "full path."

Relative path: A path beginning at the present working directory. For example, if the present working directory is /home/ford/, then teach/e6 is the relative path to the directory e6.

Unix File Management

The following are the basic Unix commands for file management. For more detailed information, type `man` *command*. Most of these operations can be done from MATLAB, so it won't be necessary to learn them in great detail.

`pwd`: Print working directory – displays the full path of the present working directory.

`cd` *path*: Change to directory given by *path*, which can be either a relative or an absolute path.

`ls`: Display a list of the names of the directories and files in the present working directory.

`ls -l`: Display a list of names and associated access permissions, owner, size, and modification date of the directories and files in the present working directory.

`mkdir` *dir*: Create the directory named *dir* in the present working directory.

`rm` *file*: Delete *file* from current directory.

`more` *file*: Display the contents of *file* (text file only), one screen at a time (press spacebar to display the next screen, q to quit).

`cp` *file1 file2*: Make of copy of *file1* named *file2*.

`mv` *file1 file2*: Change the name of *file1* to *file2*.

`lp` *file*: Prints *file* (which must be a text file) on the user's default printer.

Matlab File Management

Matlab provides a group of commands to manage user files that are similar to those of Unix. For more information, type `help iofun`.

`pwd`: Print working directory – displays the full path of the present working directory.

`cd` *path*: Change to directory (folder) given by *path*, which can be either a relative or absolute path.

`dir` or `ls`: Display the names of the directories (folders) and files in the present working directory.

`what`: Display the names of the M-files and MAT-files in the current directory.

`delete` *file*: Delete *file* from current directory

`type` *file*: Display contents of *file* (text file only, such as an M-file).

3.2 Saving and Restoring Matlab Information

It is good engineering practice to keep records of calculations. These records can be used for several purposes, including:

- To revise the calculations at a later time.
- To prepare a report on the project.

Matlab provides several methods for saving information from a workspace session. Saving the session output with the `diary` command and saving and loading the variables with the `save` and `load` command are described in this section.

3.2.1 Diary Command

The `diary` commands allows you to record all of the input and displayed output from a Matlab interactive workspace session. The commands include:

- `diary` *file*: Saves all text from the Matlab session, except for the prompts (`>>`), as text in *file*, written to the present working directory. If *file* is not specified, the information is written to the file named `diary`.
- `diary off`: Suspends diary operation.
- `diary on`: Turns diary operation back on.
- `diary`: Toggles diary state

Example 3.1 *Use of diary command*

Consider again the example involving roots of the quadratic equation.

Problem: solve for s: $s^2 + 5s + 6 = 0$

$$s = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Matlab session:

```
>> diary roots
>> a=1;
>> b=5;
>> c=6;
>> x = -b/(2*a);
>> y = sqrt(b^2-4*a*c)/(2*a);
>> s1 = x+y
s1 =
    -2
>> s2 = x-y
s2 =
    -3
```

The file `roots` is written in your current working directory. It can be displayed by the Matlab command `type roots`. It can also be displayed in a Unix terminal window by the command `more roots` or printed with the command `lp roots`.

```
a=1;
b=5;
```

```
c=6;
x = -b/(2*a);
y = sqrt(b^2-4*a*c)/(2*a);
s1 = x+y
s1 =
    -2
s2 = x-y
s2 =
    -3
diary off
```

Note that this is nearly the same as the display in the command window, with the exception that the Matlab prompt (>>) is not included.

■

3.2.2 Saving and Retrieving Matlab Variables

There will be occasions when you want to save your Matlab variables so that you can later retrieve them to continue your work. In this case, you must save the information in the Matlab binary format, so that the full precision of the variables is retained. Files in this Matlab binary format are known as MAT-files and they have an extension of `mat`.

Storing and Loading Workspace Values

`save`	Stores workspace values (variable names, sizes, and values), in the binary file `matlab.mat` in the present working directory
`save data`	Stores all workspace values in the file `data.mat`
`save data_1 x y`	Stores only the variables `x` and `y` in the file `data_1.mat`
`load data_1`	Loads the values of the workspace values previously stored in the file `data_1.mat`

Exporting and Importing Data

There are also situations in which you wish to export Matlab data to be operated upon with other programs, or to import data created by other programs. This must be done with text files written with `save` or read with `load`.

To write a text file `data1.dat` in the current working directory containing values of Matlab variables in `long e` format:

```
save data1.dat -ascii
```

Note that the individual variables will not be identified with labels or separated in any way. Thus, if you have defined variables `a` and `b` in the Matlab workspace, the command above will output first the values in `a`, then the values in `b`, with nothing separating them. Thus, it is often desirable to write text files containing the values of only a single variable, such as:

```
save data2.dat a -ascii
```

This command causes each row of the array `a` (more on arrays later) to be written to a separate line in the data file. Array elements on a line are separated by spaces. The separating character can be made a tab with the command:

```
save data2.dat a -ascii -tab
```

The `.mat` extension is not added to an ASCII text file. However, it is recommended that ASCII text file names include the extension `.dat` so that it is easy to distinguish them from MAT-files and M-files (to be described in the next section).

For an example of importing text data, suppose that a text file named `data3.dat` contains a set of values that represent the time and corresponding distance of a runner from the starting line in a race. Each time and its corresponding distance value are on a separate line of the data file. The values on a line must be separated by one or more spaces. Consider a data file named `data3.dat` containing the following:

```
0.0    0.0
0.1    3.5
0.2    6.8
```

The `load` command followed by the filename will read the information into an array with the same name as the base name of the data file (extension removed). For example, the command

```
load data3.dat
```

reads the data in the file `data3.dat` and stores it in the Matlab array variable named `data3`, having two columns and three rows.

Matlab can also import data from and export data to two common formats that are used in spreadsheet and database program formats. The more common of these file formats is the *comma separated values* or `.csv` format. This is an ASCII text file, with values on each line separated by commas, as the name implies. The second format is the spreadsheet `.wk1` format, often used as an interchange format for different spreadsheet programs. The Matlab import and export commands for these formats are:

`A = csvread(’file’)` Reads a comma separated value formatted file `file`. The result is returned in `A`. The file can only contain numeric values.

`csvwrite(’file’,A)` Writes matrix `A` into `file` as comma separated values.

`A = wk1read(’file’)` Reads all the data from a WK1 spreadsheet file named `file` into matrix `A`.

`wk1write(’file’,A)` Writes matrix `A` into a WK1 spreadsheet file with the name `file`. ’`.wk1`’ is appended to the filename if no extension is given.

Use the help facility for information on options for these commands.

3.3 Script M-Files

For simple problems, entering commands at the Matlab prompt in the Command window is simple and efficient. However, when the number of commands increases, or you want to change the value of one or more variables, reevaluate a number of commands, typing at the Matlab becomes tedious. You will find that for most uses of Matlab, you will want to prepare a *script*, which is a sequence of commands written to a file. Then, by simply typing the script file name at a Matlab prompt, each command in the script file is executed as if it were entered at the prompt.

For more information, type `help script`.

Script File: Group of Matlab commands placed in a text file with a text editor. Matlab can open and execute the commands exactly as if they were entered at the Matlab prompt. The term "script" indicates that Matlab reads from the "script" found in the file. Also called "M-files," as the filenames must end with the extension '.m', e.g. `example1.m`.

M-files are text files and may be created and modified with any text editor. You need to know how to open, edit, and save a file with the text editor you are using. On a PC or Macintosh, an **M-file editor** may be brought up by choosing **New** from the **File** menu in the Matlab Command window and selecting **M-file**.

The script M-file is executed by choosing **Run Script...** from the **File** menu on a PC or Macintosh, or simply typing the name of the script file at the prompt in the Matlab command window.

Example 3.2 *Quadratic root finding script*

Create the file named `qroots.m` in your present working directory using a text editor:

```
% qroots: Quadratic root finding script
format compact;
a=1
b=5
c=6
x = -b/(2*a);
y = sqrt(b^2-4*a*c)/(2*a);
s1 = x+y
s2 = x-y
```

To execute the script M-file, simply type the name of the script file qroots at the Matlab prompt:

```
>> qroots
a =
     1
b =
     5
c =
     6
s1 =
    -2
s2 =
    -3
```

Comments:

- % `qroots`: % allows a comment to be added
- `format compact`: Suppresses the extra lines in the output display
- `a=1, b=5, c=6`: Sets values of `a, b, c`, will display result
- `x` & `y`: Semicolon at end of command means these intermediate values won't be displayed
- `s1, s2`: Compute and display roots

∎

Search rules for qroots command:

1. Display current Matlab variable qroots if defined
2. Execute built-in Matlab command qroots if it exists
3. Execute qroots.m if it can be found in the Matlab search path (described below)

Commands within the M-file have access to all variables in the Matlab workspace and all variables created by the M-file become part of the workspace.

Commands themselves are not normally displayed as they are evaluated. Placing the `echo on` command in the M-file will cause commands to be displayed. The command `echo off` (the default) turns off command display and `echo` by itself toggles the command echo state.

You could repeatedly edit `qroots.m`, change the values of a, b, and c, save the file, then have Matlab execute the revised script to compute a new pair of roots.

Matlab functions useful in M-files:

Command	Description
`disp(ans)`	Display results without identifying variable names
`echo [on|off]`	Control Command window echoing of script commands
`input('prompt')`	Prompt user with text in quotes, accept input until "Enter" is typed
`keyboard`	Give control to keyboard temporarily. Type `Return` to return control to the executing script M-file.
`pause`	Pause until user presses any keyboard key
`pause(n)`	Pause for n seconds
`waitforbuttonpress`	Pause until user presses mouse button or keyboard key

The input command is used for user input of data in a script and it is used in the form of an assignment statement. For example:

```
a = input('Enter quadratic coefficient a: ');
```

When this command is executed, the text string `Enter quadratic coefficient a:` is displayed as a user prompt in the command window. The user then types in data value, which is assigned to the variable a. Because this input command ends with a semicolon, the entered value of a is not displayed when the command is completed.

Example 3.3 *Revised quadratic roots script*

Change script for computing quadratic roots by:

1. Prompting for input of coefficients a, b, and c;
2. Format the display of the computed roots s1 and s2;

Script rqroots.m:

```
% rqroots: Revised quadratic root finding script

format compact;

% prompt for coefficient input

a = input('Enter quadratic coefficient a: ');
b = input('Enter quadratic coefficient b: ');
c = input('Enter quadratic coefficient c: ');
disp('')

% compute intermediate values x & y

x = -b/(2*a);
y = sqrt(b^2-4*a*c)/(2*a);

% compute and display roots

s1 = x+y;
disp('Value of first quadratic root:  '),disp(s1);
s2 = x-y;
disp('Value of second quadratic root: '),disp(s2);
```

Two examples of the results from this script:

```
>> rqroots
Enter quadratic coefficient a: 1
Enter quadratic coefficient b: 5
Enter quadratic coefficient c: 6
Value of first quadratic root:
    -2
Value of second quadratic root:
    -3
>> rqroots
Enter quadratic coefficient a: 1
Enter quadratic coefficient b: 4
Enter quadratic coefficient c: 8
Value of first quadratic root:
  -2.0000+ 2.0000i
Value of second quadratic root:
  -2.0000- 2.0000i
```

■

M-file Commands

Command	Description
`edit test`	Open `test.m` for editing using the built-in MATLAB text editor, same as **Open...** in **File** menu

Effective Use of Script Files

The following are some suggestions on the effective use of MATLAB scripts:

1. The name of a script file must follow the MATLAB convention for naming variables; that is, the name must begin with a letter and may include digits and the underscore character.

2. Do not give a script file the same name as a variable it computes, because MATLAB will not be able to execute that script file more than once unless the variable is cleared. Recall that typing a variable name at the command prompt causes MATLAB to display the value of that variable. If there is no variable by that name, then MATLAB searches for a script file having that name. For example, if the variable `rqroot` was created in a script file having the name `rqroot.m`, then after the script is executed the first time, the variable `rqroot` exists in the MATLAB workspace. If the script file is modified and an attempt is made to run it a second time, MATLAB will display the value of `rqroot` and will not execute the script file.

3. Do not give a script file the same name as a MATLAB command or function. You can check to see whether a function already exists by using the the `which` command. For example, to see whether `rqroot` already exists, type `which rqroot`. If it doesn't exist, MATLAB will display `rqroot not found.` If it does exist, MATLAB will display the full path to the function. For more details as to the existence of a variable, script, or function having the name `rqroot`, type `exist(’rqroot’)`. This command returns one of the following values:

0 if `rqroot` does not exist
1 if `rqroot` is a variable in the workspace
2 if `rqroot` is an M-file or a file of unknown type in the MATLAB search path
3 if `rqroot` is a MEX-file in the MATLAB search path
4 if `rqroot` is a MDL-file in the MATLAB search path
5 if `rqroot` is a built-in MATLAB function
6 if `rqroot` is a P-file in the MATLAB search path
7 if `rqroot` is a directory

4. As in interactive mode, all variables created by a script file are defined as variables in the workspace. After script execution, you can type `who` or `whos` to display information about the names, data types and sizes of these variables.

5. You can use the `type` command to display an M-file without opening it with a text editor. For example, to view the file `rqroot.m`, the command is `type rqroot`.

3.4 Errors and Debugging

You will find that you will seldom get your scripts to run correctly the first time. However, you shouldn't despair, as this is also the case for experienced programmers. You will need to learn to identify and correct the errors, known in computer jargon as *bugs*.

Syntax Errors

The most common type of error is the syntax error, a typing error in a MATLAB command (for example, `srqt` instead of `sqrt`. These errors are said to be *fatal*, as they cause MATLAB to stop execution and display an error message. Unfortunately, since the MATLAB command interpreter can easily be confused by these errors, the displayed error message may not be very helpful. It is your job as a programmer to make sense of the message and correct your script, a process known as *debugging*.

Some examples of syntax errors and associated messages include:

- Missing parenthesis

  ```
  >> 4*(2+5
  ??? 4*(2+5
            |
  A closing right parenthesis is missing.
  Check for a missing ")" or a missing operator.
  ```

  This message is helpful, as it even points out the location of the error.

- Missing operator

  ```
  >> 4(2+5)
  ```

```
??? 4(
     |
Missing operator, comma, or semi-colon.
```

In this error, an operator such as * was left out after 4.

- Misspelled variable name

  ```
  >> 2x = 4*(2+5)
  ??? 2
      |
  Missing operator, comma, or semi-colon.
  ```

  Here, if you intended the variable name to be x2 instead of 2x, the error message is misleading. You are at least shown the location of the error, allowing you to identify the error.

There are a large number of possible syntax errors, many of which you will discover in writing scripts for this course. With experience you will gradually become more adept at understanding and correcting your mistakes.

Run-time and Logic Errors

After correcting syntax errors, your script will execute, but it still may not produce the results you desire.

Run-time Errors: These errors occur when your script is run on a particular set of data. They typically occur when the result of some operation leads to NaN (not a number), Inf (infinity), or an empty array (to be covered later in the course).

Logic Errors: These are errors in your programming logic, when your script executes properly, but does not produce the intended result. When this occurs, you need to re-think the development and implementation of your problem-solving algorithm.

The following are suggestions to help with the debugging of Matlab scripts or M-files:

- Execute the script on a simple version of the problem, using test data for which the results are known, to determine which displayed results are incorrect.
- Remove semicolons from selected commands in the script so that intermediate results are displayed.
- Add statements that display variables of interest within the script.
- Place the keyboard command at selected places in the script to give temporary control to the keyboard. By doing so, the workspace can be interrogated and values changed as necessary. Resume script execution by issuing a return command at the keyboard prompt.

Later in the course, the Matlab debugging functions will be introduced to provide additional tools for correcting these errors.

Programming Style

As discussed previously, good programming style requires that comments be included liberally in a script file. Furthermore, the first comment line (known as the `H1` line) before any executable statement is the line that is searched by the `lookfor` command. This command aids the user in finding a suitable MATLAB command:

`lookfor xyz`	Looks for the string `xyz` in the first comment line (the `H1` line) in all M-files found on `matlabpath`. For all files in which a match occurs, `lookfor` displays the `H1` line.
`lookfor xyz -all`	Searches the entire first comment block of each M-file.

For example:

```
>> lookfor roots
rqroots.m: % rqroots: Revised quadratic root finding script
POLY Convert roots to polynomial.
ROOTS  Find polynomial roots.
```

Thus, the first line of a MATLAB script should be a comment containing the script name and key words describing its function. The next several lines (to be searched with the `-all` option of `lookfor`) should contain comments giving the author's name, the date the script was created, and a detailed description of the script.

3.5 Matlab Search Path, Path Management, and Startup

Matlab search path: Ordered list of directories that MATLAB searches to find script and function M-files stored on disk. Commands to manage this search path:

`matlabpath`: Display search path.

`addpath` *dir*: Add directory *dir* to beginning of matlabpath. If you create a directory to store your script and function M-files, you will want to add this directory to the search path.

`rmpath` *dir*: Remove directory *dir* from the matlabpath.

`path(p1,p2)`: Changes the path to the concatenation of the two path strings `p1` and `p2`. Thus `path(path,p)` appends a new directory to the current path and `path(p,path)` prepends a new path. If `p1` or `p2` are already on the path, they are not added. For example, the following statements add another directory to MATLAB's search path:

- Unix: `path(path,'/home/ford/matlabm')`
- DOS: `path(path,'TOOLS\GOODSTUFF')`

`editpath`: Edit matlabpath using MATLAB editor, same as **Set Path** in **File** menu.

`which test`: Display the directory path to `test.m`. Used to tell you which M-file script will be executed when you enter the command `test`.

Matlab at Startup

Two files are executed at startup: `matlabrc.m` and `startup.m`

`matlabrc.m`: Comes with Matlab, shouldn't be modified. Sets default Figure window size and placement, as well as a number of other default features.

`startup.m`: An optional M-file to be created by the user, typically containing commands that add personal default features. It is common to put addpath or path commands in `startup.m` to append additional directories to the Matlab search path. Since `startup.m` is a standard script M-file, then there are no restrictions as to what commands can be placed in it. For example, you might want to include the command `format compact` in `startup.m` so that the compact results display format becomes the default.

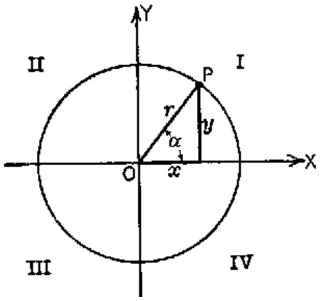
Section 4

Trigonometry and Complex Numbers

In this section, we will consider in greater detail two scalar mathematics tools that are important to engineers: trigonometry and complex numbers. We will find that these two topics are closely related.

4.1 Trigonometry

Definitions



In quadrant I:

$$\sin\alpha = \frac{y}{r}, \qquad \alpha = \arcsin\left(\frac{y}{r}\right) = \sin^{-1}\left(\frac{y}{r}\right)$$

$$\cos\alpha = \frac{x}{r}, \qquad \alpha = \arccos\left(\frac{x}{r}\right) = \cos^{-1}\left(\frac{x}{r}\right)$$

$$\tan\alpha = \frac{y}{x}, \qquad \alpha = \arctan\left(\frac{y}{x}\right) = \tan^{-1}\left(\frac{y}{x}\right)$$

$$r = \sqrt{x^2 + y^2}$$

`sin(alpha)` Sine of `alpha`

`cos(alpha)` Cosine of `alpha`

`tan(alpha)` Tangent of `alpha`

`asin(z)` Arcsine or inverse sine of `z`, where `z` must be between -1 and 1. Returns an angle between $-\pi/2$ and $\pi/2$ (quadrants I and IV).

`acos(z)` Arccosine or inverse cosine of `z`, where `z` must be between -1 and 1. Returns an angle between 0 and π (quadrants I and II).

`atan(z)` Arctangent or inverse tangent of `z`. Returns an angle between $-\pi/2$ and $\pi/2$ (quadrants I and IV).

`atan2(y,x)` Four quadrant arctangent or inverse tangent, where x and y are the coordinates in the plane shown in the figure above. Returns an angle between $-\pi$ and π (all quadrants), depending on the signs of `x` and `y`.

Example:

```
>>  z = sqrt(2)/2
z =
    0.7071
>> alpha = asin(z)
alpha =
    0.7854
>> alpha_deg = alpha*180/pi
alpha_deg =
   45.0000
```

Where `sqrt()` is square root and `asin()` is inverse sine or arcsine. Note that Matlab only works in radians, so the result `alpha` returned by asin is in radians. Using the property that 2π radians equals 360 degrees, a statement has been written to compute `alpha_deg`, the angle in degrees.

Note that care must be taken in computing an angle from an inverse trigonometric function. The functions `asin` and `atan` will yield angles only in quadrants I and IV: I for a positive argument and IV for a negative argument. The function `acos` yields angles only in quadrants I and II: I for a positive argument and II for a negative argument. The function `atan2` is best used to compute angles, as the lengths of both the x and y sides of the triangle are used in the calculation. Consider a case for quadrant II, with $x = -1$, $y = 1$:

```
>> x=-1;
>> y=1;
>> r=sqrt(x^2+y^2)
r =
    1.4142
>> theta_1=(180/pi)*asin(y/r)
theta_1 =
    45
>> theta_2=(180/pi)*atan(y/x)
theta_2 =
   -45
```

```
>> theta_3=(180/pi)*atan2(y,x)
theta_3 =
   135
```

Observe that only `theta_3` is the proper result, in quadrant II.

Example 4.1 *Estimating the height of a building*

Problem: Consider the problem of estimating the height of a building, as illustrated in Fig. 4.1. If the observer is a distance D from the building, the angle from the observer to the top of the building is θ, and the height of the observer is h, what is the building height?

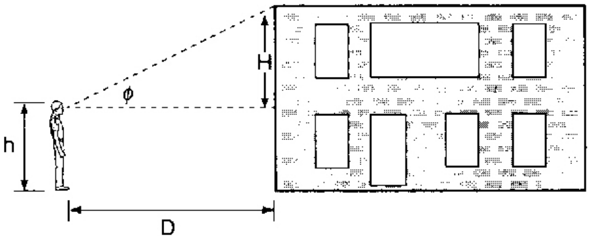


Figure 4.1: Estimating Building Height

Solution: Draw a simple diagram as shown in Fig. 4.1. The building height B is

$$B = h + H$$

where H is the length of the triangle side opposite the observer, which can be found from the triangle relationship

$$\tan\theta = \frac{H}{D}$$

Therefore, the building height is

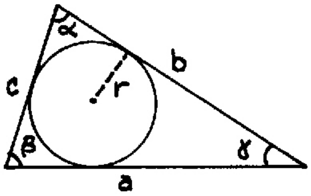
$$B = h + D \cdot \tan\theta$$

If h= 2 meters, D= 50 meters, and θ is 60 degrees, the MATLAB solution is

```
>> h=2;
>> theta=60;
>> D=50;
>> B = h+D*tan(theta*pi/180)
B =
   88.6025
```

∎

Solution of Triangles



Law of sines: $\dfrac{a}{\sin\alpha} = \dfrac{b}{\sin\beta} = \dfrac{c}{\sin\gamma}$

Law of cosines: $a^2 = b^2 + c^2 - 2bc\cos\alpha$

Law of tangents: $\dfrac{a-b}{a+b} = \dfrac{\tan\frac{1}{2}(\alpha-\beta)}{\tan\frac{1}{2}(\alpha+\beta)}$

Included angles: $\alpha + \beta + \gamma = \pi$ radians $= 180°$

Example 4.2 *Use of triangle relations*

In Figure 4.2, measurements of distances and angles around a lake have been indicated. The problem is to use these measurements to calculate the distance DE across the lake.

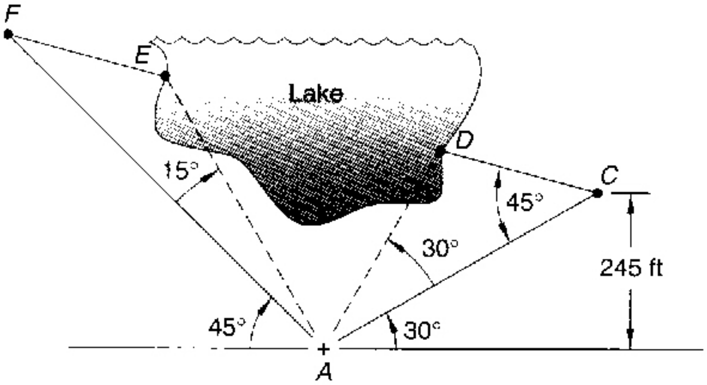


Figure 4.2: Distance and angle measurements near a lake

The solution is to sequentially apply the various triangle relations to determine the distances CF, CD, and EF, which can then be used to determine the specified distance DE.

First, determine the length of the sides of triangle ACF, beginning with the right triangle relationship on the bottom right:

$$\sin 30° = \frac{245}{AC}$$

Solving this equation for AC:

$$AC = \frac{245}{\sin 30^\circ}$$

From the included angles in triangle ACD:

$$D = 180^\circ - 30^\circ - 45^\circ$$

Applying the law of sines to triangle ACD:

$$\frac{CD}{\sin 30^\circ} = \frac{AC}{\sin D}$$

Solving for CD:

$$CD = AC\frac{\sin 30^\circ}{\sin D}$$

From the construction of angle A in triangle ACF:

$$A = 180^\circ - 30^\circ - 45^\circ$$

Triangles ACF and ACD are similar (have the same angles), which means that the lengths of corresponding sides are proportional:

$$\frac{AC}{CF} = \frac{CD}{AC}$$

Solving for CF:

$$CF = \frac{(AC)^2}{CD}$$

From the similar triangle, angle F must be the same as angle CAD, so $F = 30^\circ$.

Applying the Law of sines to triangle ACF:

$$\frac{AF}{\sin 45^\circ} = \frac{AC}{\sin F}$$

Solving for AF:

$$AF = AC\frac{\sin 45^\circ}{\sin F}$$

From the included angles in triangle AEF:

$$E = 180^\circ - 15^\circ - F$$

Applying the Law of sines to triangle AEF:

$$\frac{EF}{\sin 15^\circ} = \frac{AF}{\sin E}$$

Solving for EF:

$$EF = AF \frac{\sin 15^\circ}{\sin E}$$

Finally, the distance DE across the lake is:

$$DE = CF - CD - EF$$

A MATLAB script to compute DE, with angles in degrees and distances in feet:

```
% Lake distance problem
%
AC = 245/sin(30*pi/180)
D = 180-30-45
CD = AC*sin(30*pi/180)/sin(D*pi/180)
CF = AC^2/CD
F = 30
AF = AC*sin(45*pi/180)/sin(F*pi/180)
E = 180-15-F
EF = AF*sin(15*pi/180)/sin(E*pi/180)
DE = CF - CD - EF
```

The displayed results from MATLAB:

```
AC =
  490.0000
D =
   105
CD =
  253.6427
CF =
  946.6073
F =
   30
```

```
AF =
  692.9646
E =
   135
EF =
  253.6427
DE =
  439.3220
```

The final step in the solution to this problem is to carefully consider the results to determine if they make sense. This is done visually by comparing the computed lengths with those of Fig. 4.2.

∎

4.1.1 Hyperbolic Functions

The **hyperbolic functions** are functions of the natural exponential function e^x, where e is the base of the natural logarithms, which is approximately $e = 2.71828182845904$. The inverse hyperbolic functions are functions of the natural logarithm function, $\ln x$.

The curve $y = \cosh x$ is called a *catenary* (from the Latin word meaning "chain"). A chain or rope, suspended from its ends, forms a curve that is part of a catenary.

Matlab includes several hyperbolic functions, as described briefly in the table below.

`sinh(x)`	Hyperbolic sine of `x`; $\frac{1}{2}(e^x - e^{-x})$
`cosh(x)`	Hyperbolic cosine of `x`; $\frac{1}{2}(e^x + e^{-x})$
`tanh(x)`	Hyperbolic tangent of `x`; $\frac{\sinh(x)}{\cosh(x)}$
`asinh(x)`	Inverse hyperbolic sine of `x`; $\ln(x + \sqrt{x^2+1})$
`acosh(x)`	Inverse hyperbolic cosine of `x`; $\ln(x + \sqrt{x^2-1})$ for $x \geq 1$
`atanh(x)`	Inverse hyperbolic tangent of `x`; $\frac{1}{2}\ln\left(\frac{1+x}{1-x}\right)$ for $\lvert x\rvert \leq 1$.

4.2 Complex Numbers

Complex numbers find widespread applications in many fields. They are used throughout mathematics, applied science, and engineering to represent the harmonic nature of vibrating systems and oscillating fields.

A powerful feature of Matlab is that it does not require any special handling for complex numbers. In this section, we develop the algebra and geometry of complex numbers and describe how they are represented and handled by Matlab.

4.2.1 Definitions and Geometry

Imaginary number: The most fundamental new concept in the study of complex numbers is the "imaginary number" j. This imaginary number is defined to be the square root of -1:

$$j = \sqrt{-1}$$

$$j^2 = -1$$

You may be more familiar with the imaginary number being denoted by i, which is the common notation in mathematics. However, in engineering, an electrical current is denoted by i, so j is used for the imaginary number.

Rectangular Representation: A complex number z consists of the "real part" x and the "imaginary part" y and is expressed as

$$z = x + jy$$

where

$$x = \mathrm{Re}[z]; \quad y = \mathrm{Im}[z]$$

This form of representation for complex numbers is called the rectangular or cartesian form since z can be represented in rectangular coordinates by the point (x, y) in a plane having a horizontal axis being the "real axis" and the vertical axis being the "imaginary axis," as shown in Figure 4.3. This plane is called the "complex plane."

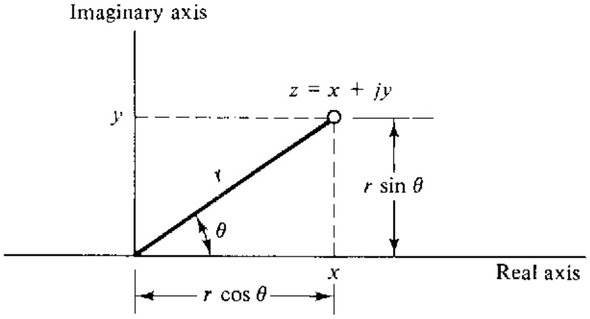


Figure 4.3: The complex number z in the complex plane

In Matlab, `i` and `j` are variable names that default to the imaginary number. You have to be careful with their use, however, as they can be overridden and used as general variables. You can insure that `j` is the imaginary number by explicitly computing it as the square root of -1:

```
>> j = sqrt(-1)
j =
        0+ 1.0000i
```

The result is displayed in rectangular form, with `i` used as the imaginary number.

A general complex number can be formed in three ways:

```
>> z = 1 + j*2
z =
   1.0000+ 2.0000i
>> z = 1 + 2j
z =
   1.0000+ 2.0000i
>> z = complex(1,2)
z =
   1.0000 + 2.0000i
```

In the first method above, the imaginary number `j` explicitly multiplies the real number (imaginary part) 2. In the second method, the imaginary number `j` is used as notation to produce an imaginary part of 2. Note however, that `j` cannot precede the imaginary part:

```
>> z = 1 + j2
??? Undefined function or variable 'j2'.
```

The error message indicates that MATLAB interprets `j2` as a variable that has not been defined and thus does not have a value.

In MATLAB, the function `real(z)` returns the real part and `imag(z)` returns the imaginary part:

```
>> z = 3 + 4j
z =
   3.0000+ 4.0000i
>> x = real(z)
x =
     3
>> y = imag(z)
y =
     4
```

We call $z = x + jy$ the Cartesian or rectangular representation of z, with real component x and imaginary component y. We say that the Cartesian pair (x, y) codes the complex number z.

Polar Representation: Defining the radius r and the angle θ of the complex number z shown in Figure 4.3, z can be represented in polar form and written as

$$z = r\cos\theta + jr\sin\theta$$

or, in shortened notation

$$z = r\angle\theta$$

where

	Math definition	MATLAB
Magnitude:	$\lvert z\rvert = r = \sqrt{x^2 + y^2}$	`abs(z)`
Argument or angle:	$\theta = \tan^{-1}\left(\frac{y}{x}\right) = \sin^{-1}\left(\frac{y}{r}\right) = \cos^{-1}\left(\frac{x}{r}\right)$	`angle(z)`

For example:

```
>> z = 3 + 4j;
>> r = abs(z)
r =
      5
>> theta = angle(z)
theta =
    0.9273
```

Recall that angles in MATLAB are given in radians. To compute the angle in degrees, the result in radians must be multiplied by $(360/2\pi)$ or $(180/\pi)$:

```
>> theta = (180/pi)*angle(z)
theta =
   53.1301
```

Principal value of the complex argument: The angle θ is defined only for nonzero complex numbers and is determined only up to integer multiples of 2π, since adding 2π radians rotates the complex number one revolution around the axis and leaves it in the same location. The value of θ that lies in the interval $-\pi < \theta \leq \pi$ is called the principal value of the argument of z, and is the value computed by `angle(z)` in MATLAB.

Example 4.3 *Principal value of complex argument*

For the complex number $z = 1 + j\sqrt{3}$

$$r = \sqrt{1^2 + (\sqrt{3})^2} = 2 \quad \text{and} \quad \theta = \angle z = \tan^{-1}\sqrt{3} = \frac{\pi}{3} + 2n\pi$$

The principal value of θ is $\pi/3$ and therefore

$$z = 2(\cos\pi/3 + j\sin\pi/3)$$

Confirming with MATLAB:

```
>> z = 1 + j*sqrt(3)
z =
   1.0000+ 1.7321i
>> theta = angle(z)
theta =
    1.0472
>> pi/3
ans =
    1.0472
>> r = abs(z)
r =
    2.0000
>> z1 = r*(cos(theta) + j*sin(theta))
z1 =
   1.0000+ 1.7321i
>> theta2 = theta +2*pi
theta2 =
    7.3304
>> z2 = r*(cos(theta2) + j*sin(theta2))
z2 =
   1.0000 + 1.7321i
```

■

Polar to rectangular conversion: To obtain the rectangular representation from the polar representation, apply the trigonometric relationships between the radius and angle and the real and imaginary parts:

$$x = r\cos\theta$$

$$y = r\sin\theta$$

For example:

```
>> r = 5; theta = 0.9273;
>> z = r*cos(theta) + j*r*sin(theta)
z =
   3.0000+ 4.0000i
```

The complex number z can be written as

$$\begin{aligned} z &= x + jy \\ &= r\cos\theta + jr\sin\theta \\ &= r(\cos\theta + j\sin\theta) \end{aligned}$$

Exponential Representation: The base of the natural logarithms, $e = 2.71828182845904$, is used to develop the exponential representation for complex numbers, through the **Euler** (sounds like oiler) formula

$$e^{j\theta} = \cos\theta + j\sin\theta$$

From this identity, two additional formulas can be derived:

$$\cos\theta = \frac{1}{2}\left(e^{j\theta} + e^{-j\theta}\right)$$

$$\sin\theta = \frac{1}{2j}\left(e^{j\theta} - e^{-j\theta}\right)$$

These formulas are derived and discussed in greater detail in a calculus course. Our purpose here is to use them to represent the complex number z in the exponential form

$$\begin{aligned} z = re^{j\theta} &= r\cos\theta + jr\sin\theta \\ &= r(\cos\theta + j\sin\theta) \end{aligned}$$

Note that this has the same functional form as the polar representation for z. While it appears to differ from the polar representation only in notation at this point, we will continue to expand on the properties of the exponential representation to show that the differences are more than symbolic. The graphical representation shown in Figure 4.3 still applies.

Magnitude of z: $|z| = r$

Angle or **phase** of z: $\theta = \angle z$

Exponential representation of z:

$$z = |z|e^{j\angle z} = re^{j\theta}$$

Confirming with Matlab:

```
>> z = 3 + 4j
z =
   3.0000+ 4.0000i
>> r = abs(z)
r =
     5
>> theta = angle(z)
theta =
    0.9273
>> z = r * exp(j*theta)
z =
   3.0000+ 4.0000i
```

Consider the special case for which the magnitude $|z| = r = 1$

$$z = e^{j\theta}$$

For this case, z lies on on a circle of radius 1 centered on the origin of the complex plane, at angle θ, as shown in Figure 4.4.

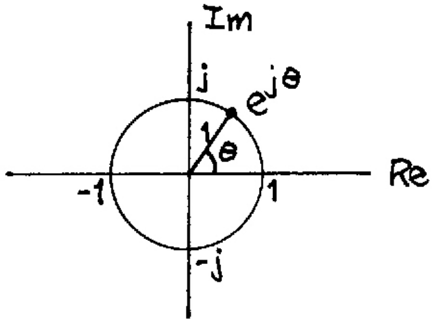


Figure 4.4: Complex number $z = e^{j\theta}$ in the complex plane

There are several values of θ for which you should know the value of $e^{j\theta}$, as shown in Figure 4.4.

$\theta = 0$: $e^{j0} = \cos 0 + j \sin 0 = 1 + j0 = 1$
$\theta = \pi/2$: $e^{j\pi/2} = \cos \pi/2 + j \sin \pi/2 = 0 + j1 = j$
$\theta = \pi$: $e^{j\pi} = \cos \pi + j \sin \pi = -1 + j0 = -1$
$\theta = 2\pi$: $e^{j2\pi} = \cos 2\pi + j \sin 2\pi = 1 + j0 = 1$
$\theta = -\pi/2$: $e^{j(-\pi/2)} = \cos(-\pi/2) + j \sin(-\pi/2) = 0 - j1 = -j$

Using the function `exp(x)`, these values can be confirmed in Matlab:

```
>> z = exp(j*0)
z =
     1
>> z = exp(j*pi/2)
z =
   0.0000+ 1.0000i
>> z = exp(j*pi)
z =
  -1.0000+ 0.0000i
>> z = exp(j*2*pi)
z =
   1.0000- 0.0000i
>> z = exp(-j*pi/2)
z =
   0.0000- 1.0000i
```

Summary: Complex Number Representations

We can summarize the representations of the complex number z as follows:

$$z = x + jy = r\angle\theta = re^{j\theta} = |z|e^{j\angle z}$$

4.2.2 Algebra of Complex Numbers

The algebraic operations of addition, subtraction, multiplication, and division can be defined for complex numbers in much the same way as they are defined for real numbers. Also, additional algebraic operations can be defined for complex numbers that have no meaning for real numbers. The complex operations have simple geometric interpretations.

Addition and Subtraction: The complex numbers z_1 and z_2 are added (or subtracted) by separately adding (or subtracting) the real and imaginary parts:

$$\begin{aligned} z_1 + z_2 &= (x_1 + jy_1) + (x_2 + jy_2) \\ &= (x_1 + x_2) + j(y_1 + y_2) \end{aligned}$$

$$\begin{aligned} z_1 - z_2 &= (x_1 + jy_1) - (x_2 + jy_2) \\ &= (x_1 - x_2) + j(y_1 - y_2) \end{aligned}$$

As shown in Figure 4.5, the geometric interpretation of complex addition is the "parallelogram rule," where $z_1 + z_2$ lies on the node of a parallelogram formed from z_1 and z_2. For complex subtraction, $-z_2$ is represented in the complex plane by reversing its direction and then adding to z_1 using the parallelogram rule, as shown in Figure 4.5. If z is given in polar or complex exponential form, it must be converted to rectangular form to perform the addition.

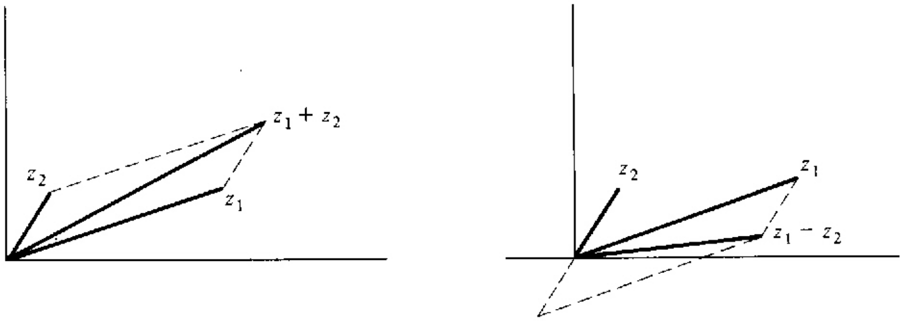


Figure 4.5: Addition and subtraction of complex numbers

In Matlab, complex addition is performed in the same way as it is performed for real numbers:

```
>> z1 = 1 + 2j
```

```
z1 =
   1.0000+ 2.0000i
>> z2 = 4 + 3j
z2 =
   4.0000+ 3.0000i
>> z3 = z1 + z2
z3 =
   5.0000+ 5.0000i
```

Multiplication: The product of z_1 and z_2 is found using the "first-outer-inner-last (FOIL)" method from polynomial multiplication, applying the identity $j^2 = -1$, and writing the result in rectangular form:

$$\begin{aligned} z_1 z_2 &= (x_1 + jy_1)(x_2 + jy_2) \\ &= x_1 x_2 + jx_1 y_2 + jx_2 y_1 + j^2 y_1 y_2 \\ &= (x_1 x_2 - y_1 y_2) + j(x_1 y_2 + x_2 y_1) \end{aligned}$$

Multiplication is better understood if the complex exponential representations are used:

$$\begin{aligned} z_1 z_2 &= r_1 e^{j\theta_1} r_2 e^{j\theta_2} \\ &= r_1 r_2 e^{j(\theta_1 + \theta_2)} \end{aligned}$$

Here, we have used the mathematical property that exponents $j\theta_1$ and $j\theta_2$ of common base e add. We say from the above that the magnitudes multiply and the angles add.

$$|z_1 z_2| = r_1 r_2$$

$$\angle(z_1 z_2) = \theta_1 + \theta_2$$

In Matlab, complex multiplication is performed in the same way as it is performed for real numbers. In the example below, magnitudes and angles have been computed to allow you to confirm that magnitudes multiply and angles add:

```
>> z1 = 1 +0.5j; z2 = 2 + 1.5j;
>> z4 = z1 * z2
z4 =
   1.2500+ 2.5000i
>> magz1 = abs(z1)
magz1 =
    1.1180
>> magz2 = abs(z2)
magz2 =
    2.5000
```

```
>> magz4 = abs(z4)
magz4 =
    2.7951
>> mag_test = magz1 * magz2
mag_test =
    2.7951
>> angz1 = angle(z1)
angz1 =
    0.4636
>> angz2 = angle(z2)
angz2 =
    0.6435
>> angz4 = angle(z4)
angz4 =
    1.1071
>> ang_test = angz1 + angz2
ang_test =
    1.1071
```

Rotation: There is a special case of complex multiplication that is important to understand. When $z_1 = r_1 e^{j\theta_1}$ and $z_2 = e^{j\theta_2}$ (i.e., the magnitude of z_2 is 1), then the product of z_1 and z_2 is

$$\begin{aligned} z_1 z_2 &= r_1 e^{j\theta_1} e^{j\theta_2} \\ &= r_1 e^{j(\theta_1+\theta_2)} \end{aligned}$$

As shown in Figure 4.6, $z_1 z_2$ is just a rotation of z_1 through the angle θ_2 . A particular case of rotation results from the multiplication by j. Recalling that $e^{j\pi/2} = j$, the product jz_1 becomes:

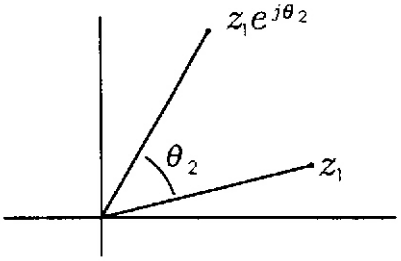


Figure 4.6: Rotation of complex numbers

$$jz_1 = r_1 e^{j(\theta_1+\pi/2)}$$

Thus, multiplying by j results in a rotation by $\pi/2$ or 90°, producing a result that is perpendicular to z_1 in the complex plane.

For example:

```
>> z1 = 3 + 4j;
>> z2 = z1 * exp(j*pi/2)
z2 =
  -4.0000+ 3.0000i
>> z3 = j * z1
z3 =
  -4.0000+ 3.0000i
>> theta1 = (180/pi) * angle(z1)
theta1 =
   53.1301
>> theta2 = (180/pi) * angle(z2)
theta2 =
  143.1301
>> theta_diff = theta2 - theta1
theta_diff =
    90
```

Complex Conjugate. For every complex number $z = x+jy = re^{j\theta}$ there is the complex conjugate

$$z^* = x - jy$$

Plotting z^* in the complex plane, by replacing y with $-y$, as shown in Figure 4.7, we see that the angle or phase of z^* is $-\theta$. Thus,

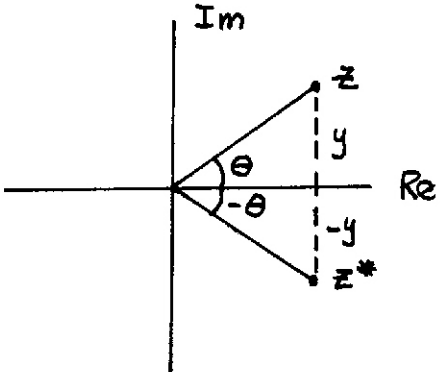


Figure 4.7: Complex conjugate

$$z^* = x - jy = re^{-j\theta}$$

The mathematical procedure for finding a complex conjugate is to "replace j with $-j$," changing the sign of the imaginary part of the complex number.

For example:

```
>> z = 3 + 4j
z =
   3.0000+ 4.0000i
>> zconj = conj(z)
zconj =
   3.0000- 4.0000i
>> theta = angle(z)
theta =
    0.9273
>> r = abs(z)
r =
     5
>> zconj1 = r * exp(-j*theta)
zconj1 =
   3.0000- 4.0000i
```

Magnitude Squared: The product of z and its complex conjugate is

$$z^*z = (x - jy)(x + jy) = x^2 + jxy - jxy - j^2y^2 = x^2 + y^2 = r^2 = |z|^2$$

or,

$$|z|^2 = z^*z$$

and we see that the magnitude squared $|z|^2$ is the product of z and its complex conjugate z^* .

For example:

```
>> z = 3 + 4j;
>> zz = conj(z) * z
zz =
    25
>> zmsq = abs(z)^2
zmsq =
    25
```

Division: This operation is defined as the inverse operation of multiplication. The quotient z_1/z_2 is obtained in rectangular form by multiplying both the numerator and denominator of the quotient by the conjugate of z_2:

$$\begin{aligned}
\frac{z_1}{z_2} &= \frac{x_1 + jy_1}{x_2 + jy_2} \\
&= \frac{(x_1 + jy_1)(x_2 - jy_2)}{(x_2 + jy_2)(x_2 - jy_2)} \\
&= \frac{(x_1 + jy_1)(x_2 - jy_2)}{x_2^2 + y_2^2}
\end{aligned}$$

$$= \frac{x_1x_2 + y_1y_2}{x_2^2 + y_2^2} + j\frac{x_2y_1 - x_1y_2}{x_2^2 + y_2^2}$$

Division is more easily performed in exponential form

$$\begin{aligned}
\frac{z_1}{z_2} &= \frac{r_1e^{j\theta_1}}{r_2e^{j\theta_2}} \\
&= \frac{r_1e^{j\theta_1}e^{-j\theta_2}}{r_2e^{j\theta_2}e^{-j\theta_2}} \\
&= \frac{r_1}{r_2}e^{j(\theta_1-\theta_2)}
\end{aligned}$$

Thus, the magnitude of the quotient is the quotient of the magnitudes and the angle of the quotient is the difference of the angle of the numerator and the angle of the denominator:

$$\left|\frac{z_1}{z_2}\right| = \frac{r_1}{r_2}$$

$$\angle\left(\frac{z_1}{z_2}\right) = \theta_1 - \theta_2$$

For example:

```
>> z2 = 2 + 1.5j
z2 =
   2.0000+ 1.5000i
>> z4 = 1.25 + 2.5j
z4 =
   1.2500+ 2.5000i
>> z1 = z4/z2
z1 =
   1.0000+ 0.5000i
>> magz2 = abs(z2)
magz2 =
    2.5000
>> magz4 = abs(z4)
magz4 =
    2.7951
>> magz1 = abs(z1)
magz1 =
    1.1180
>> magtest = magz4/magz2
magtest =
    1.1180
>> argz2 = angle(z2)
argz2 =
    0.6435
>> argz4 = angle(z4)
argz4 =
    1.1071
>> argz1 = angle(z1)
argz1 =
    0.4636
>> ang_test = argz4 - argz2
ang_test =
    0.4636
```

Algebraic Rules

The algebraic rules you have learned for real variables and numbers apply to complex variables and numbers.

Commutativity: Addition and multiplication can be done in either order without changing the result.

$$z_1 + z_2 = z_2 + z_1$$

$$z_1 z_2 = z_2 z_1$$

Associativity: Sums and products of three or more variables can be performed in sequential groups of two without changing the result.

$$(z_1 + z_2) + z_3 = z_1 + (z_2 + z_3)$$

$$(z_1 z_2) z_3 = z_1 (z_2 z_3)$$

Distributivity: Multiplication can be distributed across a sum without changing the result.

$$z_1(z_2 + z_3) = z_1 z_2 + z_1 z_3$$

Matlab Functions for Complex Numbers

To summarize, the following are the MATLAB functions for complex numbers:

`abs(z)`	Complex magnitude $\|z\|$ (absolute value for real z)
`angle(z)`	Phase angle or argument of z
`conj(z)`	Complex conjugate z^*
`imag(z)`	Complex imaginary part $\mathrm{Im}(z)$
`real(z)`	Complex real part $\mathrm{Re}(z)$

4.2.3 Roots of a Quadratic Polynomial

In a previous example, we found that the roots of the quadratic polynomial

$$as^2 + bs + c = 0$$

are given by:

$$s_{1,2} = -\frac{b}{2a} \pm \frac{1}{2a}\sqrt{b^2 - 4ac}$$

For the values of the coefficients considered in that example, the resulting roots were real. However, for other values of the coefficients, the roots can be complex. Having now reviewed complex numbers, we can investigate the problem of quadratic roots in more detail. There are three different cases for the solution, dependent on the value of the **discriminant** of the quadratic equation:

$$d = b^2 - 4ac$$

- Overdamped ($d > 0$): Both roots are real and are given by

  $$s_{1,2} = -\frac{b}{2a} \pm \frac{1}{2a}\sqrt{b^2 - 4ac}$$

  The roots are located symmetrically about the point $-b/2a$. When $b = 0$, they are located symmetrically about 0 at the points $\pm(1/2a)\sqrt{-4ac}$ (in this case, $-4ac > 0$).

- Critically Damped ($d = 0$): The two roots are real and equal (we say they are repeated):

  $$s_{1,2} = -\frac{b}{2a}$$

- Underdamped ($d < 0$): The square root in the quadratic equation produces an imaginary number, so the roots are complex

  $$\begin{aligned} s_{1,2} &= -\frac{b}{2a} \pm \frac{1}{2a}\sqrt{-(4ac - b^2)} \\ &= -\frac{b}{2a} \pm j\frac{1}{2a}\sqrt{4ac - b^2} \end{aligned}$$

  Note that in this case, the roots s_1 and s_2 are complex conjugates:

  $$s_2 = s_1^*$$

  The roots are purely imaginary when $b = 0$

  $$s_{1,2} = \pm j\sqrt{\frac{c}{a}}$$

  In MATLAB, it is not necessary to determine which of the three cases applies to a given problem, as the square root function will appropriately return real or imaginary values as needed.

Example:

```
>> a=1; b=4; c=8;
>> x = -b/(2*a);
>> y = sqrt(b^2-4*a*c)/(2*a);
>> s1 = x + y
s1 =
  -2.0000+ 2.0000i
>> s2 = x - y
s2 =
  -2.0000- 2.0000i
```

4.3 Two-Dimensional Plotting

One of the major advantages of Matlab are its capabilities for displaying its results in the form of two-dimensional and three-dimensional graphics, providing what has come to be known as scientific visualization. In this section, Matlab capabilities are explained for producing two-dimensional plots in the context of the display of complex variables in the complex plane. In later sections, these two-dimensional graphics will be extended to other variables, in the context of the presentation of those variable types. For more information, type `help graph2d`.

Example: Plotting Complex Variables

The plot command can be used to plot complex variables in the complex plane. For example:

```
>> z = 1 + 0.5j;
>> plot(z,'.')
```

This creates a graphics window, called a Figure window, named by default "Figure No. 1," which becomes the active (top) window. The plot produced is shown in Figure 4.8, with z plotted as a point (due to the command `'.'`) in the complex plane, with the real value (1.0) on the horizontal (x) axis and the imaginary value (0.5) on the vertical (y) axis. The axes have been scaled automatically, with a range of 0.0 to 2.0 on the horizontal axis and a range of -0.5 to 1.5 on the vertical axis. Numerical scales and tick marks have been added automatically.

This plot was exceedingly easy to produce, but it has many deficiencies that we can improve upon. Here are some improvements to consider:

- Control the scaling of the axes
- Produce a plot that is square instead of rectangular, having the same scale on both axes
- Include several complex variables on a single plot

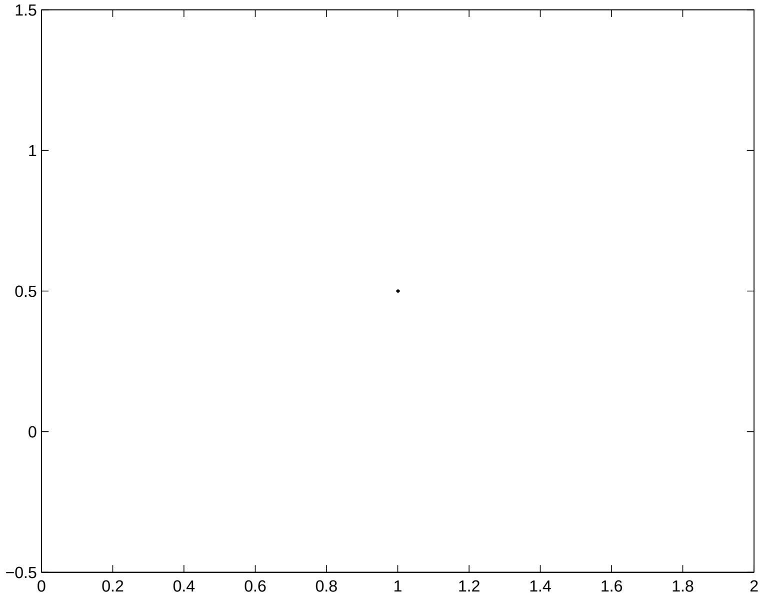


Figure 4.8: Simple plot of a complex number

- Label the axes
- Title the plot
- Label the plotted complex variables

The commands to implement each of these improvements will be introduced and discussed. The resulting plot produced by each command won't be given, so you might try each of them in Matlab as you read this. Finally, when all of the commands have been discussed, a Matlab example session using them will be given and the results will be displayed.

4.3.1 2D Plotting Commands

For more information, type `help graph2d`.

Colors and Markers

Color and markers can be specified by giving `plot` an additional argument following the complex variable name. This optional additional argument is a character string (enclosed in single quotes) consisting of characters from the following table:

Symbol	Color	Symbol	Marker
y	yellow	.	●
m	magenta	o	○
c	cyan	x	×
r	red	+	+
g	green	*	∗
b	blue	s	□
w	white	d	◇
k	black	v	▽
		ˆ	△
		<	◁
		>	▷
		p	⋆
		h	hexagram

Examples:

```
plot(z1,’b.’)      % plot variable z1 as a blue point
plot(z2,’go’)      % plot variable z2 as a green circle
plot(z3,’r*’)      % plot variable z3 as a red asterisk
```

Customizing Plot Axes

The `axis` command provides control over the scaling and appearance of both the horizontal and vertical axes of a plot. This command has many features, so only the most useful will be discussed here. For more complete information, refer to on-line help. The primary features are given in the following table

Command	Description
`axis([xmin xmax ymin ymax])`	Define minimum and maximum values of the axes
`axis square`	Produce a square plot instead of rectangular
`axis equal`	Equal scaling factors for both axes
`axis normal`	Turn off axis square, equal
`axis(auto)`	Return the axis to automatic defaults
`axis off`	Turn off axis background, labeling, grid, box, and tick marks. Leave the title and any labels placed by the `text` and `gtext` commands
`axis on`	Turn on axis background, labeling, tick marks, and, if they are enabled, box and grid

Adding New Curves

Using the `hold` command to add lines to an existing plot:

Command	Description
`hold on`	Retain existing axes, add new curves to current axes when new plot commands are issued. If the new data does not fit within the current axes limits, the axes are rescaled (for automatic scaling only)
`hold off`	Releases the current figure window for new plots
`ihold`	Logical command that returns 1 (True) if hold is on and 0 (False) if hold is off

Plot Grids, Axes Box, and Labels

There are several commands to control the appearance of the plot. These include:

Command	Description
`grid on`	Adds dashed grid lines at the tick marks
`grid off`	Removes grid lines (default)
`grid`	Toggles grid status (off to on, or on to off)
`box on`	Adds axes box, consisting of boundary lines and tick marks on top and right of plot
`box off`	Removes axes box (default)
`box`	Toggles box status
`title('text')`	Labels top of plot with text in quotes
`xlabel('text')`	Labels horizontal (x) axis with text in quotes
`ylabel('text')`	Labels vertical (y) axis with text in quotes
`text(x,y,'text')`	Adds text in quotes to location (x,y) on the current axes, where (x,y) is in units from the current plot
`gtext('text')`	Place text in quotes with mouse: displays the plot window, puts up a cross-hair to be positioned with the mouse, and write the text onto the plot at the selected position when the left mouse button or any keyboard key is pressed

Printing Figures and Saving Figure Files

Plots can be printed using a figure window menu bar selection or with MATLAB commands from the Command window.

To print a plot using commands from the menu bar, make the Figure window the active window by clicking it with the mouse. Then select the **Print** menu item from the **File** menu. Using the parameters set in the **Print Setup** or **Page Setup** menu item, the current plot is sent to the printer.

MATLAB has its own printing commands that can be executed from the Command window. To print a Figure window, click it with the mouse or use the `figure(n)` command, where `n` is the figure number, to make it active, and then execute the print command:

```
>> print   % prints the current plot to the system printer
```

The orient command changes the print orientation mode, as follows:

Command	Description
`orient portrait`	Prints vertically in middle of page (default)
`orient landscape`	Prints horizontally, stretches to fill the page
`orient tall`	Prints vertically, stretches to fill the page
`orient`	Displays the current orientation

Options to the `print` command provide for writing the plot to a file in many different formats. For example, to write the plot as a PostScript format file in the current directory:

`print -deps` *file*

where *file* is the name of the file to be written, usually having the file name extension .ps to remind you that it is a PostScript file.

To view the contents of a PostScript file, type the following in a UNIX command window:

`gv` *file*

where *file* is the name of the PostScript file (including the .ps extension).

To print a PostScript file, type the following in a UNIX command window:

`lp` *file*

Refer to on-line help for the `print` command for further information on other file formats and options.

4.3.2 Complex Plot Examples

Consider three examples to summarize the concepts of complex numbers, algebra, and geometry, plus the handling of complex numbers and plots by MATLAB.

Rectangular Form Plot

The MATLAB commands for a rectangular form plot:

```
z1 = 1.5 + 0.5j;
z2 = 0.5 + 1.5j;
z3 = z1 + z2;
z4 = z1 * z2;
z5 = conj(z1);
z6 = j * z2;
z7 = z2/z1;
axis([-3 3 -3 3]);
axis square;
grid on;
```

```
hold on;
plot(z1,’b.’);
plot(z2,’go’);
plot(z3,’rx’);
plot(z4,’m*’);
plot(z5,’c+’);
plot(z6,’kd’);
plot(z7,’kp’);
text(real(z1)+0.1,imag(z1),’z1’);
text(real(z2)+0.1,imag(z2),’z2’);
text(real(z3)+0.1,imag(z3),’z1+z2’);
text(real(z4)+0.1,imag(z4),’z1*z2’);
text(real(z5)+0.1,imag(z5),’z1*’);
text(real(z6)+0.1,imag(z6),’j*z2’);
text(real(z7)+0.1,imag(z7),’z2/z1’);
xlabel(’Real Part’);
ylabel(’Imaginary Part’);
title(’Complex Numbers’);
```

The plot that results is shown in Figure 4.9.

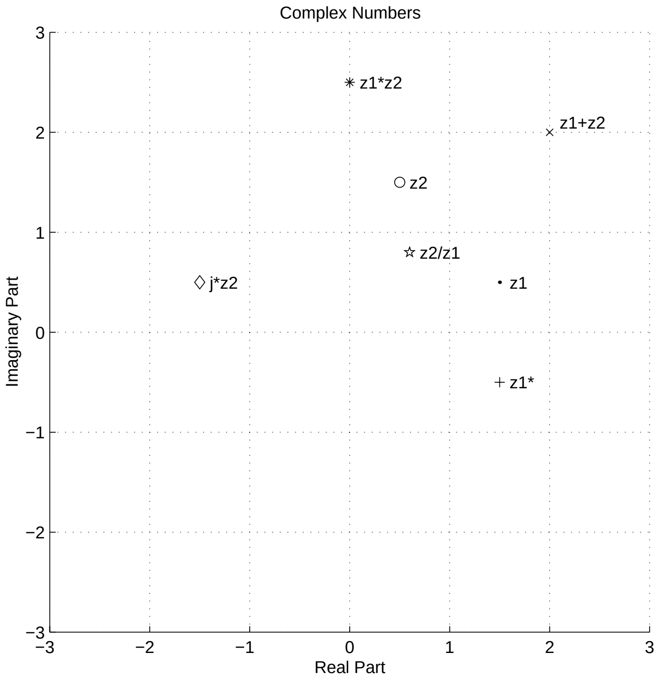


Figure 4.9: Complex number plot

Vector Form Plot

Complex numbers can also be represented graphically as arrows, with the base at the origin of the complex plane and the arrow head at the location of the complex number. This represents the complex number as a **vector**, which will be defined and described in greater detail later in the course.

Matlab supports the plotting of complex numbers as vectors through the `compass` command. Consider the following example, in which two complex numbers z_1 and z_2 are added to form z_3 and the three complex numbers are plotted using `compass`.

```
z1 = 1.5 + 0.5j;
z2 = 0.5 + 1.5j;
z3 = z1 + z2;
axis([-3 3 -3 3]);
axis square;
grid on;
hold on;
plot(z1,'b.');
plot(z2,'go');
plot(z3,'rx');
compass(z1);
compass(z2);
compass(z3);
text(real(z1)+0.1,imag(z1),'z1');
text(real(z2)+0.1,imag(z2),'z2');
text(real(z3)+0.1,imag(z3)+0.1,'z1+z2');
xlabel('Real Part');
ylabel('Imaginary Part');
title('Compass Plot');
```

The plot that results, shown in Figure 4.10, clearly shows the parallelogram property of the sum of two complex numbers.

Polar Form Plot

The magnitudes and angles of complex numbers can also be plotted in polar coordinates using the `polar` command. Consider the following example, in which two complex numbers z_1 and z_2 are multiplied to form z_3 and the three complex numbers are plotted using `polar`.

```
z1 = sqrt(3) + j;
z2 = 1.5*(1 + j*sqrt(3));
z3 = z1 * z2;
r1 = abs(z1);
phi1 = angle(z1);
r2 = abs(z2);
phi2 = angle(z2);
r3 = abs(z3);
phi3 = angle(z3);
```

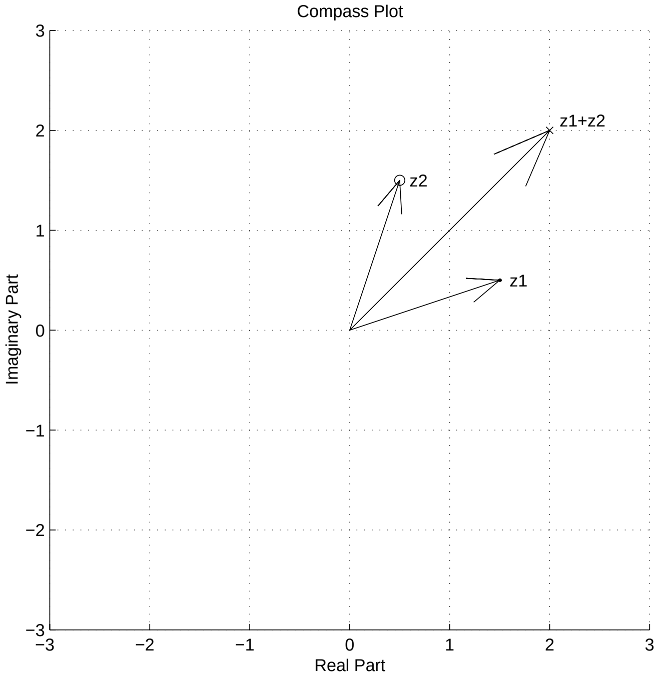


Figure 4.10: Compass plot of complex number addition

```
polar(phi3,r3,'*');
hold on
text(real(z1)+0.1,imag(z1),'z1')
polar(phi2,r2,'+')
text(real(z2)+0.1,imag(z2),'z2')
polar(phi1,r1,'o')
text(real(z3)+0.1,imag(z3)-0.2,'z1*z2')
```

The plot that results, shown in Figure 4.11, clearly shows that the magnitudes multiply and the angles add under complex multiplication. Note that while Matlab usually uses radian units for angles, degrees are used in this plot.

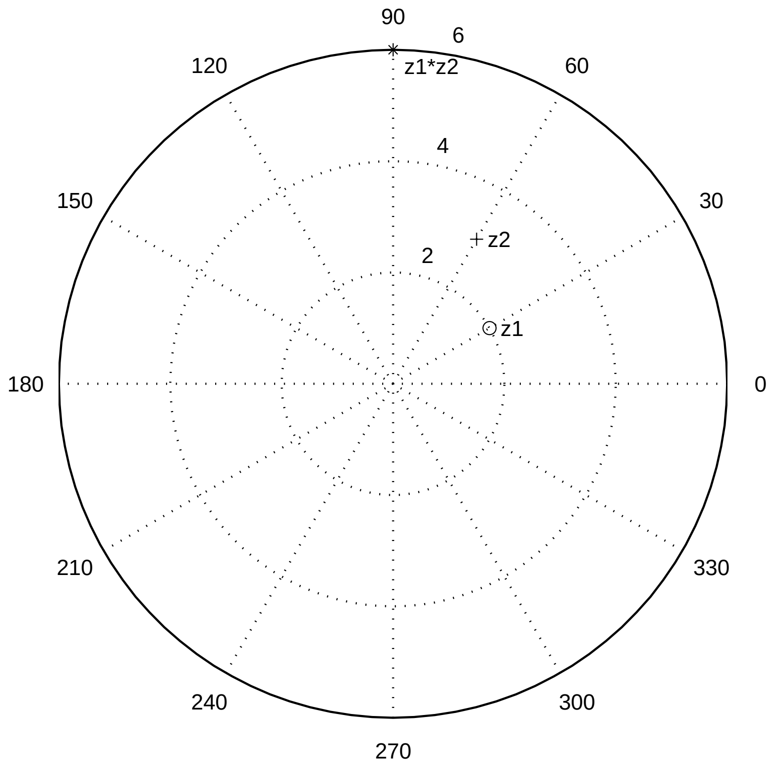


Figure 4.11: Polar plot of complex number multiplication

Section 5

Arrays and Array Operations

Scalars: Variables that represent single numbers, as considered to this point. Note that complex numbers are scalars, even though they have two components, the real part and the imaginary part.

Arrays: Variables that represent more than one number. Each number is called an **element** of the array. Rather than than performing the same operation on one number at a time, array operations allow operating on multiple numbers at once.

Row and Column Arrays: A row of numbers (called a row vector) or a column of numbers (called a column vector).

Two-Dimensional Arrays: A two-dimensional table of numbers, called a matrix.

Array Indexing or Addressing: Indicates the location of an element in the array.

5.1 Vector Arrays

First consider one-dimensional arrays: row vectors and column vectors.

Consider computing $y = \sin(x)$ for $0 \leq x \leq \pi$. It is impossible to compute y values for all values of x, since there are an infinite number of values, so we will choose a finite number of x values. Consider computing

$$y = \sin(x), \quad \text{where } \ x = 0, 0.1\pi, 0.2\pi, \ldots, \pi$$

You can form a list, or **array** of the values of x, and then using a calculator you can compute the corresponding values of y, forming a second array y. These can be written down as:

x	0	0.1 π	0.2 π	0.3 π	0.4 π	0.5 π	0.6 π	0.7 π	0.8 π	0.9 π	π
y	0	.31	.59	.81	.95	1.0	.95	.81	.59	.31	0

x and y are ordered lists of numbers, i.e., the first value or element of y is associated with the first value or element of x.

Elements can be denoted by subscripts, e.g. x_1 is the first element in x, y_5 is the fifth element in y. The subscript is the index, address, or location of the element in the array.

Vector Creation by Explicit List

A vector in Matlab can be created by an explicit list, starting with a left bracket, entering the values separated by spaces (or commas) and closing the vector with a right bracket.

```
>> x=[0 .1*pi .2*pi .3*pi .4*pi .5*pi .6*pi .7*pi .8*pi .9*pi pi]
x =
  Columns 1 through 7
         0    0.3142    0.6283    0.9425    1.2566    1.5708    1.8850
  Columns 8 through 11
    2.1991    2.5133    2.8274    3.1416
```

Matlab functions can be applied to vectors to compute a resulting vector:

```
>> y=sin(x)
y =
  Columns 1 through 7
         0    0.3090    0.5878    0.8090    0.9511    1.0000    0.9511
  Columns 8 through 11
    0.8090    0.5878    0.3090    0.0000
```

The two vectors produced above are row vectors, each containing one row and 11 columns. They are each also known as a **one-by-eleven array**, a 1 × 11 array, or a row vector of length 11.

Vector Addressing

A vector element is addressed in Matlab with an integer index (also called a *subscript*) enclosed in parentheses. For example, to access the third element of `x` and the fifth element of `y`:

```
>> x(3)
ans =
    0.6283
>> y(5)
ans =
    0.9511
```

Colon notation: Addresses a block of elements

The format for colon notation is:

```
(start:increment:end)
```

where start is the starting index, increment is the amount to add to each successive index, and end is the ending index, where start, increment, and end must be integers. The increment can be negative, but negative indices are not allowed to be generated. If the increment is to be 1, a shortened form of the notation may be used:

```
(start:end)
```

- 1:5 means "start with 1 and count up to 5."

  ```
  >> x(1:5)
  ans =
           0    0.3142    0.6283    0.9425    1.2566
  ```

- 7:end means "start with 7 and count up to the end of the vector."

  ```
  >> x(7:end)
  ans =
      1.8850    2.1991    2.5133    2.8274    3.1416
  ```

- 3:-1:1 means "start with 3 and count down to 1."

  ```
  >> y(3:-1:1)
  ans =
      0.5878    0.3090         0
  ```

  Thus, a negative index can be used, but the indices generated must all be positive integers.

- 2:2:7 means "start with 2, count up by 2, and stop at 7."

  ```
  >> x(2:2:7)
  ans =
      0.3142    0.9425    1.5708
  ```

- 3:1 means "start with 3, count up by 1, and stop at 1." This clearly doesn't make sense, so Matlab generates the error message:

  ```
  >> x(3:1)
  ans =
     Empty matrix: 1-by-0
  ```

A vector can be used to access elements of an vector in a desired order. To access elements 8, 2, 9, and 1 of y in order:

```
>> y([8 2 9 1])
ans =
    0.8090    0.3090    0.5878         0
```

Vector Creation Alternatives

Explicit lists are often cumbersome ways to enter vector values.

Alternatives:

- **Combining**: A vector can also be defined using another vector that has already been defined. For example:

  ```
  >> B = [1.5, 3.1];
  >> S = [3.0 B]
  S =
      3.0000    1.5000    3.1000
  ```

- **Changing**: Values can be changed by referencing a specific address. For example,

  ```
  >> S(2) = -1.0;
  >> S
  S =
      3.0000   -1.0000    3.1000
  ```

  changes the second value in the vector `S` from 1.5 to -1.0.

- **Extending**: Additional values can be added using a reference to a specific address. For example, the following command:

  ```
  >> S(4) = 5.5;
  >> S
  S =
      3.0000   -1.0000    3.1000    5.5000
  ```

  extends the length of vector `S` from 3 to 4.

  Applying the following command

  ```
  >> S(7) = 8.5;
  >> S
  S =
      3.0000   -1.0000    3.1000    5.5000         0         0    8.5000
  ```

  extends the length of `S` to 7, and the values of `S(5)` and `S(6)` are automatically set to zero because no values were given for them.

- **Colon notation:** A generalized version of the colon notation used to index array elements can be used to generate array elements. The format for this notation is:

  ```
  (start:increment:end)
  ```

  where `start`, `increment`, and `end` can now be floating point numbers and there is no restriction that the values generated must be positive.

  Thus, to create `x` in the example above:

```
>> x=(0:0.1:1)*pi
x =
  Columns 1 through 7
         0    0.3142    0.6283    0.9425    1.2566    1.5708    1.8850
  Columns 8 through 11
    2.1991    2.5133    2.8274    3.1416
```

As in addressing, (0:5) creates a vector that starts at 0, increments by 1 (implied), and ends at 5.

```
>> x = 0:5
x =
     0     1     2     3     4     5
```

A particular choice of `start` and `increment` values may not lead to the generation of a last value equal to `end`. The last value generated must be less than or equal to `end`. For example:

```
>> x = 0:2:9
x =
     0     2     4     6     8
```

In this case, the next value to be generated would be 10, but this is not included in the output since it would be greater than 9, the value of `end`.

- `linspace`: This function generates a vector of uniformly incremented values, but instead of specifying the increment, the number of values desired is specified. This function has the form:

  ```
  linspace(start,end,number)
  ```

  The increment is computed internally, having the value:

  $$\texttt{increment} = \frac{\texttt{end} - \texttt{start}}{\texttt{number} - 1}$$

  For example:

  ```
  >> x=linspace(0,pi,11)
  x =
    Columns 1 through 7
           0    0.3142    0.6283    0.9425    1.2566    1.5708    1.8850
    Columns 8 through 11
      2.1991    2.5133    2.8274    3.1416
  ```

  In this example:

  $$\texttt{increment} = \frac{\pi}{11 - 1} = 0.1\pi$$

- `logspace`: This function generates logarithmically spaced values and has the form

```
logspace(start_exponent,end_exponent,number)
```

To create a vector starting at $10^0 = 1$, ending at $10^2 = 100$ and having 11 values:

```
>> logspace(0,2,11)
ans =
  Columns 1 through 7
    1.0000    1.5849    2.5119    3.9811    6.3096   10.0000   15.8489
  Columns 8 through 11
   25.1189   39.8107   63.0957  100.0000
```

Vector Length

To determine the length of a vector array:

`length(x)`: Returns the length of vector `x`.

Example:

```
>> x = [0 1 2 3 4 5]
x =
     0     1     2     3     4     5
>> length(x)
ans =
     6
```

Summary of Vector Addressing, Creation, and Length

Command	Description
`x=[2 2*pi sqrt(2) 2-3j]`	Create row vector `x` containing elements specified
`x=start:end`	Create row vector `x` starting with `start`, counting by one, ending at or before `end`
`x=start:increment:end`	create row vector `x` starting with `start`, counting by `increment`, ending at or before `end`
`linspace(start,end,number)`	Create row vector `x` starting with `start`, ending at `end`, having `number` elements
`logspace(start_exp,end_exp,number)`	Create row vector `x` starting with `10^(start_exp)`, ending at `10^(end_exp)`, having `number` elements
`length(x)`	Returns the length of vector `x`.

Vector Orientation

In the preceding examples, vectors contained one row and multiple columns and were therefore called row vectors.

A column vector, having one column and multiple rows, can be created by specifying it element by element, separating element values with semicolons:

```
>> c = [1;2;3;4;5]
c =
     1
     2
     3
     4
     5
```

Thus, separating elements by spaces or commas specifies elements in different columns, whereas separating elements by semicolons specifies elements in different rows.

The **transpose** operator (’) is used to transpose a row vector into a column vector

```
>> a = 1:5
a =
     1     2     3     4     5
>> b = a’
b =
     1
     2
     3
     4
     5
```

For a complex vector, the transpose operator (’) produces the complex conjugate transpose (i.e. the sign on the imaginary part is changed as part of the transpose operation). The **dot-transpose** operator (.’) transposes the vector but does not conjugate it. The two transpose operators are identical for real data.

```
>> a = [1+j 2-3j -3+4j]
a =
   1.0000+ 1.0000i   2.0000- 3.0000i  -3.0000+ 4.0000i
>> b = a’
b =
   1.0000- 1.0000i
   2.0000+ 3.0000i
  -3.0000- 4.0000i
>> c = a.’
c =
   1.0000+ 1.0000i
   2.0000- 3.0000i
  -3.0000+ 4.0000i
```

Vectors containing zeros and ones

Two special functions in MATLAB can be used to generate new vectors containing all zeros or all ones.

`zeros(m,1)`	Returns an `m`-element column vector of zeros
`zeros(1,n)`	Returns an `n`-element row vector of zeros
`ones(m,1)`	Returns an `m`-element column vector of ones
`ones(1,n)`	Returns an `n`-element row vector of ones

Examples:

```
>> x = zeros(3,1)
x =
     0
     0
     0
>> y = ones(1,3)
y =
     1     1     1
```

5.2 Matrix Arrays

A matrix array is two-dimensional, having both multiple rows and multiple columns. Creation of two-dimensional arrays follows that of row and column vectors:

- Begin with [, end with]
- Spaces or commas are used to separate elements in a row.
- A semicolon or Enter is used to separate rows.

```
>> f = [1 2 3; 4 5 6]
f =
     1     2     3
     4     5     6
>> g = f'
g =
     1     4
     2     5
     3     6
>> h = [1 2 3
4 5 6
7 8 9]
h =
     1     2     3
     4     5     6
     7     8     9
```

```
>> k = [1 2;3 4 5]
??? Number of elements in each row must be the same.
```

Here `f` is an array having 2 rows and 3 columns, called a 2 by 3 matrix. Applying the transpose operator to `f` produces `g`, a 3 by 2 matrix. Array `h` is 3 by 3, created using $\boxed{\text{Enter}}$ to start new rows. The incorrect attempt to construct `k` shows that all rows must contain the same number of columns.

The transpose operator can be used to generate tables from vectors. For example, generate two vectors, `x` and `y`, then display the values such that `x(1)` and `y(1)` are on the same line, and so on, as follows:

```
>> x = 0:4;
>> y = 5:5:25;
>> A = [x' y']
A =
     0     5
     1    10
     2    15
     3    20
     4    25
```

Note that the matrix `A` has been created from the vectors `x` and `y`.

Matrix Addressing

To refer to the element in row 2 and column 3 of matrix `f`:

```
>> f(2,3)
ans =
     6
```

Thus, the first number in parentheses addresses the row and the second number (following a comma) addresses the column.

Colon in place of a row or column reference represents the entire row or column. To create a column vector from the first column of the array `h` above:

```
>> h_1 = h(:,1)
h_1 =
     1
     4
     7
```

Submatrix creation: To create a submatrix using the colon operator, consider:

```
>> c = [1,0,0; 1,1,0; 1,-1,0; 0,0,2]
c =
     1     0     0
     1     1     0
     1    -1     0
     0     0     2
>> c_1 = c(:,2:3)
c_1 =
     0     0
     1     0
    -1     0
     0     2
>> c_2 = c(3:4,1:2)
c_2 =
     1    -1
     0     0
```

Thus, `c_1` contains all rows and columns 2 and 3 of `c` and `c_2` contains rows 3 and 4 of columns 1 and 2 of `c`.

Matrix size

To determine the size of a matrix array:

Command	Description
`s = size(A)`	For an `m` × `n` matrix `A`, returns the two-element row vector `s = [m, n]` containing the number of rows and columns in the matrix.
`[r,c] = size(A)`	Returns two scalars `r` and `c` containing the number of rows and columns in `A`, respectively.
`r = size(A,1)`	Returns the number of rows in `A` in the variable `r`.
`c = size(A,2)`	Returns the number of columns in `A` in the variable `c`.
`n=length(A)`	Returns the larger of the number of rows or columns in `A` in the variable `n`.
`whos`	Lists variables in the current workspace, together with information about their size, bytes, class, etc. (Long form of `who`.

Examples:

```
>> A = [1 2 3; 4 5 6]
A =
     1     2     3
     4     5     6
>> s = size(A)
s =
     2     3
>> [r,c] = size(A)
r =
     2
c =
     3
>> r = size(A,1)
r =
     2
>> c = size(A,2)
c =
     3
>> length(A)
ans =
     3
>> whos
  Name      Size         Bytes  Class

  A         2x3             48  double array
  ans       1x1              8  double array
  c         1x1              8  double array
  r         1x1              8  double array
  s         1x2             16  double array

Grand total is 11 elements using 88 bytes
```

Matrices containing zeros and ones

Two special functions in MATLAB can be used to generate new matrices containing all zeros or all ones.

`zeros(n)`	Returns an $n \times n$ array of zeros.
`zeros(m,n)`	Returns an $m \times n$ array of zeros.
`zeros(size(A))`	Returns an array the same size as `A` containing all zeros.
`ones(n)`	Returns an $n \times n$ array of ones.
`ones(m,n)`	Returns an $m \times n$ array of ones.
`ones(size(A)`	Returns an array the same size as `A` containing all ones.

Examples of the use of `zeros`:

```
>> A = zeros(3)
A =
     0     0     0
     0     0     0
     0     0     0
>> B = zeros(3,2)
B =
     0     0
     0     0
     0     0
>> C =[1 2 3; 4 2 5]
C =
     1     2     3
     4     2     5
>> D = zeros(size(C))
D =
     0     0     0
     0     0     0
```

5.2.1 Array Operations

Scalar-Array Mathematics

Addition, subtraction, multiplication, and division of an array by a scalar simply apply the operation to all elements of the array.

```
>> f = [1 2 3; 4 5 6]
f =
     1     2     3
     4     5     6
>> g = 2*f -1
g =
     1     3     5
     7     9    11
```

Element-by-Element Array-Array Mathematics

When two arrays have the same dimensions, addition, subtraction, multiplication, and division apply on an element-by-element basis. The notation for some operations is somewhat unconventional.

Operation	Algebraic Form	Matlab
Addition	$a + b$	`a + b`
Subtraction	$a - b$	`a - b`
Multiplication	$a \times b$	`a.*b`
Division	$a \div b$	`a./b`
Exponentiation	a^b	`a.^b`

Examples:

```
>> A = [2 5 6];
>> B = [2 3 5];
>> C = A.*B
C =
     4    15    30
>> D = A./B
D =
    1.0000    1.6667    1.2000
>> E = A.^B
E =
           4         125        7776
```

```
>> F = 3.0.^A
F =
     9   243   729
```

Array-array mathematics on other than an element-by-element basis will be discussed later.

5.3 Array Plotting Capabilities

The `plot` command introduced previously can be used to produce an *xy* plot of vector `x` against vector `y`, as linear plots with the *x* and *y* axes divided into equally spaced intervals. Other plots use a logarithmic scale (base 10) when a variable ranges over many orders of magnitude because the wide range of values can be graphed without compressing the smaller values.

`plot(x,y)` Generates a linear plot of the values of `x` (horizontal axis) and `y` (vertical axis).

`semilogx(x,y)` Generates a plot of the values of `x` and `y` using a logarithmic scale for `x` and a linear scale for `y`.

`semilogy(x,y)` Generates a plot of the values of `x` and `y` using a linear scale for `x` and a logarithmic scale for `y`.

`loglog(x,y)` Generates a plot of the values of `x` and `y` using logarithmic scales for both `x` and `y`.

Note that the logarithm of a negative value or of zero does not exist and if the data contains negative or zero values, a warning message will be printed by MATLAB informing you that these data points have been omitted from the data plotted. Examples are shown in Figure 5.4 below.

Plot Linestyles

An optional argument to the various plot commands above controls the linestyle. This argument is a character string consisting of one of the characters shown below. This character can be combined with the previously described characters that control colors and markers.

Symbol	Linestyle
−	solid line
:	dotted line
−.	dash-dot line
−−	dashed line

Example 5.1 *Vertical motion under gravity*

An object is thrown vertically upward with an initial speed *v*, under acceleration of gravity *g*. Neglecting air resistance, determine and plot the height of the object as a function of time, from time zero when the object is thrown until it returns to the ground.

The height of the object at time t is

$$h(t) = vt - \frac{1}{2}gt^2$$

The object returns to the ground at time t_g, when

$$h(t_g) = vt_g - \frac{1}{2}gt_g^2 = 0$$

or

$$t_g = \frac{2v}{g}$$

If the initial velocity v is 60 m/s, the following script will compute and plot the vertical height as a function of the time of flight.

```
% Vertical motion under gravity

% Define input values
g = 9.8;                         % acceleration of gravity (m/s^2)
v = 60;                          % initial speed (m/s)

% Calculate times
t_g = 2*v/g;                     % time (s) to return to ground
t = linspace(0,t_g,256);         % 256 element time vector (s)

% Calculate values for h(t)
h = v * t - g/2 * t.^2;          % height (m)

% Plot h(t)
plot(t, h, ':'), title('Vertical mtion under gravity'), ...
     xlabel('time (s)'), ylabel('Height (m)'), grid
```

The resulting plot is shown in Figure 5.1. Note that the option `':'` in the `plot` command produces a dotted line in the plot.

■

Multiple Curves

Multiple curves can be plotted on the same graph by using multiple arguments in a plot command, as in `plot(x,y,w,z)`, where the variables `x`, `y`, `w` and `z` are vectors. The curve of `y` versus `x` will be plotted, and then the curve corresponding to `z` versus `w` will be plotted on the same graph.

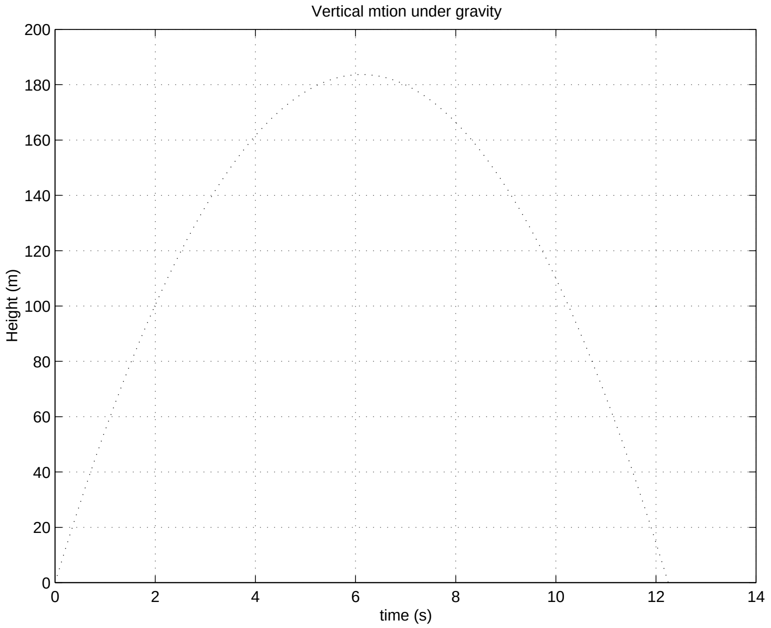


Figure 5.1: Height-time graph of vertical motion under gravity

Another way to generate multiple curves on one plot is to use a single matrix with multiple columns as the second argument of the `plot` command. Thus, the command `plot(x,F)`, where `x` is a vector and `F` is a matrix, will produce a graph in which each column of `F` is a curve plotted against `x`.

To distinguish between plots on the graph, the `legend` command is used. This command has the form:

```
legend('string1','string2','string3',...)
```

It places a box on the plot, with the curve type for first curve labeled with the text string `'string1'`, the second curved labeled with the text string `'string2'`, and so on. The position of the curve can be controlled by adding a final integer position argument `pos` to the `legend` command. Values of `pos` and the corresponding positions are:

0 = Automatic "best" placement (least conflict with data)
1 = Upper right-hand corner (default)
2 = Upper left-hand corner
3 = Lower left-hand corner
4 = Lower right-hand corner
-1 = To the right of the plot

Example 5.2 *Multiple plot of polynomial curves*

Consider plotting the two polynomial functions:

$$f_1 = x^2 - 3x + 2$$

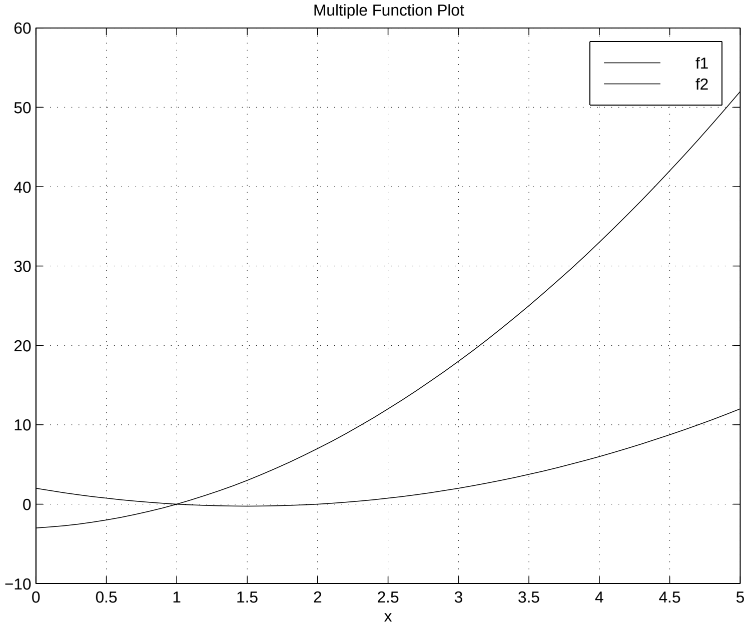


Figure 5.2: Plot with two polynomial curves

$$f_2 = 2x^2 + x - 3$$

```
% Generate plots of polynomials
%
x = 0:0.1:5;
f(:,1) = x'.^2 -3*x' + 2;
f(:,2) = 2*x'.^2 +x' - 3;
plot(x,f),title('Multiple Function Plot'),...
xlabel('x'), grid, legend('f1','f2')
```

The resulting plot is shown in Figure 5.2 (the line colors differ, allowing the curves to be distinguished from one another).

∎

To plot two curves with separate y-axis scaling and labeling, the `plotyy` function can be used. This command is apparently not well developed, as it does not accept line style options and the `legend` command causes one of the curves to disappear.

Example 5.3 *Multiple plot of polynomial curves with separate y axes*

Again consider plotting the two polynomial functions from the previous example, but with separate y axes.

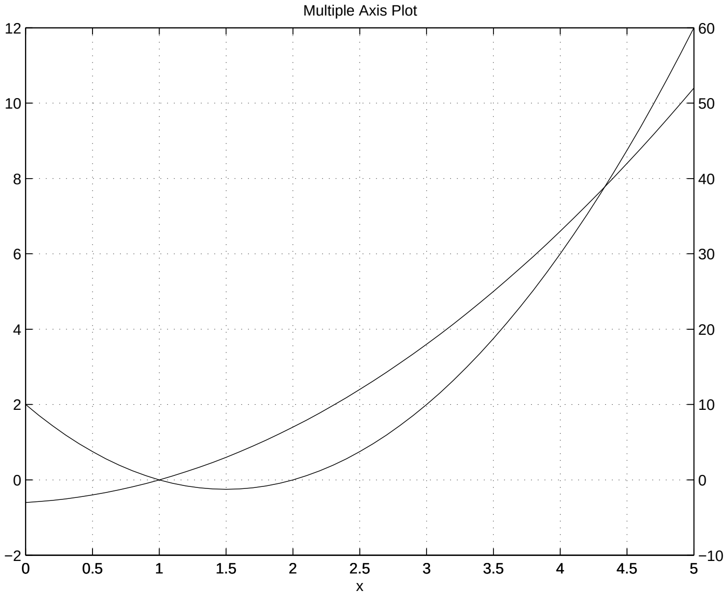


Figure 5.3: Plot with two polynomial curves, separate y axes

```
>> x = 0:0.1:5;
>> f1 = x’.ˆ2 -3*x’ +2;
>> f2 = 2*x’.ˆ2 +x’ -3;
>> plotyy(x,f1,x,f2),title(’Multiple Axis Plot’),...
xlabel(’x’),grid
```

The resulting plot is shown in Figure 5.3 (the line colors differ, allowing the curves to be distinguished from one another). The y axis for function f_1 is on the left and the y axis for function f_2 is on the right.

■

Multiple Figures

As has been described, when the first `plot` command is issued in a MATLAB session, a new window named “Figure No. 1” is created, containing the plot. A subsequent `plot` command will draw a new graph in this same window. The `figure` command allows the creation of multiple plot windows, allowing the display of multiple graphs. The command:

```
figure(n)
```

where `n` is a positive integer, will create a window named “Figure No. `n`.” Subsequent `plot` commands will draw graphs in this window, known as the active window.

To reuse a *Figure* window for a new plot, it must be made the active or *current* figure. Clicking on the figure off choice with the mouse makes it the active current figure. The command `figure(n)` makes the corresponding figure active or current. Only the active window is responsive to `axis`, `hold`, `xlabel`, `ylabel`, `title`, and `grid` commands.

Figure windows can be deleted by closing them with a mouse, using the conventional method for the windowing system in use on your computer. The current window can be closed with the command:

```
>> close
```

The figure `n` window can be closed with the command:

```
>> close(n)
```

All figure windows are closed with the command:

```
>> close(all)
```

To erase the contents of a figure window without closing it, use the command:

```
>> clf
```

To erase the contents and also reset all properties, such as `hold`, to their default states:

```
>> clf reset
```

Subplots

The `subplot` command allows you to split the graph window into **subwindows**. The possible splits are two subwindows (top and bottom or left and right) or four subwindows (two on top, two on the bottom).

`subplot(m,n,p)`: `m` by `n` grid of windows, with `p` specifying the current plot as the `p`th window.

Example 5.4 *Subplots of a polynomial function*

Consider plotting the polynomial

$$y = 5x^2$$

in subplots of different plot types.

```
% Generate subplots of a polynomial
%
```

```
x = 0:0.5:50;
y = 5*x.^2;
subplot(2,2,1),plot(x,y),...
  title('Polynomial - linear/linear (plot)'),...
  ylabel('y'),grid,...
subplot(2,2,2),semilogx(x,y),...
  title('Polynomial - log/linear (semilogx)'),...
  ylabel('y'),grid,...
subplot(2,2,3),semilogy(x,y),...
  title('Polynomial - linear/log (semilogy)'),...
  xlabel('x'),ylabel('y'),grid,...
subplot(2,2,4),loglog(x,y),...
  title('Polynomial - log/log (loglog)'),...
  xlabel('x'),ylabel('y'),grid
```

The resulting plot is shown in Figure 5.4.

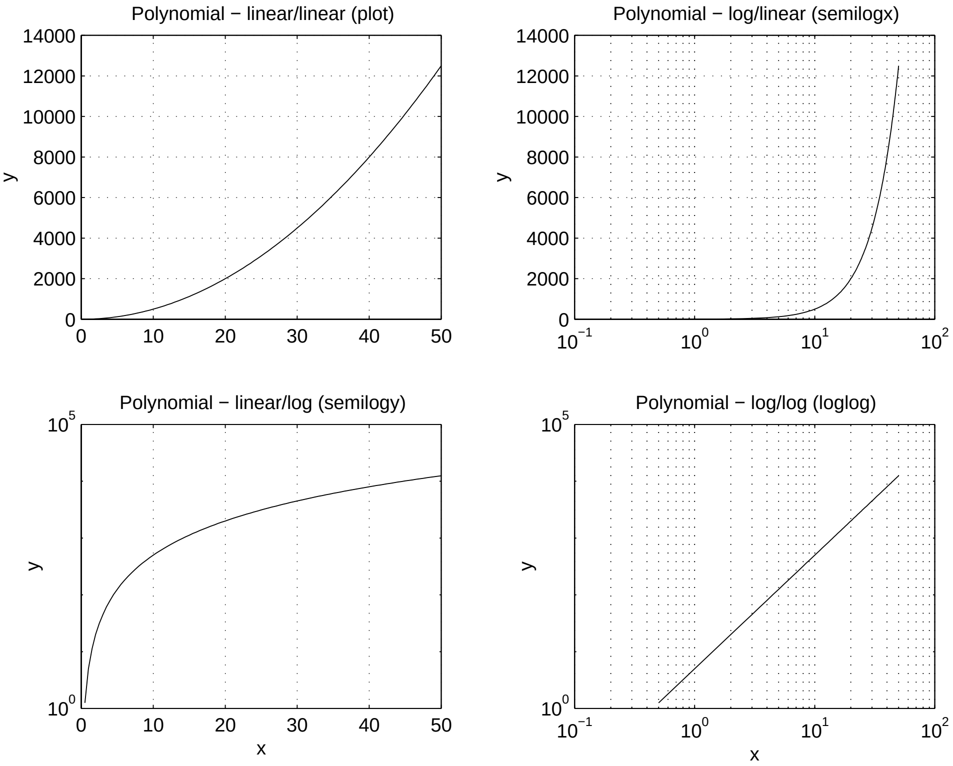


Figure 5.4: Plot with four polynomial subplots

■

Graphics Window Updates

The graphics or figure window is updated, or rendered, after each graphics command. This can be a time-consuming task, which can be avoided by entering all of the graphics command for a given figure window on the same line. This has been done in the example above, where all of the graphics commands have effectively been placed on the same line, by separating commands with commas and using ... to continue commands on the next line.

Section 6

Mathematical Functions and Applications

Outline

- Signals
- Polynomials
- Partial fraction expansion
- Functions of two variables
- User-defined functions
- Plotting functions

6.1 Signal Representation, Processing, and Plotting

As we have seen, one application of a one-dimensional array in MATLAB is to represent a function by its uniformly spaced samples. One example of such a function is a signal, which represents a physical quantity such as voltage or force as a function of time, denoted mathematically as $x(t)$. The value of x is known generically as the signal **amplitude**.

Sinusoidal Signal

A sinusoidal signal is a **periodic signal**, which satisfies the condition

$$x(t) = x(t + nT), \qquad n = 1, 2, 3, \ldots$$

where T is a positive constant known as the **period**. The smallest non-zero value of T for which this condition holds is the **fundamental period** T_0. Thus, the amplitude of the signal repeats every T_0.

Consider the signal

$$x(t) = A\cos(\omega t + \theta)$$

with **signal parameters**:

- A is the amplitude, which characterizes the peak-to-peak swing of $2A$, with units of the physical quantity being represented, such as volts.
- t is the independent time variable, with units of seconds (s).
- ω is the angular frequency, with units of radians per second (rad/s), which defines the fundamental period $T_0 = 2\pi/\omega$ between successive positive or negative peaks.
- θ is the initial phase angle with respect to the time origin, with units of radians, which defines the time shift $\tau = -\theta/\omega$ when the signal reaches its first peak.

With τ so defined, the signal $x(t)$ may also be written as

$$x(t) = A\cos\omega(t - \tau)$$

When τ is positive, it is a "time delay," that describes the time (greater than zero) when the first peak is reached. When τ is negative, it is a "time advance" that describes the time (less than zero) when the last peak was achieved. This sinusoidal signal is shown in Figure 6.1.

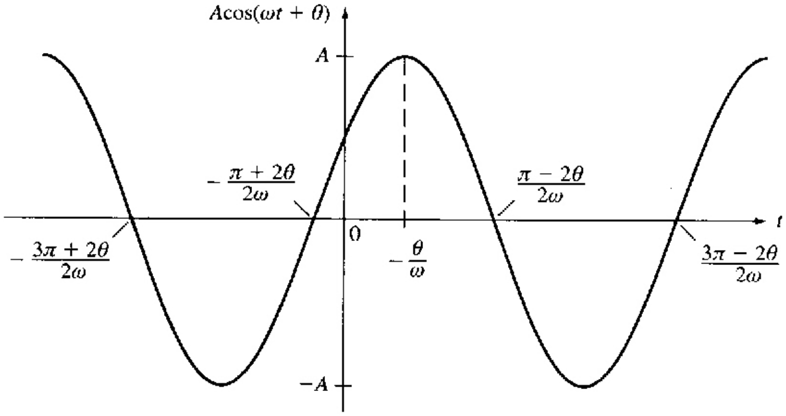


Figure 6.1: Sinusoidal signal $A\cos(\omega t + \theta)$ with $-\pi/2 < \theta < 0$.

Consider computing and plotting the following cosine signal

$$x(t) = 2\cos 2\pi t$$

Identifying the parameters:

- Amplitude $A = 2$
- Frequency $\omega = 2\pi$. The fundamental period $T_0 = 2\pi/\omega = 2\pi/2\pi = 1s$.
- Phase $\theta = 0$.

A time-shifted version of this signal, having the same amplitude and frequency is

$$y(t) = 2\cos[2\pi(t - 0.125)]$$

where the time shift $\tau = 0.125s$ and the corresponding phase is $\theta = 2\pi(-0.125) = -\pi/4$.

A third signal is the related sine signal having the same amplitude and frequency

$$z(t) = 2\sin 2\pi t$$

Preparing a script to compute and plot $x(t)$, $y(t)$, and $z(t)$ over two periods, from $t = -1s$ to $t = 1s$:

```
% sinusoidal representation and plotting
%
t = linspace(-1,1,101);
x = 2*cos(2*pi*t);
y = 2*cos(2*pi*(t-0.125));
z = 2*sin(2*pi*t);
plot(t,x,t,y,t,z),...
  axis([-1,1,-3,3]),...
  title('Sinusoidal Signals'),...
  ylabel('Amplitude'),...
  xlabel('Time (s)'),...
  text(-0.13,1.75,'x'),...
  text(-0.07,1.25,'y'),...
  text(0.01,0.80,'z'),grid
```

Thus, `linspace` has been used to compute the time row vector `t` having 101 samples or elements with values from -1 to 1. The three signals are computed as row vectors and plotted, with axis control, labels and annotation. The resulting plot is shown in Figure 6.2. Observe that $x(t)$ reaches its peak value of 2 at time $t = 0$, which we can verify as being correct since we know $\cos 0 = 1$. The signal $y(t)$ is $x(t)$ delayed by $\tau = 0.125s$. The signal $z(t)$ is 0 for $t = 0$, since $\sin 0 = 0$. It reaches its first peak at $t = T_0/4 = 0.25s$, as $\sin(2\pi \cdot 0.25) = \sin(\pi/2) = 1$.

Phasor Representation of a Sinusoidal Signal

An important and very useful representation of a sinusoidal signal is as a function of a complex exponential signal.

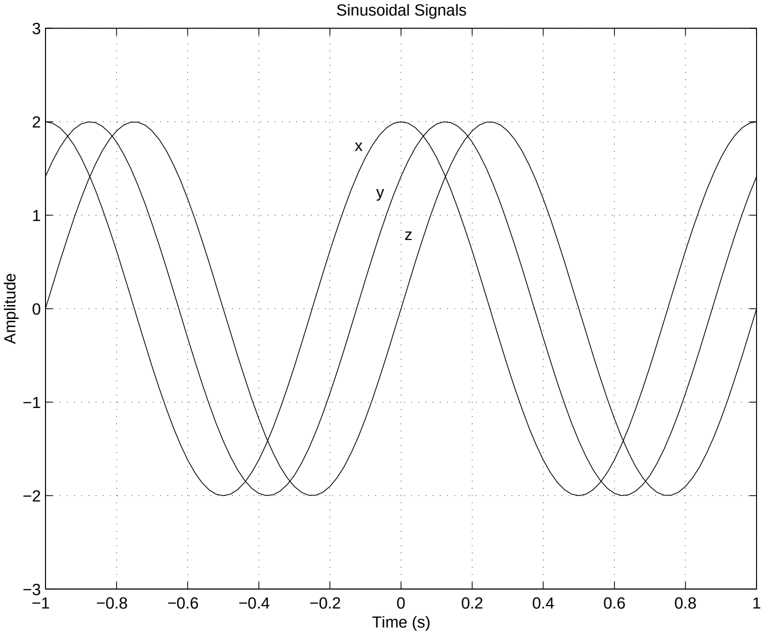


Figure 6.2: Sinusoidal signals

Recall

$$e^{j\theta} = \cos\theta + j\sin\theta$$

Thus,

$$\cos\theta = \text{Re}\left[e^{j\theta}\right]$$

As a result, the signal

$$x(t) = A\cos(\omega t + \phi)$$

can be *represented* as the real part of the complex exponential

$$\begin{aligned} x(t) &= \text{Re}\left[Ae^{j(\omega t+\phi)}\right] \\ &= \text{Re}\left[Ae^{j\phi}e^{j\omega t}\right] \end{aligned}$$

We call $Ae^{j\phi}e^{j\omega t}$ the complex representation of $x(t)$ and write

$$x(t) \longleftrightarrow Ae^{j\phi}e^{j\omega t}$$

meaning that the signal $x(t)$ may be reconstructed by taking the real part of $Ae^{j\phi}e^{j\omega t}$. In this representation, we call $Ae^{j\phi}$ the *phasor* or complex amplitude representation of $x(t)$ and write

$$x(t) \longleftrightarrow Ae^{j\phi}$$

meaning that the signal $x(t)$ may be reconstructed from $Ae^{j\phi}$ by multiplying by $e^{j\omega t}$ and taking the real part.

The phasor representation of the sinusoid $x(t) = A\cos(\omega t + \phi)$ is shown in the complex plane in Figure 6.3. At $t = 0$, the complex representation produces the phasor $Ae^{j\phi}$, where ϕ is approximately

$$Ae^{j\phi}e^{j\omega t_1}.$$

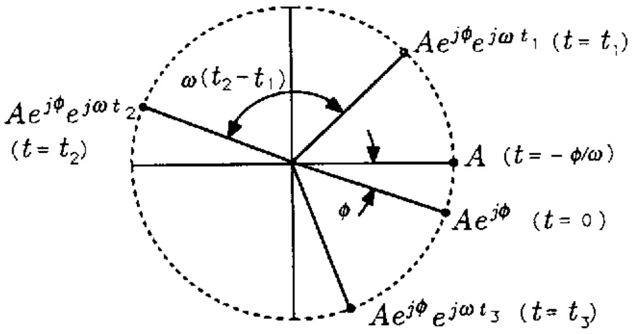

$Ae^{j\phi}e^{j\omega t_1}$ $(t = t_1)$
$\omega(t_2 - t_1)$
$Ae^{j\phi}e^{j\omega t_2}$
$(t = t_2)$
A $(t = -\phi/\omega)$
ϕ
$Ae^{j\phi}$ $(t = 0)$
$Ae^{j\phi}e^{j\omega t_3}$ $(t = t_3)$

Figure 6.3: Rotating phasor

$-\pi/10$. If time t increases to time t_1, then the complex representation produces

$$Ae^{j\phi}e^{j\omega t_1}$$

From our discussion of complex numbers, we know that $e^{j\omega t_1}$ rotates the phasor $Ae^{j\phi}$ through an angle ωt_1. Therefore, as we increase t from 0, we rotate the phasor from $Ae^{j\phi}$, producing the circular trajectory around the circle of radius A shown in Figure 6.3. When $t = 2\pi/\omega$, then $e^{j\omega t} = e^{j\pi} = 1$. Therefore, every $2\pi/\omega$ seconds, the phasor revisits any given position on the circle. The quantity $Ae^{j\phi}e^{j\omega t}$ is called a *rotating phasor* whose rotation rate is the frequency ω:

$$\frac{d}{dt}\omega t = \omega$$

The rotation rate is also the frequency of the sinusoidal signal $x(t) = A\cos(\omega t + \phi)$.

The real part of the complex, or rotating phasor, representation $Ae^{j\phi}e^{j\omega t}$ is the desired signal $x(t) = A\cos(\omega t + \phi)$. This real part is read off of the rotating phasor diagram as shown in Figure 6.4.

Consider computing and plotting the phasor

$$x(t) = e^{j\omega t}$$

where $\omega = 2000\pi$ rad/s and t ranges from -2×10^{-3} s to 2×10^{-3} s, in steps of 0.02×10^{-3} s. Also to be plotted are $y(t) = \text{Re}[x(t)]$ and $z(t) = \text{Im}[x(t)]$. The script file is

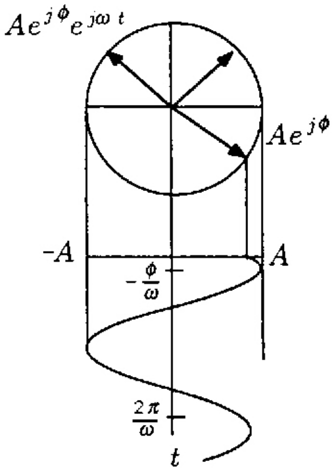


Figure 6.4: Reading a real signal from a complex, rotating phasor

```
% Phasor computation and plot
%
t = (-2e-03:0.02e-03:2e-03);
x = exp(j*2000*pi*t);
y = real(x);
z = imag(x);
subplot(2,1,1),plot(x,':'),...
   axis square,...
   title('exp(jwt)'),...
   xlabel('Real'),...
   ylabel('Imaginary'),...
subplot(2,1,2),plot(t,y,'-',t,z,':'),...
   title('Re[exp(jwt)] and Im[exp(jwt)] vs t   w=1000*2*pi'),...
   xlabel('Time (s)'),grid on,...
   legend('Re[exp(j \omega t)]','Im[exp(j \omega t)]',-1)
```

and the resulting plot is shown in Figure 6.5, where $y(t)$ is plotted with a solid line and $z(t)$ is plotted with a dotted line.

Harmonic Motion

A periodic motion that can be described by a sinusoidal function of time is called **harmonic motion**. A mechanism that produces such motion is a *scotch yoke*, shown in Figure 6.6. This mechanism is used in machines known as *shakers*, for testing the behavior of equipment subject to vibrations. The driving element is a rotating disk with a pin mounted a distance A from the center. The pin can slide in the slot of the element marked *x-yoke*. The motion of the *x-yoke* is restricted by a guided rod attached to it, so that this yoke can move only horizontally. A similar slotted element, marked *y-yoke*, is assembled above the *x-yoke*. The motion of the *y-yoke* is restricted by a guided rod that allows only vertical displacements.

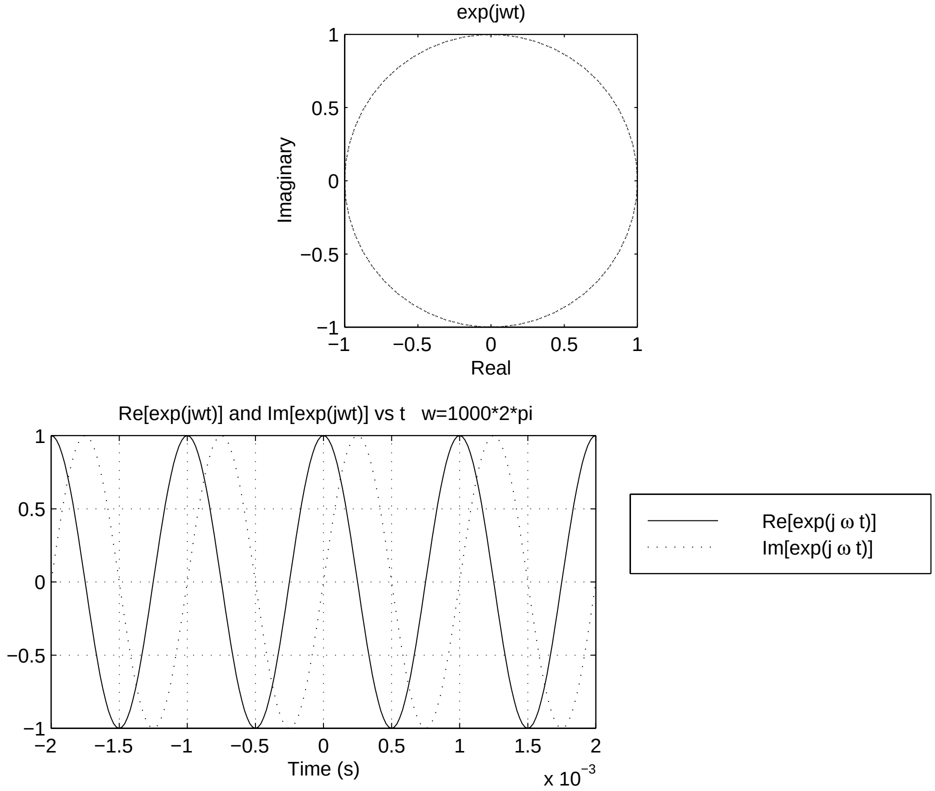


Figure 6.5: The signals $e^{j\omega t}$, $\text{Re}\left[e^{j\omega t}\right]$, and $\text{Im}\left[e^{j\omega t}\right]$

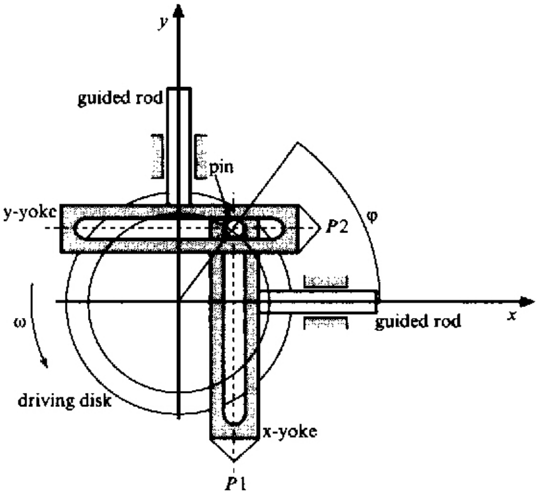


Figure 6.6: Scotch yoke mechanism

The motion is described mathematically by first assuming that at time $t = 0$ the position of the pin relative to the x axis is defined by the angle ϕ. The x-coordinate of the point $P1$ is

$$x(0) = A\cos(\phi)$$

and the y-coordinate of the point $P2$ is

$$y(0) = A\sin(\phi)$$

If the angular speed of the disk is ω radians per second, then the x-coordinate of the point $P1$ at time t is

$$x(t) = A\cos(\omega t + \phi)$$

and the y-coordinate of the point $P2$ is

$$y(t) = A\sin(\omega t + \phi)$$

Using the phasor representation for this motion, we can define

$$z(t) = Ae^{j\omega t + \phi}$$

and we observe that $x(t)$ is the real part and $y(t)$ is the imaginary part of $z(t)$. The point z can be considered to be the end of a vector with origin at the coordinate origin and magnitude A. This vector rotates with angular velocity ω, starting from angle ϕ.

If $y(t)$ is the **displacement** of the harmonic motion, the **velocity** is

$$v(t) = \frac{dy}{dt} = \omega A\cos(\omega t + \phi) = \omega A\sin(\omega t + \phi + \pi/2)$$

and the **acceleration** is

$$a(t) = \frac{d^2y(t)}{dt^2} = -\omega^2 A\sin(\omega t + \phi) = \omega^2 A\sin(\omega t + \phi + \pi)$$

We conclude that the velocity of the harmonic motion can be represented by a rotating vector leading the displacement by the phase $\pi/2$, and the acceleration can be represented by another rotating vector, leading the displacement by the phase angle π.

Phasor Properties

Positive and Negative Frequencies

An alternative phasor representation for the signal

$$x(t) = A\cos(\omega t + \phi)$$

is obtained by using the Euler identity

$$\cos\theta = \frac{1}{2}\left(e^{j\theta} + e^{-j\theta}\right)$$

which yields

$$\begin{aligned} x(t) &= \frac{A}{2}\left[e^{j(\omega t+\phi)} + e^{-j(\omega t+\phi)}\right] \\ &= \frac{A}{2}e^{j\phi}e^{j\omega t} + \frac{A}{2}e^{-j\phi}e^{-j\omega t} \end{aligned}$$

In this equation, the term $\frac{A}{2}e^{j\phi}e^{j\omega t}$ is a rotating phasor that begins at the phasor value $\frac{A}{2}e^{j\phi}$ and rotates counterclockwise with frequency ω. The term $\frac{A}{2}e^{-j\phi}e^{-j\omega t}$ is a rotating phasor that begins at the (complex conjugate) phasor value $\frac{A}{2}e^{-j\phi}$ (for $t = 0$) and rotates clockwise with (negative) frequency ω. The physically meaningful frequency for a cosine is ω, a positive quantity. A negative frequency is not physically meaningful, but just means that the direction of rotation for the rotating phasor is clockwise, not counterclockwise, in the complex exponential representation of the real sinusoid. Thus, the concept of negative frequency is just an artifact of the two-phasor representation. In the one-phasor representation, when we take the "real part," the artifact does not arise. You are likely to encounter both the one-phasor and two-phasor representations, so you should be familiar with both.

Adding Phasors

The sum of two signals with common frequencies but different amplitudes and phases is

$$A_1\cos(\omega t + \phi_1) + A_2\cos(\omega t + \phi_2).$$

The rotating phasor representation for this sum is

$$\left(A_1e^{j\phi_1} + A_2e^{j\phi_2}\right)e^{j\omega t}$$

The new phasor is

$$A_1e^{j\phi_1} + A_2e^{j\phi_2}$$

and the corresponding real signal is

$$x(t) = \mathrm{Re}\left[\left(A_1e^{j\phi_1} + A_2e^{j\phi_2}\right)e^{j\omega t}\right]$$

The new phasor is shown in Figure 6.7, where the parallelogram rule for adding complex numbers applies.

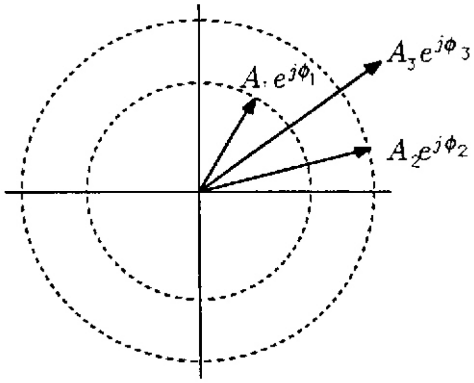


Figure 6.7: Adding phasors

Beating Between Tones

If you have heard two slightly mistuned musical instruments playing pure tones whose frequencies were close but not equal, you have sensed a beating phenomenon in which you perceive a single pure tone whose amplitude slowly varies periodically. The single perceived tone can be shown to have a frequency that is the average of the two mismatched frequencies, amplitude modulated by a tone whose "beat" frequency is half the difference between the two mismatched frequencies. This effect is shown in Figure 6.8.

To understand this phenomenon, begin with two pure tones whose frequencies are $\omega_0 + \omega_b$ and $\omega_0 - \omega_b$ (for example, $\omega_0 = 2\pi \times 1400$ rad/s and $\omega_b = 2\pi \times 100$ rad/s). The average frequency is ω_0 and the difference frequency is $2\omega_b$. You hear the sum of the two tones:

$$x(t) = A_1 \cos\left[(\omega_0 + \omega_b)t + \phi_1\right] + A_2 \cos\left[(\omega_0 - \omega_b)t + \phi_2\right]$$

Assume that the amplitudes are equal, with $A = A_1 = A_2$. The phases may be written as

$$\phi_1 = \phi + \psi \quad \text{and} \quad \phi_2 = \phi - \psi$$

with

$$\phi = \frac{1}{2}(\phi_1 + \phi_2) \quad \text{and} \quad \psi = \frac{1}{2}(\phi_1 - \phi_2)$$

Representing $x(t)$ as a complex phasor

$$\begin{aligned}
x(t) &= A \,\mathrm{Re}\left\{e^{j[(\omega_0+\omega_b)t+\phi+\psi]} + e^{j[(\omega_0-\omega_b)t+\phi-\psi]}\right\} \\
&= A \,\mathrm{Re}\left\{e^{j(\omega_0 t+\phi)}\left[e^{j(\omega_b t+\psi)} + e^{-j(\omega_b t+\psi)}\right]\right\} \\
&= 2A \,\mathrm{Re}\left\{e^{j(\omega_0 t+\phi)} \cos(\omega_b t + \psi)\right\} \\
&= 2A \cos(\omega_0 t + \phi)\cos(\omega_b t + \psi)
\end{aligned}$$

This is an amplitude modulated waveform, in which a low frequency signal with beat frequency ω_b modulates a high frequency signal with carrier frequency ω_0 rad/s. Over short periods of time, the modulating signal $\cos(\omega_b t + \psi)$ remains relatively constant while the carrier term $\cos(\omega_0 + \phi)$ produces many cycles of its pure tone. Thus, we perceive the pure tone at the carrier frequency ω_0, having an amplitude that varies sinusoidally at the beat frequency ω_b.

The following is the script to simulate beating tones in which $\omega_0 = 2\pi \times 1400$ rad/s and $\omega_b = 2\pi \times 100$ rad/s, resulting in the plot shown in Figure 6.8.

```
% beating sinusoidal tones
%
t = linspace(-1e-2,1e-2,1001);
x = cos(2*pi*1500*t) + cos(2*pi*1300*t);
m = 2*cos(2*pi*100*t);
plot(t,m,'b:',t,-m,'b:',t,x,'k'),...
  axis([-0.01 0.01 -2.4 2.4]),...
  title('Beating between tones'),...
  xlabel('Time (s)'),...
  ylabel('Amplitude')
```

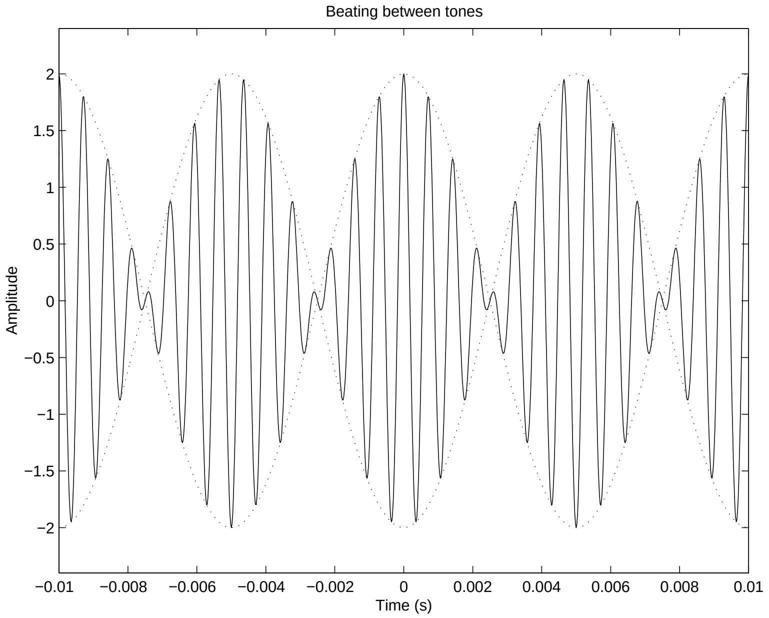


Figure 6.8: Beating between tones

6.2 Polynomials

A **polynomial** is a function of a single variable that can be expressed in the general form

$$A(s) = a_1 s^N + a_2 s^{N-1} + a_3 s^{N-2} + \cdots + a_N s + a_{N+1}$$

where the variable is s and the **polynomial coefficients** are the $N+1$ constants $a_1, a_2, \ldots, a_{N+1}$. The polynomial is of **degree** N, the largest value used as an exponent. The general form of a degree 3 (cubic) polynomial is

$$A(s) = a_1 s^3 + a_2 s^2 + a_3 s + a_4$$

and a specific example of a cubic polynomial is

$$A(s) = s^3 + 4s^3 - 7s - 10$$

Note that the notation used here is nonstandard, as the coefficient of term s^k is usually denoted as a_k. However, the nonstandard notation is more compatible with the indexing of arrays in MATLAB, as will be explained below.

For information on MATLAB functions supporting polynomial computations, type `help polyfun`.

Polynomial Evaluation

There are several ways to evaluate a polynomial for a set of values. Consider the cubic polynomial:

$$A(s) = s^3 + 4s^2 - 7s - 10$$

- Scalar `s`: use scalar operations

  ```
  A = s^3 + 4*s^2 - 7*s - 10;
  ```

- Vector or matrix `s`: use array or element-by-element operations:

  ```
  A = s.^3 + 4*s.^2 - 7*s - 10;
  ```

  The size of the vector or matrix `A` will be the same as that of `s`.

- Using `polyval(a,s)`: Evaluates a polynomial with coefficients in vector `a` for the values in `s`. The result is a matrix the same size as `s`. Element `a(1)` corresponds to coefficient a_1.

Consider evaluating and plotting $A(s)$ over the interval [-1,3]:

```
s = linspace(-1,3,201);
a = [1 4 -7 -10];
A = polyval(a,s);
plot(s,A),...
  title('Polynomial Function A(s) = s^3 + 4s^2 -7s -10'),...
  xlabel('s'),...
  ylabel('A(s)')
```

The resulting plot is shown in Figure 6.9.

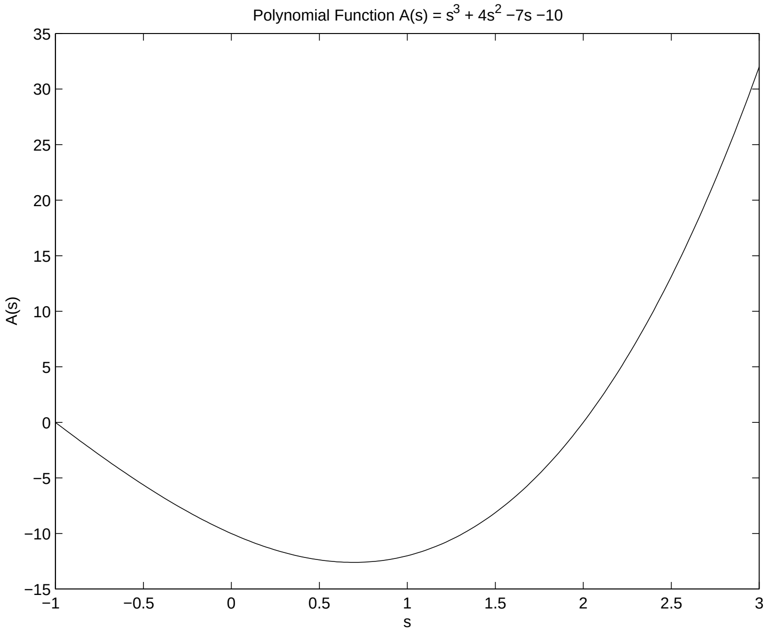


Figure 6.9: Plot of polynomial function $A(s)$

Polynomial Operations

By characterizing polynomial $A(s)$ by coefficients a_k stored in vector `a` and polynomial $B(s)$ by coefficients b_k stored in vector `b`, algebraic operations can be performed on the two polynomials.

- **Addition:** The coefficients of the sum of two polynomials is the sum of the coefficients of the two polynomials. The vectors containing the coefficients must be of the same length. For example, to add

$$A(s) = s^4 - 3s^3 - s + 2$$

$$B(s) = 4s^3 - 2s^2 + 5s - 16$$

$$\begin{aligned} C(s) &= A(s) + B(s) \\ &= s^4 + (4-3)s^3 - 2s^2 + (5-2)s + (2-16) \\ &= s^4 + s^3 - 2s^2 + 4s - 14 \end{aligned}$$

The script to perform this addition:

```
>> a = [1 -3 0 -1 2];
>> b = [0 4 -2 5 -16];
>> c = a + b
c =
     1     1    -2     4   -14
```

Thus,

$$C(s) = s^4 + s^3 - 2s^2 + 4s - 14$$

- **Scalar multiple:** The coefficient vector of the scalar multiple of a polynomial is simply the scalar times the coefficient vector of the polynomial. To specify

  $$C(s) = 3A(s)$$

  for $A(s)$ above, the following commands are used:

```
>> a = [1 -3 0 -1 2];
>> c = 3*a
c =
     3    -9     0    -3     6
```

  Thus

  $$C(s) = 3s^4 - 9s^3 - 3s + 6$$

- **Multiplication:** Apply the function `conv(a,b)`, which returns the coefficient vector for the polynomial resulting from the product of polynomials represented by the coefficients in `a` and `b`. The vectors `a` and `b` don't have to be of the same length. The resulting vector has length equal to the sum of the lengths of `a` and `b` minus one.

  Consider the following polynomial product, evaluated by hand using the "first-outer-inner-last" (FOIL) method:

  $$F(s) = (s + 2)(s + 3) = s^2 + 3s + 2s + 6 = s^2 + 5s + 6$$

  Using a MATLAB script

```
>> a = [1 2];
>> b = [1 3];
>> c = conv(a,b)
c =
     1      5      6
```

  Now consider multiplying polynomials $A(s)$ and $B(s)$ above:

  $$C(s) = A(s)B(s) = (s^4 - 3s^3 - s + 2)(4s^3 - 2s^2 + 5s - 16)$$

  Recall from your algebra class that evaluation by hand is a tedious process. Using a MATLAB script:

```
>> a = [1 -3 0 -1 2];
>> b = [4 -2 5 -16];
>> c = conv(a,b)
c =
     4   -14    11   -35    58    -9    26   -32
```

Thus,

$$C(s) = 4s^7 - 14s^6 + 11s^5 - 35s^4 + 58s^3 - 9s^2 + 26s - 32$$

- **Division:** You may have learned the “long division” method for evaluating the division of two polynomials. This method won’t be reviewed here, except to remind you that two results are determined: the quotient and the remainder. The result, expressed mathematically, is

  $$\frac{N(s)}{D(s)} = Q(s) + \frac{R(s)}{D(s)}$$

  where $N(s)$ is the numerator polynomial, $D(s)$ is the denominator polynomial, $Q(s)$ is the quotient polynomial, and $R(s)$ is the remainder polynomial.

  The MATLAB function to perform polynomial division:

  `[q,r] = deconv(n,d)` Returns quotient polynomial coefficients `q` and remainder polynomial coefficients `r` from numerator coefficients `n` and denominator coefficients `d`.

  Consider the evaluation of polynomial division in which the numerator is $C(s)$ and the denominator is $A(s)$, both from the multiplication example above. The result for the quotient should then be $B(s)$ above and the remainder should be all zeroes.

```
>> c = [4   -14    11   -35    58    -9    26   -32];
>> a =  [1 -3 0 -1 2];
>> [q r] = deconv(c,a)
q =
     4    -2     5   -16
r =
     0     0     0     0     0     0     0     0
```

  Now consider the following:

  $$H(s) = \frac{N(s)}{D(s)} = \frac{s^3 + 5s^2 + 11s + 13}{s^2 + 2s + 4}$$

  In MATLAB:

```
>> n = [1 5 11 13];
>> d = [1 2 4];
>> [q r] = deconv(n,d)
q =
     1     3
r =
     0     0     1     1
```

  Placing the result in mathematical form:

  $$\begin{aligned} H(s) &= Q(s) + \frac{R(s)}{D(s)} \\ &= s + 3 + \frac{s+1}{s^2 + 2s + 4} \end{aligned}$$

- **Derivatives:** You should be familiar with the rule for differentiating a polynomial term

$$\frac{d}{ds}as^n = nas^{n-1}$$

Applying this rule to the polynomial

$$P(s) = s^3 + 4s^2 - 7s - 10$$

produces the derivative

$$\frac{d}{ds}P(s) = 3s^2 + 8s - 7$$

Matlab provides function `polyder` for polynomial differentiation.

`polyder(p)`	Returns the coefficients of the derivative of the polynomial whose coefficients are the elements of vector p.
`polyder(a,b)`	Returns the coefficients of the derivative of the product polynomial $A(s) * B(s)$.
`[n,d] = polyder(b,a)`	Returns the derivative of the polynomial ratio $B(s)/A(s)$, represented as $N(s)/D(s)$.

Confirming the example above using the first form of `polyder`:

```
>> p = [1 4 -7 -10];
>> d = polyder(p)
d =
     3     8    -7
```

The second form of `polyder` returns the coefficients of the derivative of a product of two polynomials. This is equivalent to multiplying the two polynomials with `conv` and then differentiating using the first form of `polyder`. Consider applying the second form of `polyder` to the second example of polynomial multiplication above. The result can be confirmed by differentiating $C(s)$ above by hand.

```
>> a = [1 -3 0 -1 2];
>> b = [4 -2 5 -16];
>> dc = polyder(a,b)
dc =
    28   -84    55  -140   174   -18    26
```

The derivative of a polynomial ratio is more difficult to evaluate by hand. Using the derivative notation

$$f' = \frac{df}{ds},$$

recall the quotient rule for differentiation

$$\left(\frac{f}{g}\right)' = \frac{gf' - fg'}{g^2}$$

Thus, for the polynomial ratio

$$H(s) = \frac{s+2}{s+3}$$

the derivative is

$$\frac{dH(s)}{ds} = \frac{(s+3)-(s+2)}{(s+3)^2} = \frac{1}{(s+3)^2} = \frac{1}{s^2+6s+9}$$

Applying the third form of `polyder`

```
>> b = [1 2];
>> a = [1 3];
>> [q,d] = polyder(b,a)
q =
     1
d =
     1     6     9
```

Note that the denominator polynomial `d` in the result is in expanded, rather than factored form.

Roots of Polynomials

In many engineering problems, there is a need to find the **roots** of a polynomial $P(s)$, which are the values of s for which $P(s) = 0$. When $P(s)$ is of degree N, then there are exactly N roots, which may be **repeated roots** or **complex roots**. If the polynomial coefficients $(a_1, a_2, \ldots)$ are real, then any complex roots will always occur in complex conjugate pairs.

For degree one (linear) or two (quadratic), the roots are easily determined. The quadratic equation can be used for degree two, as described earlier. For polynomials of degree 3 and higher, numerical techniques are required to find the roots. The Matlab function for finding the roots:

`roots(a)`	Returns as a vector the roots of the polynomial represented by the coefficient vector `a`.

Consider the denominator polynomial from the example above

$$D(s) = (s+3)^2 = s^2 + 6s + 9$$

```
>> d = [1 6 9];
>> roots(d)
ans =
    -3
    -3
```

This confirms that there are two roots at -3 in the denominator.

Consider the cubic polynomial

$$P(s) = s^3 - 2s^2 - 3s + 10$$

The commands to compute and display the roots:

```
>> p = [1,-2,-3,10];
>> r = roots(p)
r =
   2.0000+ 1.0000i
   2.0000- 1.0000i
  -2.0000
```

Note that there are 3 roots for the degree 3 polynomial, with a complex conjugate pair of roots and a real root. To verify that these values are roots, evaluate the polynomial at the roots:

```
>> P= polyval(p,r)
P =
  1.0e-013 *
  -0.0355+ 0.1377i
  -0.0355- 0.1377i
   0.0711
```

While $P(r)$ is not exactly zero, due to the limitations on numerical accuracy, for each root it is of the order of 10^{-14}.

The roots can be used to express the polynomial in factored form. For example:

$$P(s) = s^3 - 2s^2 - 3s + 10 = (s - r_1)(s - r_2)(s - r_3) = (s - 2 - j)(s - 2 + j)(s + 2)$$

The coefficients of the polynomial can be determined from the roots using the `poly` function:

`poly(r)` Returns as a row vector the coefficients of the polynomial whose roots are contained in the vector `r`.

For example:

```
>> a = poly(r)
a =
    1.0000   -2.0000   -3.0000   10.0000
```

Example 6.1 *Finding the depth of a well using roots of a polynomial*

Consider the problem of finding the depth of a well by dropping a stone and measuring the time t to hear a splash. This time is composed of the time t_1 of free fall from release to reaching the water and the time t_2 that the sound takes to travel from the water surface to the ear of the person dropping the stone. Let g denote the acceleration of gravity, d the well depth (approximately equal to the distance between the hand or the ear of the person and the water surface), and c the speed of sound in air. The depth d, the distance traveled by the stone during time t_1 is

$$d = \frac{g}{2} \cdot t_1^2$$

or

$$t_1 = \sqrt{2d/g}$$

The same distance traveled by the sound during t_2 is

$$d = c \cdot t_2$$

or

$$t_2 = d/c$$

The total time is

$$t = t_1 + t_2 = \sqrt{2d/g} + d/c$$

Squaring the equation above and rearranging the terms

$$d^2 - 2(tc + c^2/g)d + c^2t^2 = 0$$

The depth d is the solution (roots) of the quadratic polynomial equation above. If the measured time t was 2.5s and the speed of sound c in air at atmospheric pressure and 20° Celsius is 343m/s, the following MATLAB script can be used to calculate the depth d.

```
% Well depth problem
%
% Define input values
t = 2.5;          % time to hear splash (s)
g = 9.81;         % acceleration of gravity (m/s^2)
c = 343;          % speed of sound in air (m/s)

% Calculate polynomial coefficients
a(1) = 1;
a(2) = -2*(t*c + c^2/g);
```

```
a(3) = (c*t)^2;

% Find roots corresponding to depth
depth = roots(a)
```

The displayed results from this script:

```
depth =
  1.0e+004 *
    2.5672
    0.0029
```

As is the case in many problems involving roots of a quadratic equation, one of the solutions is not physically reasonable. In this problem, the first root gives an impossibly large well depth, while the second root gives a reasonable depth. Investigating this solution with Matlab:

```
>> d = depth(2)
d =
   28.6425
>> t1 = sqrt(2*d/g)
t1 =
    2.4165
>> t2 = d/c
t2 =
    0.0835
>> t = t1 + t2
t =
    2.5000
```

Thus, the depth is 28.6m, confirming the time of 2.5s.

■

6.3 Partial Fraction Expansion

A **rational function** is a ratio of polynomials having the form

$$H(s) = \frac{B(s)}{A(s)} = \frac{b_1 s^m + b_2 s^{m-1} + \cdots + b_m s + b_{m+1}}{a_1 s^n + a_2 s^{n-1} + \cdots + a_n s + a_{n+1}}$$

For $m < n$, $H(s)$ is known as a **proper rational function** and for $m \geq n$, it is known as an **improper rational function**. Denoting the roots of the denominator by $r_1, r_2, \ldots r_n$, $A(s)$ can be written in factored form as

$$A(s) = a_1(s - r_1)(s - r_2) \cdots (s - r_n)$$

and $H(s)$ can be written as

$$H(s) = \frac{B(s)}{a_1(s-r_1)(s-r_2)\cdots(s-r_n)}$$

Partial fraction expansion is a technique to express $H(s)$ as a sum of terms.

Expansion of Proper Rational Functions

For a proper rational function ($m < n$), there are three different cases of partial fraction expansion that must be considered:

1. Distinct (nonrepeated) real roots.
2. Distinct complex roots.
3. Repeated roots.

Distinct Real Roots

When the roots $r_1, r_2, \ldots, r_n$ are distinct and real, then by **partial fraction expansion** $H(s)$ can be expressed as

$$H(s) = \frac{c_1}{s-r_1} + \frac{c_2}{s-r_2} + \cdots + \frac{c_n}{s-r_n}$$

where the constants c_i are called the **residues**, which can be computed using the `residue` command:

`[c,r] = residue(b,a)` finds the residues `c` and roots `r` of a partial fraction expansion of the ratio of two polynomials $B(s)/A(s)$. Vectors `b` and `a` specify the coefficients of the numerator and denominator polynomials in descending powers of s. The residues are returned in the column vector `c` and the roots in column vector `r`.

Example:

$$H(s) = \frac{s+2}{s^3+4s^2+3s}$$

```
>> b = [1 2];
>> a = [1 4 3 0];
>> [c,r] = residue(b,a)
c =
   -0.1667
   -0.5000
    0.6667
```

```
r =
    -3
    -1
     0
```

$$
\begin{aligned}
H(s) &= -\frac{0.1667}{s-(-3)} - \frac{0.5000}{s-(-1)} + \frac{0.6667}{s-0} \\
&= -\frac{1/6}{s+3} - \frac{1/2}{s+1} + \frac{2/3}{s}
\end{aligned}
$$

Distinct Complex Roots

Partial fraction expansion applies as well to distinct complex roots. Note that if root r_1 is complex, then the complex conjugate r_1^* is also a root. It can also be shown that the residue c_2 corresponding to the root r_2 is equal to the conjugate c_1^* of the residue corresponding to the root r_1.

Example:

$$H(s) = \frac{s^2 - 2s + 1}{s^3 + 3s^2 + 4s + 2}$$

```
>> b = [1 -2 1];
>> a = [1 3 4 2];
>> [c,r] = residue(b,a)
c =
  -1.5000+ 2.0000i
  -1.5000- 2.0000i
   4.0000
r =
  -1.0000+ 1.0000i
  -1.0000- 1.0000i
  -1.0000
```

$$H(s) = \frac{-1.5 + j2}{s+1-j} + \frac{-1.5 - j2}{s+1+j} + \frac{4}{s+1}$$

Repeated Roots

Again consider the general case where $m < n$ and

$$H(s) = \frac{B(s)}{A(s)}$$

Suppose that root r_1 of $A(s)$ is repeated p times and the other $n{-}p$ roots (denoted $r_{p+1}, r_{p+2}, \ldots, r_n$) are distinct. Then $H(s)$ has the partial fraction expansion

$$H(s) = \frac{c_1}{s - r_1} + \frac{c_2}{(s - r_1)^2} + \cdots + \frac{c_p}{(s - r_1)^p} + \frac{c_{p+1}}{s - r_{p+1}} + \cdots + \frac{c_n}{s - r_n}$$

Example:

$$H(s) = \frac{5s - 1}{s^3 - 3s - 2}$$

```
>> b = [5 -1];
>> a = [1 0 -3 -2];
>> [c,r] = residue(b,a)
c =
    1.0000
   -1.0000
    2.0000
r =
    2.0000
   -1.0000
   -1.0000
```

$$H(s) = \frac{1}{s - 2} - \frac{1}{s + 1} + \frac{2}{(s + 1)^2}$$

Expansion of Improper Rational Functions

Again consider the rational function

$$H(s) = \frac{B(s)}{A(s)}$$

with the degree of $B(s)$ greater than or equal to the degree of $A(s)$ ($m \geq n$). By polynomial division, $H(s)$ can be written in the form

$$H(s) = Q(s) + \frac{R(s)}{A(s)}$$

where the quotient $Q(s)$ is a polynomial in s with degree $m - n$, and the remainder $R(s)$ is a polynomial in s with degree strictly less than n. Thus, $R(s)/A(s)$ is a proper rational function that can be expanded using partial fraction expansion.

The expansion of an improper rational function in Matlab is computed with a variation of the `residue` function:

`[c,r,q] = residue(b,a)` finds the residues, roots and quotient of a partial fraction expansion of the ratio of two polynomials B(s)/A(s). The coefficients of the quotient Q(s) are returned in the row vector `q`. The number of roots is `n = length(a)-1 = length(c) = length(r)`. The quotient coefficient vector is empty if `length(b) < length(a)`, otherwise `length(q) = length(b)-length(a)+1.`

Example:

$$H(s) = \frac{s^3 + 2s - 4}{s^2 + 4s - 2}$$

```
>> b = [1 0 2 -4];
>> a = [1 4 -2];
>> [c,r,q] = residue(b,a)
c =
   20.6145
   -0.6145
r =
   -4.4495
    0.4495
q =
     1    -4
```

$$H(s) = s - 4 + \frac{20.6145}{s + 4.4495} - \frac{0.6145}{s - 0.4495}$$

Recovering the Rational Function

The coefficients of the rational function can be recovered from the residues, the roots, and the coefficients of the quotient term using yet another form of the `residue` function.

`[b,a] = residue(c,r,q)`, with three input arguments and two output arguments, converts the partial fraction expansion back to the polynomials with coefficients in `b` and `a`.

Consider the continuation of the example above.

```
>> [b,a] = residue(c,r,q)
b =
    1.0000         0    2.0000   -4.0000
a =
     1     4    -2
```

$$H(s) = \frac{s^2 + 2s - 4}{s^2 + 4s - 2}$$

6.4 Functions of Two Variables

To evaluate a function of two variables $f(x, y)$, first define a **two-dimensional grid** in the xy plane, then evaluate the function at the grid points to determine points on the **three-dimensional surface**. This is shown in Figure 6.10, where the surface values $z = f(x, y)$ are plotted above the grid of xy values.

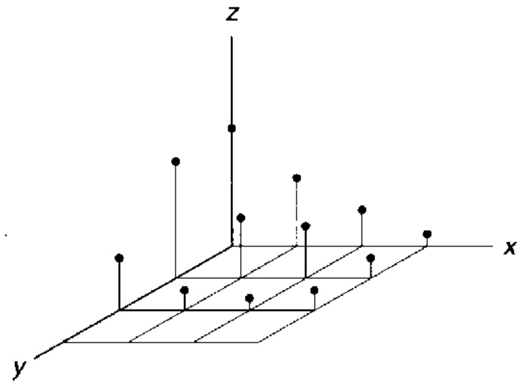


Figure 6.10: Function of two variables plotted in three dimensions

A two-dimensional grid in the xy plane is defined in MATLAB by two vectors, one containing the x-coordinates at all the points in the grid, and the other containing the y-coordinates. For example, to define a grid in x varying from -2 to 2 in increments of 1 and a grid in y varying from -1 to 2 in increments of 1, using colon notation:

```
>> x = -2:2
x =
    -2    -1     0     1     2
>> y = -1:2
y =
    -1     0     1     2
```

The `meshgrid` function generates two matrices that define the underlying grid for a two-dimensional function.

`[X,Y] = meshgrid(x,y)` Transforms the domain specified by vectors `x` and `y` into arrays `X` and `Y` that can be used for the evaluation of functions of two variables and 3-D surface plots. The rows of the output array `X` are copies of the vector `x` and the columns of the output array `Y` are copies of the vector `y`. If `x` has length n and `y` has length m, then `X` and `Y` are $m \times n$ arrays.

`[X,Y] = meshgrid(x)` Abbreviation for `[X,Y] = meshgrid(x,x)`

For example:

```
>> [X,Y] = meshgrid(x,y)
```

```
X =
    -2    -1     0     1     2
    -2    -1     0     1     2
    -2    -1     0     1     2
    -2    -1     0     1     2
Y =
    -1    -1    -1    -1    -1
     0     0     0     0     0
     1     1     1     1     1
     2     2     2     2     2
```

After the underlying grid matrices have been defined, the corresponding function values can be computed. For example, for the following function

$$f(x,y) = ye^{-(x^2+y^2)}$$

the function values would be computed as

```
Z = Y.* exp(-(X.^2 + Y.^2));
```

The operations are element-by-element, so the value `Z(1,1)` is computed from `X(1,1)` and `Y(1,1)`, and so on. Note that `Z` must be computed from `X` and `Y` and not from `x` and `y`, a common error.

Three-dimensional plots

Among several ways to plot a three-dimensional (3D) surface in MATLAB, a **mesh plot** and a **surface plot** will be described, followed by a description of a **contour plot**. For further information on 3D plots, type `help graph3d`.

`mesh(x_pts,y_pts,Z)` Generates an open mesh plot of the surface defined by matrix `Z`. The arguments `x_pts` and `y_pts` can be vectors defining the ranges of the values of the x- and y-coordinates, or they can be matrices defining the underlying grid of x- and y-coordinates.

`surf(x_pts,y_pts,Z)` Generates a shaded mesh plot of the surface defined by matrix `Z`. The arguments `x_pts` and `y_pts` can be vectors defining the ranges of the values of the x- and y-coordinates, or they can be matrices defining the underlying grid of x- and y-coordinates.

For example, to plot the function $f(x,y)$ in the example above, the following commands are executed

```
% Mesh and surface plots of a function of two variables
%
x = -2:0.1:2;
y = -1:0.1:2;
```

```
[X Y] = meshgrid(x,y);
Z = Y.*exp(-(X.^2 + Y.^2));
subplot(2,1,1),mesh(X,Y,Z),...
  title('Mesh Plot'),xlabel('x'),...
  ylabel('y'),zlabel('z'),...
subplot(2,1,2),surf(X,Y,Z),...
  title('Surface Plot'),xlabel('x'),...
  ylabel('y'),zlabel('z')
```

Note that the xy grid has been made finer by incrementing both x and y by 0.1. Also note that the arguments `X` and `Y` could have been replaced by `x` and `y` in both the `mesh` and `surf` commands. The resulting plots are shown in Figure 6.11. Note that you need to know something about the properties of the two-dimensional function $f(x, y)$ to know what range of values on x and y that you want it to be plotted.

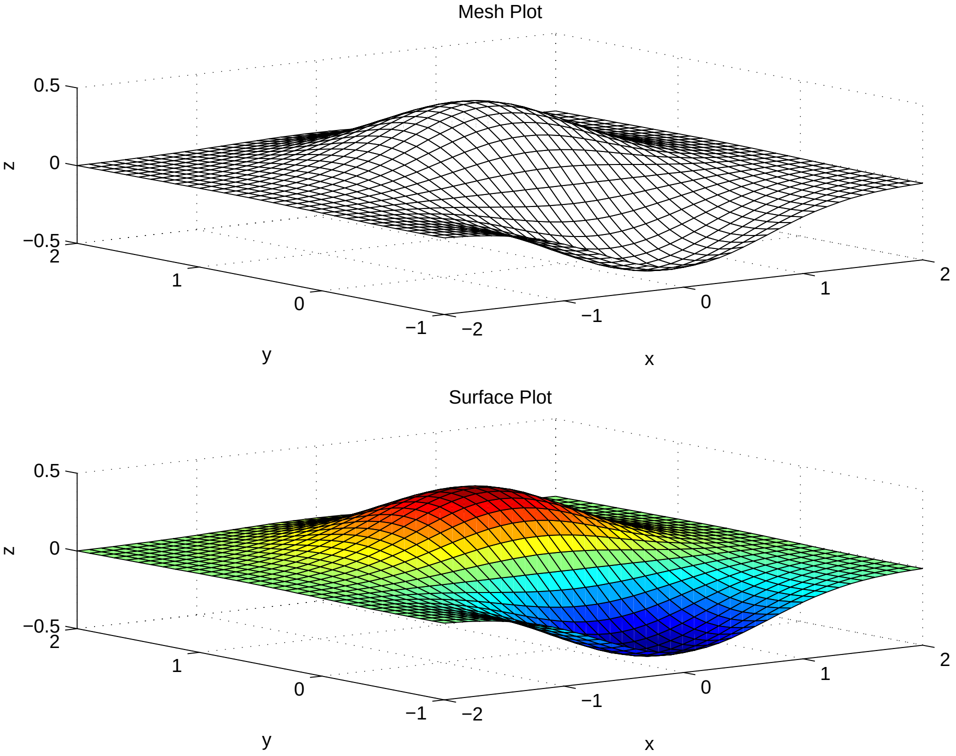


Figure 6.11: Mesh and surface plots of a function of two variables

A **contour plot** is an elevation or topographic map consisting of curves representing equal elevations or values of z, called contours of constant elevation.

`contour(x,y,Z)` Generates a contour plot of the surface defined by the matrix `Z`. The arguments `x` and `y` are vectors defining the ranges of values of the x- and y-coordinates. The number of contour lines and their values are chosen automatically.

`contour(x,y,Z,v)` Generates a contour plot of the surface defined by the matrix `Z`. The arguments `x` and `y` are vectors defining the ranges of values of the x- and y-coordinates. The vector `v` defines the values to use for the contour lines.

`meshc(x_pts,y_pts,Z)` Generates an open mesh plot of the surface defined by the matrix `Z`. The arguments `x_pts` and `y_pts` can be vectors defining the ranges of values of the x- and y-coordinates or they can be matrices defining the underlying grid of x- and y-coordinates. In addition, a contour plot is generated below the mesh plot.

The commands to produce the contour plot shown in Figure 6.12 from the function $f(x,y)$ considered in the examples above:

```
contour(x,y,Z),...
  title('Contour Plot'),xlabel('x'),...
  ylabel('y'),grid
```

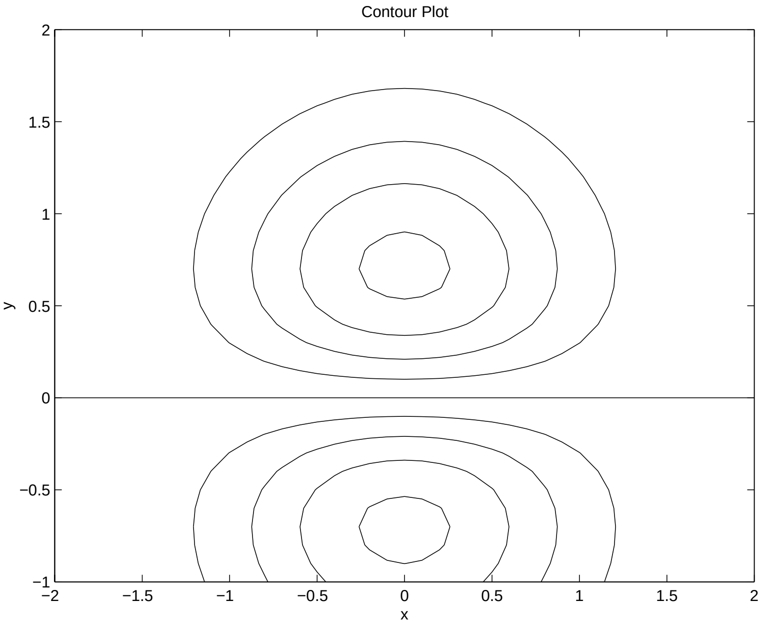


Figure 6.12: Contour plot of a function of two variables

The commands to produce the mesh/contour plot shown in Figure 6.13 from the function $f(x,y)$ considered in the examples above:

```
meshc(X,Y,Z),...
```

```
title('Mesh/Contour Plot'),xlabel('x'),...
ylabel('y'),zlabel('z')
```

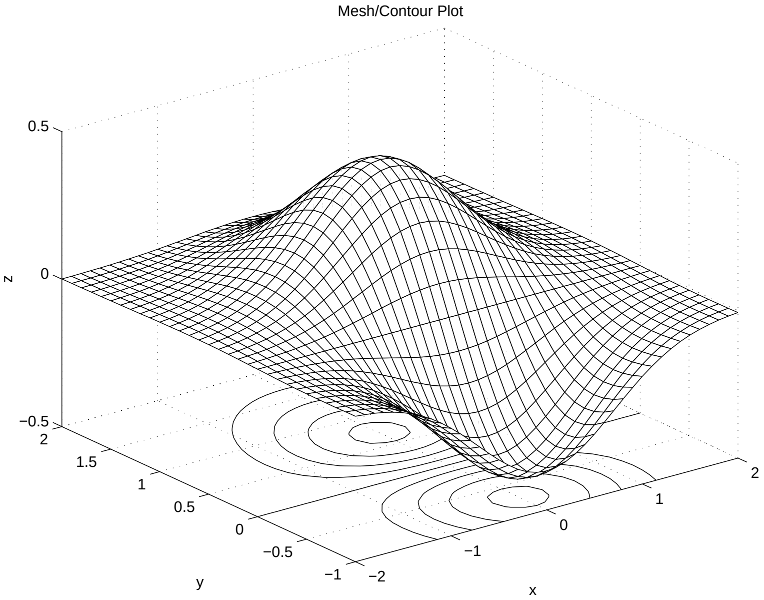


Figure 6.13: Mesh/contour plot of a function of two variables

6.5 User-Defined Functions

For more information, type `help function` in Matlab.

If you find that you are often building a function from several Matlab commands, you can develop a **user-defined function** that can can be used in a same way as the built-in Matlab functions.

A user-defined function is similar to a script file in that it is a text file having a `.m` extension and it is thus called a function M-file. Their variables are *local*, meaning that their values are available only within the function. They are the building blocks of larger scripts, facilitating a modular approach to the development of scripts.

For example, consider writing a function to compute the sine of an angle, with the angle in degrees rather than radians. The user-defined function is the following:

```
function y = sind(x)
%  SIND   Sine in degrees
%    SIND(X) is the sine of the elements of X, in degrees
y = sin(x*pi/180);
```

This function is written to a file named `sind.m`. Matlab programs and scripts can refer to this function in the same way that they refer to functions such as `sqrt` and `abs`. An example of the use of this to simplify a command in the script written for Example 4.2 is:

```
>> AC = 245/sind(30)
AC =
  490.0000
```

The rules for writing an M-file function are the following:

1. **Function definition line:** The first line of a function has the following syntax:

   ```
   function [output variables] = function_name(input variables);
   ```

   Thus, it must contain the word `function`, followed by the output variables, an equal sign, and the name of the function. The input variables, called arguments, of the function follow the function name and are enclosed in parentheses. This line distinguishes the function file from a script file. The definition line for the example above is:

   ```
   function y = sind(x)
   ```

   The output variable is `y`, the function name is `sind`, and the input argument is `x`.

2. **Function call:** A user-defined function is called by the name of the M-file in which it is defined, *not* by the name given the function in the first line of the file. Thus, if the function script above were renamed `dsin.m`, but the script itself were unchanged, then it would have to be called by the name `dsin`, as follows:

   ```
   >> y = dsin(30)
   y =
       0.5000
   ```

   An attempt to call it by the function name `sind` results in an error message:

   ```
   >> y = sind(30)
   ??? Undefined function or variable 'sind'.
   ```

   To avoid confusion, use the same name for the function and the M-file.

3. **Comments:** The first few lines should be comments, as they will be displayed if `help` is requested for the function name. The first comment line is referenced by the `lookfor` command. Each comment line must start with a percent character (%). For the example above:

   ```
   >> help sind

      SIND   Sine in degrees
        SIND(X) is the sine of the elements of X, in degrees
   ```

4. **Information returned:** The only information returned from the function is contained in the output variables (also called output arguments). A statement must always be included that assigns a value to the output variables specified in the function definition line. These output variables will be arrays. Thus, while we thought of the function `sind` as operating on a scalar and returning a scalar, it can also operate on an array and return an array:

```
>> lengths = 100*sind([30 60 90; 120 150 180])
lengths =
   50.0000   86.6025  100.0000
   86.6025   50.0000    0.0000
```

Note that output variables are optional. This allows a function to be written to perform an operation such as toggling `diary`, but not to return any information.

5. **Communication:** A function communicates with the Matlab workspace only through the variables passed to it and through the output variables it creates. Intermediate variables within the function do not appear in, or interact with, the Matlab workspace. Thus, each function has its own workspace separate from the Matlab workspace. Variables in the function M-file that are not output are input variables are said to be *local* to the function.

6. **Multiple outputs:** To return more than one output variable, they must be listed in a vector, separated by commas, as in the following example that will return three variables:

   ```
   function [distance, velocity, accel] = position(x)
   ```

   Since the function is returning multiple arguments, it must be used as follows:

   ```
   >> [d v a] = position(A)
   ```

7. **Multiple inputs:** When there are multiple input arguments, they must be separated by commas. For example:

   ```
   function c = hypot(a,b)
   ```

8. **Semicolons:** The use of semicolons (;) at the end of commands in a function script have the same purpose that they serve an any other Matlab command, suppressing display of the results of the command. In most cases, it is not desirable to display the results of commands internal to a function.

Many of the commands available in Matlab are written as function M-files. You can find the locations of these files using the `which` command. For example:

```
>> which linspace
/usr/pkg/matlab52/toolbox/matlab/elmat/linspace.m
```

You could display this M-file using the `type` command and take ideas from the function script for use in writing your own function scripts.

Application: Minimizing a Function of One Variable

To find the minimum of a function of a single variable:

`x = fminbnd('F',x1,x2)` Returns a value of `x` that is the local minimizer of `F(x)` in the interval [`x1`, `x2`], where `F` is a string containing the name of the function.

`[x,fval] = fminbnd(...)` Also returns the value of the objective function, `fval`, computed in `F`, at `x`.

For example:

```
>> fminbnd('cos',0,4)
ans =
    3.1416
```

To use this command to find the minimum of more complicated functions, it is convenient to define the function in a function M-file. For example, consider the polynomial function

$$y = 0.025x^5 - 0.0625x^4 - 0.333x^3 + x^2$$

Defining the function file `fp5.m`:

```
function y = fp5(x)
% FP5, fifth degree polynomial function
y = 0.025*x.^5 - 0.0625*x.^4 - 0.333*x.^3 + x.^2
```

Observe in Figure 6.5 that this function has two minima in the interval $-1 \leq x \leq 4$. The minimum near $x = 3$ is called a *relative* or *local* minimum because it forms a valley whose lowest point is higher than the minimum at $x = 0$. The minimum at $x = 0$ is the true minimum and is also called the *global* minimum. Searching for the minimum over the interval $-1 \leq x \leq 4$:

```
>> xmin = fminbnd('fp5',-1,4)
xmin =
  2.0438e-006
```

The resulting value for `xmin` is essentially 0, the true minimum point. Searching for the minimum over the interval $0.1 \leq x \leq 2.5$:

```
>> xmin = fminbnd('fp5',0.1,2.5)
xmin =
    0.1001
```

This misses the true minimum point, as it is not included in the specified interval. Also, `fminbnd` can give incorrect answers. If the interval is $1 \leq x \leq 4$:

```
>> xmin = fminbnd('fp5',1,4)
xmin =
    2.8236
```

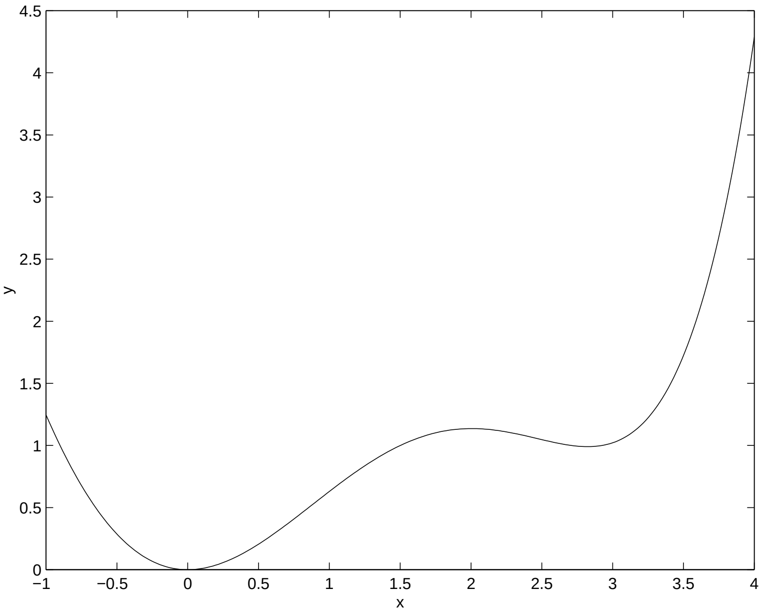


Figure 6.14: Plot of the function $y = 0.025x^5 - 0.0625x^4 - 0.333x^3 + x^2$

The result corresponds to the “valley” shown in the plot, but which is not the minimum point on this interval, which is at the boundary $x = 1$. The `fminbnd` function first looks for a minimum point corresponding to a zero slope; if it finds one, it stops. If it does not find one, it looks at the function values at the boundaries of the specified interval for x. In this example, a zero-slope minimum was found, so the true minimum at the boundary was missed.

6.6 Plotting Functions

As described previously, a function such as a polynomial can be evaluated and then plotted with the `plot` command. This can also be done in one step with the `fplot` command.

`fplot(fun,lims)` plots the function specified by the string `fun` between the x-axis limits specified by `lims = [xmin xmax]`. Using `lims = [xmin xmax ymin ymax]` also controls the y-axis limits. `fun` must be the name of an M-file function or a string with variable `x`, such as `sin(x)`, `diric(x,10)` or `[sin(x),cos(x)]`. The function `fun(x)` must return a row vector for each element of vector `x`.

Consider the plotting the following function, known as the *sinc* or *sampling* function:

$$\text{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$$

This is a function available in Matlab, which can be plotted with the command:

```
fplot('sinc',[-10 10]),ylabel('sinc(x)'),xlabel('x')
```

This generates the plot shown in Figure 6.15

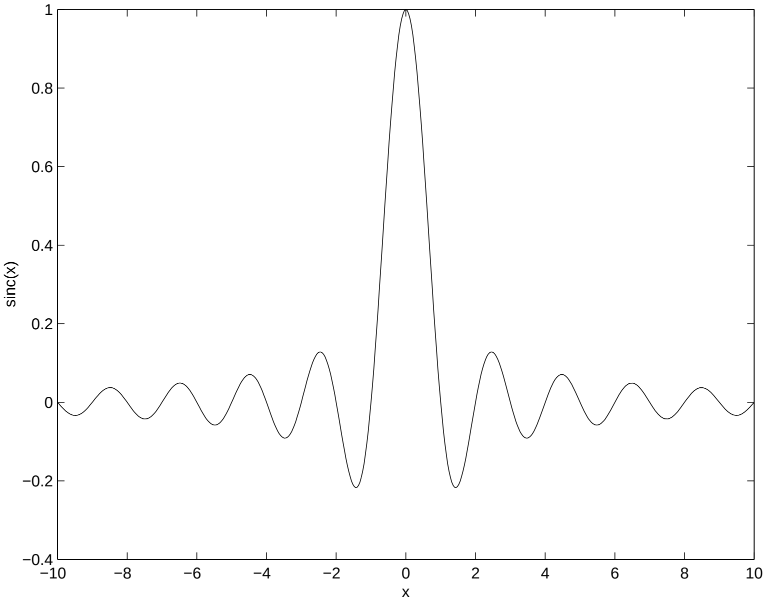


Figure 6.15: Sinc signal

Section 7

Data Analysis

Outline

- Maximum and minimum
- Sums and products
- Statistical analysis
- Random number generation

The purpose of data analysis is to determine some properties and characteristics of the data, which may contain measurement errors or other random influences.

For the examples to be considered in this section, generate, save, and plot the two data vectors, data1 and data2 as shown below. The commands used to generate this data will be explained later. The two data vectors, data_1 and data_2 are plotted in Figure 7.1.

```
data1 = 2*rand(1,500) + 2;
data2 = randn(1,500)+3;
save data
subplot(2,1,1),plot(data1),...
  axis([0 500 0 6]),...
  title('Random Numbers - data1'),...
subplot(2,1,2),plot(data2),...
  title('Random Numbers - data2'),...
  xlabel('Index, k')
```

Other data analysis functions available in MATLAB are described with the command help datafun.

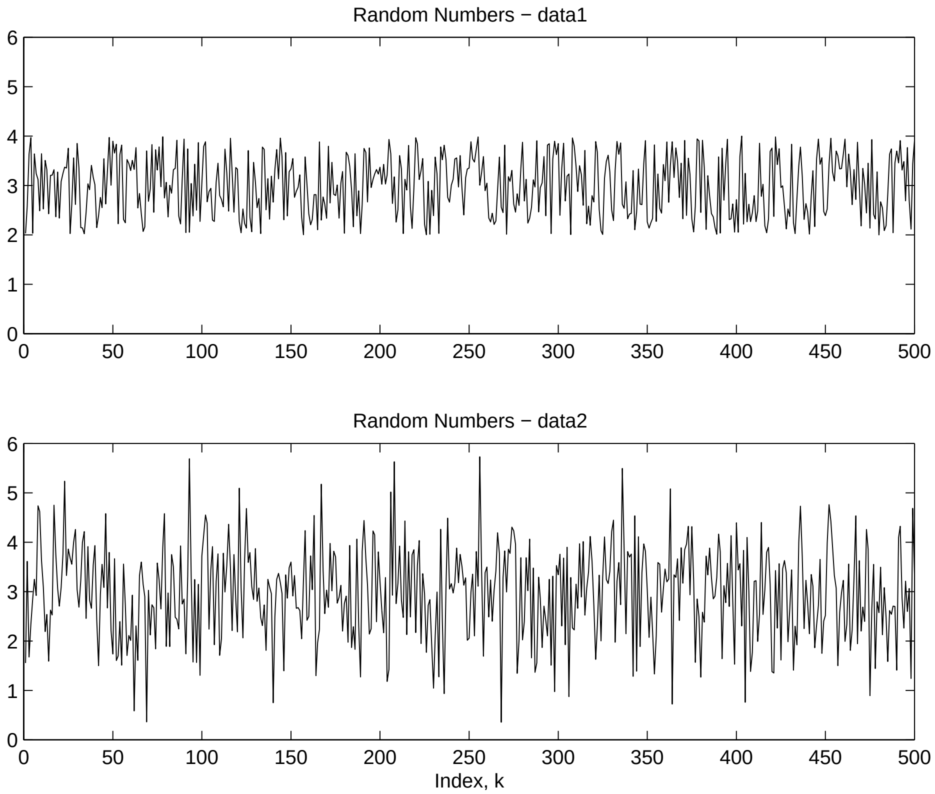


Figure 7.1: Random sequences

7.1 Maximum and Minimum

`max(x)`	Returns largest value in vector `x`, or the row vector of largest elements of each column in matrix `x`
`[y,k] = max(x)`	Returns maximum values `y` and corresponding indices `k` of the first maximum value from each column of `x`.
`max(x,y)`	Returns a matrix the same size as `x` and `y`, with each element being the maximum value from the corresponding positions in `x` and `y`.
`min(x)`	Returns smallest value in vector `x`, or the row vector of smallest elements of each column in matrix `x`
`[y,k] = min(x)`	Returns minimum values `y` and corresponding indices `k` of first minimum value from each column of `x`.
`min(x,y)`	Returns a matrix the same size as `x` and `y`, with each element being the minimum value from the corresponding positions in `x` and `y`.

Determining the maximum and minimum of a vector:

```
>> v = [3 -2 4 -1 5 0];
>> max(v)
ans =
     5
>> [vmin, kmin] = min(v)
vmin =
    -2
kmin =
     2
```

The maximum value is found to be 5, the minimum value -2, and the index of the minimum value is 2. Thus, `vmin = v(kmin) = v(2) = -2`.

For the random data vectors `data1` and `data2`:

```
>> max(data1)
ans =
    3.9985
>> min(data1)
ans =
    2.0005
>> [max2,kmax2] = max(data2)
max2 =
    5.7316
kmax2 =
   256
>> [min2,kmin2] = min(data2)
min2 =
    0.3558
kmin2 =
   268
```

These results can be approximately confirmed from Figure 7.1.

For a matrix, the `min` and `max` functions return a row vector of the minimum or maximum elements of each column of the matrix.

```
>> B = [-1 1 7 0; -3 5 5 8; 1 4 4 -8]
B =
    -1     1     7     0
    -3     5     5     8
     1     4     4    -8
>> min(B)
ans =
    -3     1     4    -8
>> max(B)
ans =
     1     5     7     8
```

Example 7.1 *Minimum cost tank design*

A cylindrical tank with a hemispherical top, as shown in Figure 7.2, is to be constructed that will hold 5.00×10^5L when filled. Determine the tank radius R and height H to minimize the tank cost if the cylindrical portion costs \$300/m^2 of surface area and the hemispherical portion costs \$400/m^2.

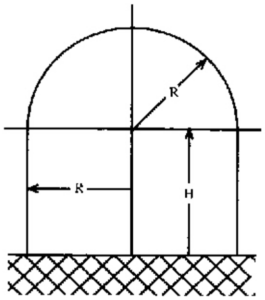


Figure 7.2: Tank configuration

Mathematical model:

Cylinder volume: $V_c = \pi R^2 H$
Hemisphere volume: $V_h = \frac{2}{3}\pi R^3$
Cylinder surface area: $A_c = 2\pi RH$
Hemisphere surface area: $A_h = 2\pi R^2$

Assumptions:

Tank contains no dead air space.
Concrete slab with hermetic seal is provided for the base.
Cost of the base does not change appreciably with tank dimensions.

Computational method:

Express total volume in meters cubed (note: $1m^3 = 1000L$) as a function of height and radius

$$V_{tank} = V_c + V_h$$

For $V_{tank} = 5 \times 10^5 L = 500m^3$:

$$500 = \pi R^2 H + \frac{2}{3}\pi R^3$$

Solving for H:

$$H = \frac{500}{\pi R^2} - \frac{2R}{3}$$

Express cost in dollars as a function of height and radius

$$C = 300A_c + 400A_h = 300(2\pi RH) + 400(2\pi R^2)$$

Method: compute H and then C for a range of values of R, then find the minimum value of C and the corresponding values of R and H.

To determine the range of R to investigate, make an approximation by assuming that $H = R$. Then from the tank volume:

$$V_{tank} = 500 = \pi R^3 + \frac{2}{3}\pi R^3 = \frac{5}{3}\pi R^3$$

Solving for R:

$$R = \left(\frac{300}{\pi}\right)^{\frac{1}{3}}$$

From Matlab:

```
>> Rest = (300/pi)^(1/3)
Rest =
    4.5708
```

We will investigate R in the range 3.0 to 7.0 meters.

Computational implementation:

Matlab script to determine the minimum cost design:

```
% Tank design problem
%
% Compute H & C as functions of R
R = 3:0.001:7.0;                       % Generate trial radius values R
H = 500./(pi*R.^2) - 2*R/3;            % Height H
C = 300*2*pi*R.*H + 400*2*pi*R.^2;     % Cost

% Plot cost vs radius
plot(R,C),title('Tank Design'),...
  xlabel('Radius R, m'),...
  ylabel('Cost C, Dollars'),grid

% Compute and display minimum cost, corresponding H & R
[Cmin kmin] = min(C);
disp('Minimum cost (dollars):')
disp(Cmin)
disp('Radius R for minimum cost (m):')
disp(R(kmin))
disp('Height H for minimum cost (m):')
disp(H(kmin))
```

Running the script:

```
Minimum cost (dollars):
  9.1394e+004
Radius R for minimum cost (m):
    4.9240
Height H for minimum cost (m):
    3.2816
```

Note that the radius corresponding to minimum cost (`Rmin = 4.9240` is close to the approximate value, `Rest = 4.5708` that we computed to assist in the selection of a range of R to investigate. The plot of cost C versus radius R is shown in Figure 7.3.

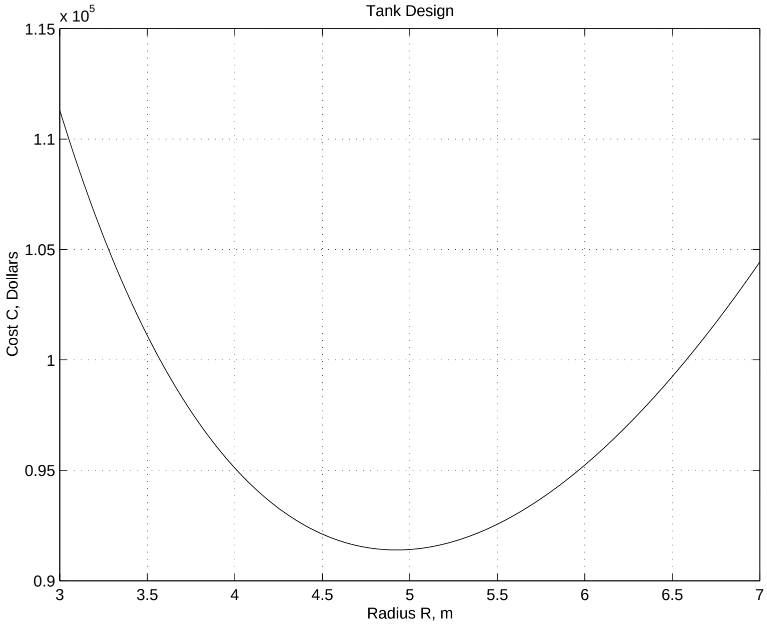


Figure 7.3: Tank design problem: cost versus radius

■

7.2 Sums and Products

If **x** is a vector with N elements denoted $x(n), \;\; n = 1, 2, \ldots, N$, then:

- The **sum** y is the scalar

$$y = \sum_{n=1}^{N} x(n) = x(1) + x(2) + \cdots + x(N)$$

- The **product** y is the scalar

$$y = \prod_{n=1}^{N} x(n) = x(1) \cdot x(2) \cdots x(N)$$

- The **cumulative sum y** is the vector having elements $y(k), \quad k = 1, 2, \ldots, N$

$$y(k) = \sum_{n=1}^{k} x(n) = x(1) + x(2) + \cdots + x(k)$$

- The **cumulative product y** is the vector having elements $y(k), \quad k = 1, 2, \ldots, N$

$$y(k) = \prod_{n=1}^{k} x(n) = x(1) \cdot x(2) \cdots x(k)$$

If **X** is a matrix with M rows and N columns with elements denoted $x(m, n), \quad m = 1, 2, \ldots, M, \quad n = 1, 2, \ldots, N$, then

- The **column sum y** is the vector having elements $y(n), \quad n = 1, 2, \ldots, N$

$$y(n) = \sum_{m=1}^{M} x(m, n) = x(1, n) + x(2, n) + \cdots + x(M, n)$$

- The **column product y** is the vector having elements $y(n), \quad n = 1, 2, \ldots, N$

$$y(n) = \prod_{m=1}^{M} x(m, n) = x(1, n) \cdot x(2, n) \cdots x(M, n)$$

- The **cumulative column sum Y** is the matrix with M rows and N columns with elements denoted $y(k, n), \quad k = 1, 2, \ldots, M, \quad n = 1, 2, \ldots, N$

$$y(k, n) = \sum_{m=1}^{k} x(m, n) = x(1, n) + x(2, n) + \cdots + x(k, n)$$

- The **cumulative column product Y** is the matrix with M rows and N columns with elements denoted $y(k, n), \quad k = 1, 2, \ldots, M, \quad n = 1, 2, \ldots, N$

$$y(k, n) = \prod_{m=1}^{k} x(m, n) = x(1, n) \cdot x(2, n) \cdots x(k, n)$$

The Matlab functions implementing these mathematical functions are the following.

`sum(x)` Returns the sum of the elements in vector `x`, or the row vector of the sum of elements of each column in matrix `x`

`prod(x)` Returns the product of the elements in vector `x`, or the row vector of the product of elements of each column in matrix `x`

`cumsum(x)` Returns a vector the same size as `x` containing the cumulative sum of the elements in vector `x`, or a matrix the same size as `x` containing cumulative sums of values from the columns of `x`.

`cumprod(x)` Returns a vector the same size as `x` containing the cumulative product of the elements in vector `x`, or a matrix the same size as `x` containing cumulative products of values from the columns of `x`.

Example 7.2 *Summation*

The following is a summation identity

$$\sum_{n=1}^{N} n = \frac{N(N+1)}{2}$$

This identity can be checked with MATLAB, for example, for $N = 8$:

```
>> N = 8;
>> n =1:8;
>> S = sum(n)
S =
    36
>> N*(N+1)/2
ans =
    36
```

■

Example 7.3 *Factorial*

The **factorial** of n is expressed and defined as

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 1$$

This can be computed as `prod(1:n)`, as in the following examples:

```
>> nfact = prod(1:4)
nfact =
    24
>> nfact = prod(1:70)
nfact =
  1.1979e+100
```

■

7.3 Statistical Analysis

Many engineering measurements have an element of randomness. Statistics is a branch of applied mathematics involving the analysis, interpretation, and presentation of data including some degree of randomness or uncertainty. In *descriptive statistics,* the important features or properties of a set of data are summarized or described.

Continuous random variable: A variable vector $\mathbf{x}$, whose elements can have any of a continuum of values for each measurement, or observation.

Population: all possible measurements of a random variable

Sample: finite number of measurements. Assume that N measurements have been made of a random variable, denoted as the vector $\mathbf{x}$, with elements $x(n), \quad n = 1, \ldots, N$.

Frequency distribution of a random variable: indicates the relative frequency with which specific values of this random variable occur in the population.

Histogram: frequency distribution of sample data, indicating the frequency with which sample values fall within ranges of values, called **bins**.

- Range of values: $x_{min} \leq x \leq x_{max}$
- Number of bins: m
- Bin width: $\Delta x = \dfrac{x_{max} - x_{min}}{m}$
- Histogram value in bin k: $N(k)$, the number of samples with value $x_{min} + (k-1)\Delta x \leq x \leq x_{min} + k\Delta x$. This is also known as the **absolute frequency**.

The Matlab commands for generating and plotting a histogram are;

Command	Description
`N = hist(x)`	Returns the row vector histogram `N` of the values in `x` using 10 bins. If `x` is matrix, returns a histogram matrix `N` operating on the columns of `x`.
`N = hist(x,m)`	Returns the row vector histogram `N` of the values in `x` using `m` bins.
`N = hist(x,xc)`	Returns the row vector histogram `N` of the values in `x` using bin centers specified by `xc`.
`N = histc(x,edges)`	Counts the number of values in vector `x` that fall between the elements in the `edges` vector (which must contain monotonically non-decreasing values). `N` is a `length(edges)` vector containing these counts. `N(k)` will count the value `x(i)` if `edges(k) >= x(i) > edges(k+1)`. The last bin will count any values of `x` that match `edges(end)`. Values outside the values in `edges` are not counted. Use `-inf` and `inf` in `edges` to include all non-`NaN` values.
`[N,xc] = hist(...)`	Also returns the position of the bin centers in `xc`.
`hist (...)`	Without output arguments produces a histogram bar plot of the results.

For example, the following commands produce the 25-bin histograms shown in Figure 7.4, operating on the random data vectors `data1` and `data2`.

```
% Histogram plots
%
subplot(2,1,1),hist(data1,25),...
  title(’Histogram of data1’),...
  xlabel(’x’),...
  ylabel(’N’),...
subplot(2,1,2),hist(data2,25),...
  title(’Histogram of data2’),...
  xlabel(’x’),...
  ylabel(’N’)
```

Note the differences in the nature of the two distributions. Random data `data1` is called a **uniform distribution**, as it has roughly equal, or uniform, frequency in all bins. Random data `data2` has a *bell-shaped* distribution, called a **Gaussian distribution** or **Normal distribution**.

Relative frequency histogram: The histogram value in bin k is normalized by the total number, N, of data samples: $N(k)/N$.

The `hist` function is limited in its ability to produce relative frequency histograms. It is better to generate such plots with the `bar` function, defined as follows:

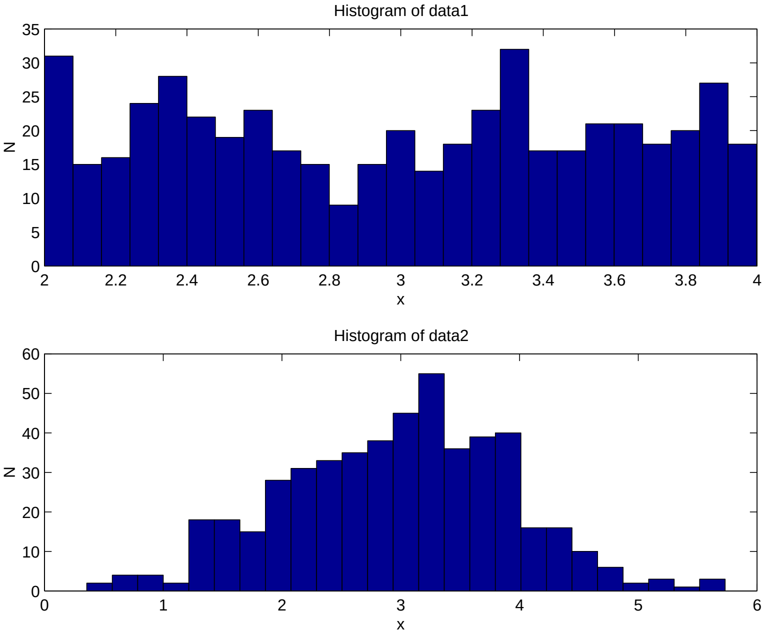


Figure 7.4: Absolute frequency histograms

`bar(X,Y)` Draws the columns of the M-by-N matrix `Y` as M groups of N vertical bars. The vector `X` must be monotonically increasing or decreasing.

`bar(Y)` Uses the default value of `X=1:M`.

`bar(x,y` For vector inputs, `length(y)` bars are drawn, with each bar centered on a value of `x`.

`bar(X,Y,width)` Specifies the width of the bars. Values of `width` ¿ 1 produce overlapped bars. The default value is `width`=0.8

For example, the following commands produce relative histogram plots shown in Figure 7.5 for the random data vectors `data1` and `data2`.

```
% Relative frequency histogram plots
%
n1 = length(data1);
[freq1,x1] = hist(data1,25);
rfreq1 = freq1/n1;
n2 = length(data2);
[freq2,x2] = hist(data2,25);
rfreq2 = freq2/n2;

subplot(2,1,1),bar(x1,rfreq1),...
  title('Relative histogram of data1'), grid,...
  xlabel('x'), ylabel('relative frequency'),...
subplot(2,1,2),bar(x2,rfreq2),...
  title('Relative histogram of data2'), grid,...
  xlabel('x'), ylabel('relative frequency')
```

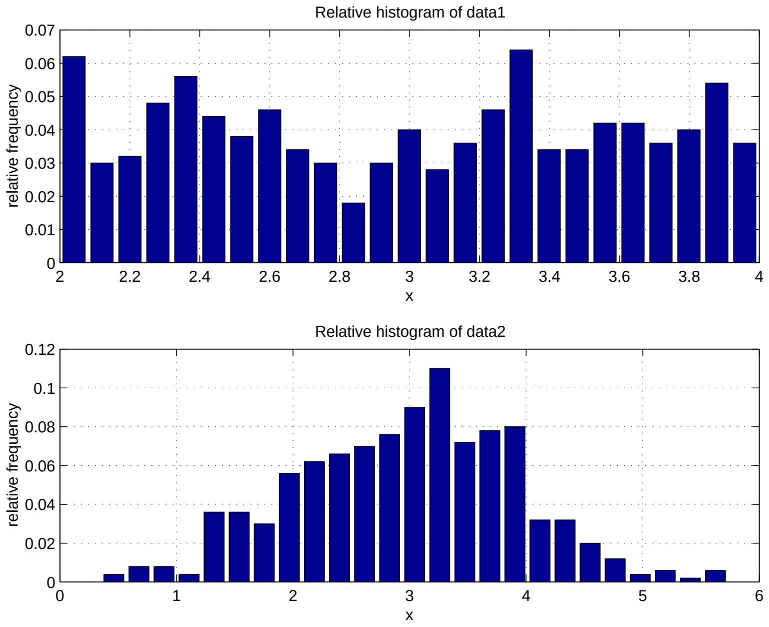


Figure 7.5: Relative frequency histograms

Measures of central tendency

The **mean** and **median** describe the middle, or center, of the range of the random variable.

The **sample mean** of vector **x** having N elements is $\bar{x}$

$$\bar{x} = \frac{1}{N}\sum_{n=1}^{N} x(m)$$

The **population mean** μ (mu) is the value of $\bar{x}$ for an infinite number of measurements, denoted by

$$\mu = \lim_{N\to\infty} \bar{x}$$

where the right hand side is read as "the limit of x bar as N approaches infinity."

Median: Middle value of a set of random samples, sorted by value (rank ordered). If there is an even number of samples, then the median is the average of the two middle sample values.

Command	Description
`mean(x)`	Returns the sample mean of the elements of the vector `x`. Returns a row vector of the sample means of the columns of matrix `x`.
`median(x)`	Returns the median of the elements of the vector `x`. Returns a row vector of the medians of the columns of matrix `x`.
`sort(x)`	Returns a vector with the values of vector `x` in ascending order. Returns a matrix with each column of matrix `x` in ascending order.

For example:

```
>> mean(data1)
ans =
    2.9940
>> sum(data1)/length(data1)
ans =
    2.9940
>> mean(data2)
ans =
    2.9768
>> median(data1)
ans =
    3.0285
>> median(data2)
ans =
    3.0574
```

These results can be confirmed by observing the plots of the data in Figure 7.1, in which the data values can be seen to be centered on 3.0. For the two data sets considered, the values of the mean and median are very close. This is not necessarily the case for other data sets. Also note that the results above show that `mean(data1) = sum(data1)/length(data1)`.

Measures of variation

Measures of variation: indicate the degree of deviation of random samples from the measure of central tendency.

Referring again to the plots of our random data sets `data1` and `data2` in Figure 7.1, observe that `data2` has greater variation from the mean.

The **sample standard deviation** of vector $\mathbf{x}$ having N elements is

$$s = \left[\frac{1}{N-1} \sum_{n=1}^{N} (x(n) - \bar{x})^2 \right]^{\frac{1}{2}}$$

The **sample variance** s^2 is the square of the standard deviation.

`std(x)` Returns the sample standard deviation of the elements of the vector `x`. Returns a row vector of the sample standard deviations of the columns of matrix `x`.

For example:

```
>> std(data1)
```

```
ans =
    0.5989
>> std(data2)
ans =
    0.9408
```

Thus, the variation of `data2` is greater than that of `data1`, as we concluded from observation of the plotted data values.

7.4 Random Number Generation

Many engineering problems require the use of **random numbers** in the development of a solution. In some cases, the random numbers are used to develop a simulation of a complex problem. The simulation can be tested over and over to analyze the results, with each test representing a repetition of the experiment. Random numbers also used to represent noise sequences, such as those heard on a radio.

Uniform Random Numbers

Random numbers are characterized by their frequency distributions. **Uniform random numbers** have a constant or uniform distribution over their range between minimum and maximum values.

The `rand` function in MATLAB generates uniform random numbers distributed over the interval [0,1]. A *state vector* is used to generate a random sequence of values. A given state vector always generates the same random sequence. Thus, the sequences are known more formally as **pseudorandom sequences**. This generator can generate all the floating point numbers in the closed interval [2^{-53}, $1 - 2^{-53}$]. Theoretically, it can generate over 2^{1492} values before repeating itself.

Command	Description
`rand(n)`	Returns an `n`×`n` matrix `x` of random numbers distributed uniformly in the interval [0,1].
`rand(m,n)`	Returns an `m`×`n` matrix `x` of random numbers distributed uniformly in the interval [0,1].
`rand`	Returns a scalar whose value changes each time it is referenced. `rand(size(A))` is the same size as A.
`s = rand('state')`	Returns a 35-element vector `s` containing the current state of the uniform generator.
`rand('state',s)`	Resets the state to `s`.
`rand('state',0)`	Resets the generator to its initial state.
`rand('state',J)`	Resets the generator to its `J`-th state for integer `J`.
`rand('state',sum(100*clock))`	Resets the generator to a different state each time.

The following commands generate and display two sets of ten random numbers uniformly distributed between 0 and 1; the difference between the two sets is caused by the different states.

```
rand('state',0)
```

```
set1 = rand(10,1);
rand('state',123)
set2 = rand(10,1);
[set1 set2]
```

The results displayed by these commands are the following, where the first column gives values of `set1` and the second column gives values of `set2`:

```
    0.9501    0.0697
    0.2311    0.2332
    0.6068    0.7374
    0.4860    0.7585
    0.8913    0.6368
    0.7621    0.6129
    0.4565    0.3081
    0.0185    0.2856
    0.8214    0.0781
    0.4447    0.9532
```

Note that as desired, the two sequences are different.

To generate sequences having values in the interval $[x_{min}, x_{max}]$, first generate a random number r in the interval $[0, 1]$, multiply by $(x_{max} - x_{min})$, producing a number in the interval $[0, (x_{max} - x_{min}]$, then add x_{min}. To generate a row vector of 500 elements, the command needed is

```
x = (xmax - xmin)*rand(1,500) + xmin
```

The sequence `data1`, plotted in Figure 7.1, was generated with the command:

```
data1 = 2*rand(1,500) + 2;
```

Thus, $x_{max} - x_{min} = 2$ and $x_{min} = 2$, so $x_{max} = 4$ and the range of the data sequence is (2,4), which can be confirmed from Figure 7.1.

Example 7.4 *Flipping a coin*

When a fair coin is flipped, the probability of getting heads or tails is 0.5 (50%). If we want to represent tails by 0 and heads by 1, we can first generate a uniform random number in the range [0,2). The probability of a result in the range [0,1) is 0.5 (50%), which can be mapped to 0 by truncation of this result. Similarly, the probability of a result in the range [1,2) is 0.5 (50%), which can be mapped to 1 by truncation. The trunction function in Matlab is `floor`.

A script to simulate an experiment to flip a coin 50 times and display the results is the following:

```
% Coin flipping simulation
%
% Coin flip:
n = 50;                       % number of flips
coin = floor(2*rand(1,n))     % vector of n flips: 0: tails, 1:heads

% Histogram computation and display:
xc = [0 1];                   % histogram centers
y = hist(coin,xc);            % absolute frequency
bar(xc,y), ylabel('frequency'),...
   xlabel('0: tails, 1: heads'),...
   title('Absolute Frequency Histogram for 50 Coin Flips')
```

The output displayed by the script:

```
coin =
  Columns 1 through 12
     1     1     0     0     1     0     1     1     1     0     1     1
  Columns 13 through 24
     1     0     0     0     0     1     1     0     1     0     1     1
  Columns 25 through 36
     0     1     1     1     0     0     0     0     0     0     1     0
  Columns 37 through 48
     0     1     0     1     0     1     1     0     1     1     0     0
  Columns 49 through 50
     0     0
```

The histogram plot is shown in Figure 7.6. Note that in this trial, there were 26 tails and 24 heads, close to the expected result. Since this is a random process, repeated trials will give different results.

■

Example 7.5 *Rolling dice*

When a fair die (singular of dice) is rolled, the number uppermost is equally likely to be any integer from 1 to 6. The following statement will generate a vector of 10 random numbers in the range 1-6:

```
die = floor( 6 * rand(1,10) + 1 )
```

The following are the results of two such simulations:

```
     2     3     4     5     2     1     3     3     1     4
     5     2     2     5     5     6     3     6     3     5
```

■

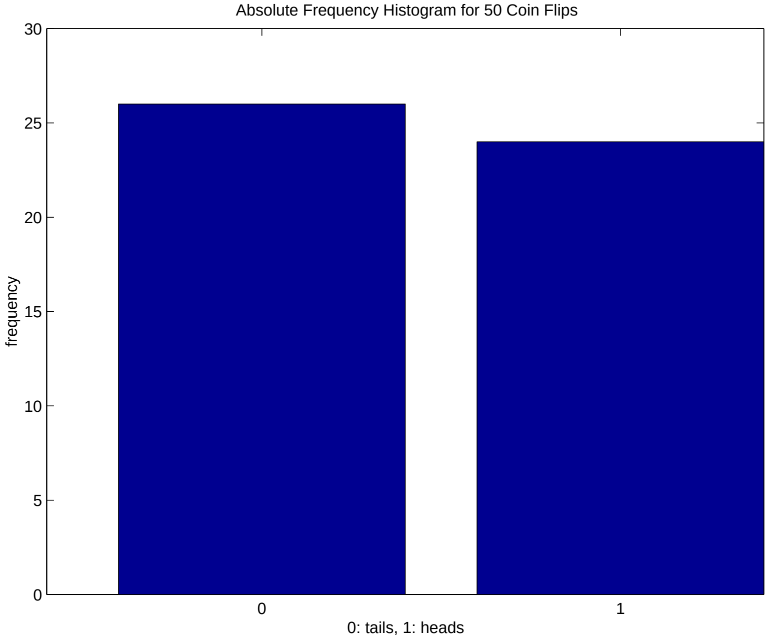


Figure 7.6: Histogram of 50 Coin Flips

Gaussian Random Numbers

A random distribution that is an appropriate model for data sets from many engineering problems is the **Gaussian** or **Normal** distribution. This is the distribution having a bell shape, with a high relative frequency at the mean, decreasing away from the mean, but extending indefinitely in both directions. The Gaussian distribution is specified by two parameters: (1) the population mean μ, and (2) the population standard deviation σ, with the distribution function given by

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/2\sigma^2}$$

This distribution is plotted in Figure 7.7, where it is observed to be symmetrical about the mean, dropping off rapidly away from the mean. The distribution can be normalized by performing a change of scale that converts the units of measurement into standard units

$$z = \frac{x-\mu}{\sigma}$$

as shown in Figure 7.7. It can be shown that approximately 68% of the values will fall within one standard deviation of the mean ($-1 < z < 1$), 95% will fall within two standard deviations of the mean ($-2 < z < 2$), and 99% will fall within three standard deviations of the mean ($-3 < z < 3$).

The Matlab functions for generating Gaussian random numbers are:

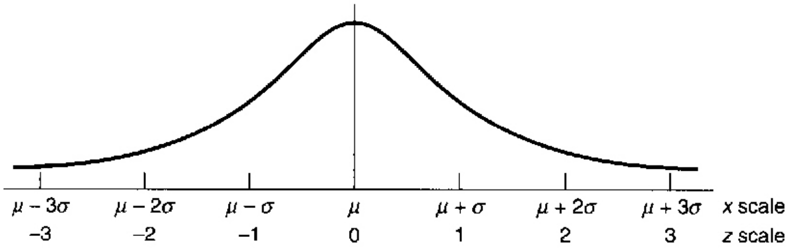


Figure 7.7: Normalized Gaussian distribution

Command	Description
`randn(n)`	Returns an `n`×`n` matrix `x` of Gaussian random numbers with a mean of 0 and a standard deviation of 1.
`randn(m,n)`	Returns an `m`×`n` matrix `x` of Gaussian random numbers with a mean of 0 and a standard deviation of 1.
`randn`	Returns a scalar whose value changes each time it is referenced. `rand(size(A))` is the same size as `A`.
`s = randn('state')`	Returns a 2-element vector `s` containing the current state of the uniform generator.
`randn('state',s)`	Resets the state to `s`.
`randn('state',0)`	Resets the generator to its initial state.
`randn('state',J)`	Resets the generator to its `J`-th state for integer `J`.
`randn('state',sum(100*clock))`	Resets the generator to a different state each time.

To generate a row vector `x` of 500 Gaussian random variables with mean m and standard deviation s:

```
x = m + s*randn(1,500);
```

The sequence `data2`, plotted in Figure 7.1, was generated with the command:

```
data2 = rand(1,500) + 3;
```

Thus, $m = 3$ and $s = 1$. The mean value can be confirmed from the plot in Figure 7.1 and the mean value computed in Section 7.3. The standard deviation can be confirmed by the value computed in Section 7.3. The computed values are not identical to the desired values because the computed values are due to a finite number of random samples. The computed and desired values become closer as the number of samples increases.

Example 7.6 *Noisy signal simulation*

Many sensors, which convert physical quantities into electrical signals, produce measurement errors, resulting in what is known as noisy signals. Additive Gaussian random numbers are often an accurate model of the errors. The signal model is

$$x = s + n$$

where s is the noise-free signal of interest, n is the Gaussian measurement error, and x is the noisy signal. A measure of the signal quality is the *signal to noise ratio* (SNR), defined by

$$\begin{aligned}\text{SNR} &= 10\log_{10}\frac{\sigma_s^2}{\sigma_n^2}\\ &= 20\log_{10}\frac{\sigma_s}{\sigma_n}\end{aligned}$$

where σ_s is the standard deviation of the signal and σ_n is the standard deviation of the noise.

Consider simulating a noisy signal consisting of a sine wave with additive Gaussian measurement noise. The variance of a sine wave can be shown to be

$$\sigma_s^2 = \frac{A}{2}$$

where A is the sinusoidal amplitude. For $A = 1$ and Gaussian noise standard deviation $\sigma_n = 0.1$, the SNR is

$$\text{SNR} = 10\log_{10}\frac{0.5}{0.1^2} = 10\log_{10}50 = 17.0$$

The following script simulates this noisy signal. In the next section, methods to reduce the affect of the noise will be considered.

```
% Noisy signal generation and analysis
t = linspace(0,10,512);             % time base
s = sin(2*pi/5*t);                  % signal
n = 0.1*randn(size(t));             % noise
x = s + n;                          % noisy signal
disp('Signal to Noise Ratio (SNR), dB')
SNR = 20*log10(std(s)/std(n))       % signal to noise ratio, dB
plot(t,x), xlabel('Time (s)'),...
  ylabel('Signal amplitude'),...
  title('Noisy signal')
```

The computed SNR is

```
Signal to Noise Ratio (SNR), dB
SNR =
   16.9456
```

This is very close to the value computed above. The reason for the difference is the finite number of samples used to estimate the noise standard deviation. The resulting noise signal is shown in Figure 7.8.

∎

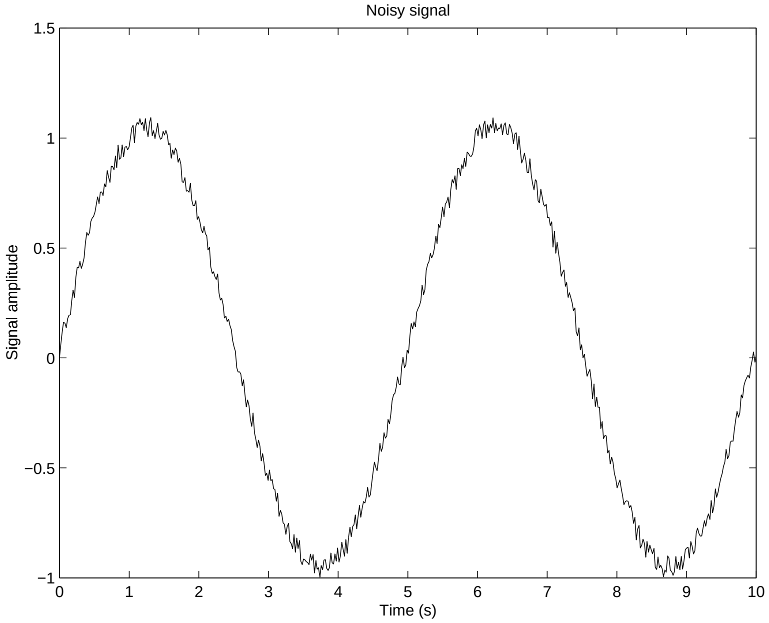


Figure 7.8: Simulated noisy sinusoidal signal

Section 7

Data Analysis

Outline

- Maximum and minimum
- Sums and products
- Statistical analysis
- Random number generation

The purpose of data analysis is to determine some properties and characteristics of the data, which may contain measurement errors or other random influences.

For the examples to be considered in this section, generate, save, and plot the two data vectors, `data1` and `data2` as shown below. The commands used to generate this data will be explained later. The two data vectors, `data_1` and `data_2` are plotted in Figure 7.1.

```
data1 = 2*rand(1,500) + 2;
data2 = randn(1,500)+3;
save data
subplot(2,1,1),plot(data1),...
  axis([0 500 0 6]),...
  title('Random Numbers - data1'),...
subplot(2,1,2),plot(data2),...
  title('Random Numbers - data2'),...
  xlabel('Index, k')
```

Other data analysis functions available in MATLAB are described with the command `help datafun`.

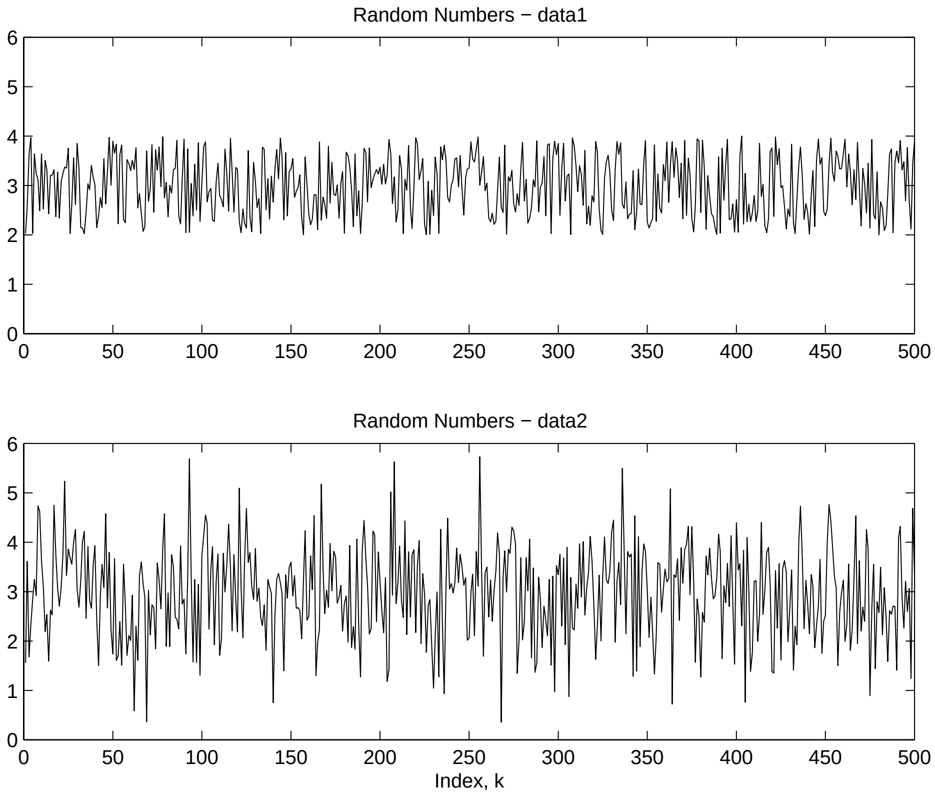


Figure 7.1: Random sequences

7.1 Maximum and Minimum

`max(x)`	Returns largest value in vector `x`, or the row vector of largest elements of each column in matrix `x`
`[y,k] = max(x)`	Returns maximum values `y` and corresponding indices `k` of the first maximum value from each column of `x`.
`max(x,y)`	Returns a matrix the same size as `x` and `y`, with each element being the maximum value from the corresponding positions in `x` and `y`.
`min(x)`	Returns smallest value in vector `x`, or the row vector of smallest elements of each column in matrix `x`
`[y,k] = min(x)`	Returns minimum values `y` and corresponding indices `k` of first minimum value from each column of `x`.
`min(x,y)`	Returns a matrix the same size as `x` and `y`, with each element being the minimum value from the corresponding positions in `x` and `y`.

Determining the maximum and minimum of a vector:

```
>> v = [3 -2 4 -1 5 0];
>> max(v)
ans =
     5
>> [vmin, kmin] = min(v)
vmin =
    -2
kmin =
     2
```

The maximum value is found to be 5, the minimum value -2, and the index of the minimum value is 2. Thus, `vmin = v(kmin) = v(2) = -2`.

For the random data vectors `data1` and `data2`:

```
>> max(data1)
ans =
    3.9985
>> min(data1)
ans =
    2.0005
>> [max2,kmax2] = max(data2)
max2 =
    5.7316
kmax2 =
   256
>> [min2,kmin2] = min(data2)
min2 =
    0.3558
kmin2 =
   268
```

These results can be approximately confirmed from Figure 7.1.

For a matrix, the `min` and `max` functions return a row vector of the minimum or maximum elements of each column of the matrix.

```
>> B = [-1 1 7 0; -3 5 5 8; 1 4 4 -8]
B =
    -1     1     7     0
    -3     5     5     8
     1     4     4    -8
>> min(B)
ans =
    -3     1     4    -8
>> max(B)
ans =
     1     5     7     8
```

Example 7.1 *Minimum cost tank design*

A cylindrical tank with a hemispherical top, as shown in Figure 7.2, is to be constructed that will hold 5.00×10^5L when filled. Determine the tank radius R and height H to minimize the tank cost if the cylindrical portion costs \$300/m^2 of surface area and the hemispherical portion costs \$400/m^2.

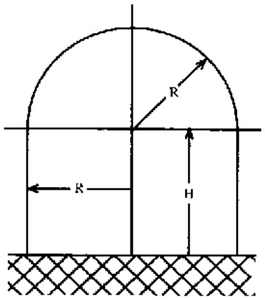


Figure 7.2: Tank configuration

Mathematical model:

Cylinder volume: $V_c = \pi R^2 H$
Hemisphere volume: $V_h = \frac{2}{3}\pi R^3$
Cylinder surface area: $A_c = 2\pi RH$
Hemisphere surface area: $A_h = 2\pi R^2$

Assumptions:

Tank contains no dead air space.
Concrete slab with hermetic seal is provided for the base.
Cost of the base does not change appreciably with tank dimensions.

Computational method:

Express total volume in meters cubed (note: $1m^3 = 1000L$) as a function of height and radius

$$V_{tank} = V_c + V_h$$

For $V_{tank} = 5 \times 10^5 L = 500m^3$:

$$500 = \pi R^2 H + \frac{2}{3}\pi R^3$$

Solving for H:

$$H = \frac{500}{\pi R^2} - \frac{2R}{3}$$

Express cost in dollars as a function of height and radius

$$C = 300A_c + 400A_h = 300(2\pi RH) + 400(2\pi R^2)$$

Method: compute H and then C for a range of values of R, then find the minimum value of C and the corresponding values of R and H.

To determine the range of R to investigate, make an approximation by assuming that $H = R$. Then from the tank volume:

$$V_{tank} = 500 = \pi R^3 + \frac{2}{3}\pi R^3 = \frac{5}{3}\pi R^3$$

Solving for R:

$$R = \left(\frac{300}{\pi}\right)^{\frac{1}{3}}$$

From Matlab:

```
>> Rest = (300/pi)^(1/3)
Rest =
    4.5708
```

We will investigate R in the range 3.0 to 7.0 meters.

Computational implementation:

Matlab script to determine the minimum cost design:

```
% Tank design problem
%
% Compute H & C as functions of R
R = 3:0.001:7.0;                       % Generate trial radius values R
H = 500./(pi*R.^2) - 2*R/3;            % Height H
C = 300*2*pi*R.*H + 400*2*pi*R.^2;     % Cost

% Plot cost vs radius
plot(R,C),title('Tank Design'),...
  xlabel('Radius R, m'),...
  ylabel('Cost C, Dollars'),grid

% Compute and display minimum cost, corresponding H & R
[Cmin kmin] = min(C);
disp('Minimum cost (dollars):')
disp(Cmin)
disp('Radius R for minimum cost (m):')
disp(R(kmin))
disp('Height H for minimum cost (m):')
disp(H(kmin))
```

Running the script:

```
Minimum cost (dollars):
  9.1394e+004
Radius R for minimum cost (m):
    4.9240
Height H for minimum cost (m):
    3.2816
```

Note that the radius corresponding to minimum cost (`Rmin = 4.9240` is close to the approximate value, `Rest = 4.5708` that we computed to assist in the selection of a range of R to investigate. The plot of cost C versus radius R is shown in Figure 7.3.

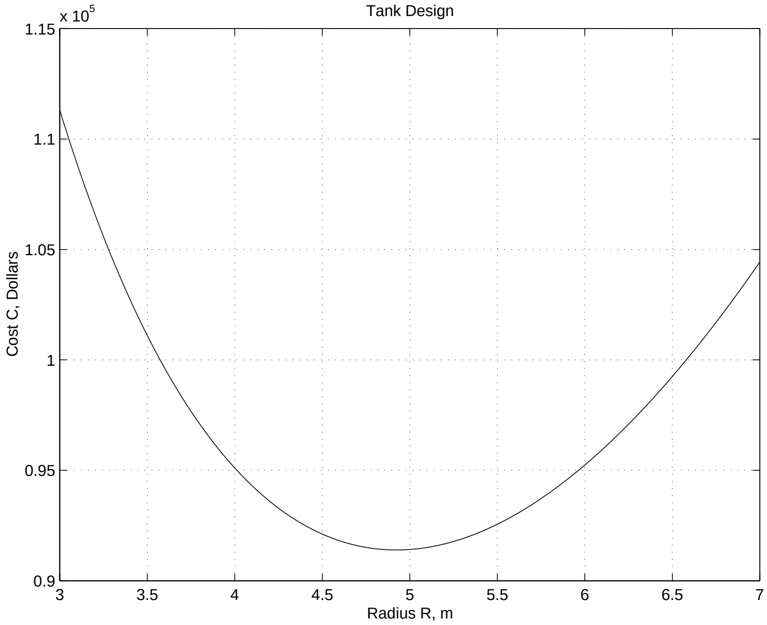


Figure 7.3: Tank design problem: cost versus radius

■

7.2 Sums and Products

If $\mathbf{x}$ is a vector with N elements denoted $x(n), \;\; n = 1, 2, \ldots, N$, then:

- The **sum** y is the scalar

$$y = \sum_{n=1}^{N} x(n) = x(1) + x(2) + \cdots + x(N)$$

- The **product** y is the scalar

$$y = \prod_{n=1}^{N} x(n) = x(1) \cdot x(2) \cdots x(N)$$

- The **cumulative sum y** is the vector having elements $y(k), \quad k = 1, 2, \ldots, N$

$$y(k) = \sum_{n=1}^{k} x(n) = x(1) + x(2) + \cdots + x(k)$$

- The **cumulative product y** is the vector having elements $y(k), \quad k = 1, 2, \ldots, N$

$$y(k) = \prod_{n=1}^{k} x(n) = x(1) \cdot x(2) \cdots x(k)$$

If **X** is a matrix with M rows and N columns with elements denoted $x(m, n), \quad m = 1, 2, \ldots, M, \quad n = 1, 2, \ldots, N$, then

- The **column sum y** is the vector having elements $y(n), \quad n = 1, 2, \ldots, N$

$$y(n) = \sum_{m=1}^{M} x(m, n) = x(1, n) + x(2, n) + \cdots + x(M, n)$$

- The **column product y** is the vector having elements $y(n), \quad n = 1, 2, \ldots, N$

$$y(n) = \prod_{m=1}^{M} x(m, n) = x(1, n) \cdot x(2, n) \cdots x(M, n)$$

- The **cumulative column sum Y** is the matrix with M rows and N columns with elements denoted $y(k, n), \quad k = 1, 2, \ldots, M, \quad n = 1, 2, \ldots, N$

$$y(k, n) = \sum_{m=1}^{k} x(m, n) = x(1, n) + x(2, n) + \cdots + x(k, n)$$

- The **cumulative column product Y** is the matrix with M rows and N columns with elements denoted $y(k, n), \quad k = 1, 2, \ldots, M, \quad n = 1, 2, \ldots, N$

$$y(k, n) = \prod_{m=1}^{k} x(m, n) = x(1, n) \cdot x(2, n) \cdots x(k, n)$$

The MATLAB functions implementing these mathematical functions are the following.

`sum(x)` Returns the sum of the elements in vector `x`, or the row vector of the sum of elements of each column in matrix `x`

`prod(x)` Returns the product of the elements in vector `x`, or the row vector of the product of elements of each column in matrix `x`

`cumsum(x)` Returns a vector the same size as `x` containing the cumulative sum of the elements in vector `x`, or a matrix the same size as `x` containing cumulative sums of values from the columns of `x`.

`cumprod(x)` Returns a vector the same size as `x` containing the cumulative product of the elements in vector `x`, or a matrix the same size as `x` containing cumulative products of values from the columns of `x`.

Example 7.2 *Summation*

The following is a summation identity

$$\sum_{n=1}^{N} n = \frac{N(N+1)}{2}$$

This identity can be checked with MATLAB, for example, for $N = 8$:

```
>> N = 8;
>> n =1:8;
>> S = sum(n)
S =
    36
>> N*(N+1)/2
ans =
    36
```

■

Example 7.3 *Factorial*

The **factorial** of n is expressed and defined as

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 1$$

This can be computed as `prod(1:n)`, as in the following examples:

```
>> nfact = prod(1:4)
nfact =
    24
>> nfact = prod(1:70)
nfact =
  1.1979e+100
```

■

7.3 Statistical Analysis

Many engineering measurements have an element of randomness. Statistics is a branch of applied mathematics involving the analysis, interpretation, and presentation of data including some degree of randomness or uncertainty. In *descriptive statistics,* the important features or properties of a set of data are summarized or described.

Continuous random variable: A variable vector $\mathbf{x}$, whose elements can have any of a continuum of values for each measurement, or observation.

Population: all possible measurements of a random variable

Sample: finite number of measurements. Assume that N measurements have been made of a random variable, denoted as the vector $\mathbf{x}$, with elements $x(n),\ \ n = 1, \ldots, N$.

Frequency distribution of a random variable: indicates the relative frequency with which specific values of this random variable occur in the population.

Histogram: frequency distribution of sample data, indicating the frequency with which sample values fall within ranges of values, called **bins**.

- Range of values: $x_{min} \leq x \leq x_{max}$
- Number of bins: m
- Bin width: $\Delta x = \dfrac{x_{max} - x_{min}}{m}$
- Histogram value in bin k: $N(k)$, the number of samples with value $x_{min} + (k-1)\Delta x \leq x \leq x_{min} + k\Delta x$. This is also known as the **absolute frequency**.

The MATLAB commands for generating and plotting a histogram are;

Command	Description
`N = hist(x)`	Returns the row vector histogram `N` of the values in `x` using 10 bins. If `x` is matrix, returns a histogram matrix `N` operating on the columns of `x`.
`N = hist(x,m)`	Returns the row vector histogram `N` of the values in `x` using `m` bins.
`N = hist(x,xc)`	Returns the row vector histogram `N` of the values in `x` using bin centers specified by `xc`.
`N = histc(x,edges)`	Counts the number of values in vector `x` that fall between the elements in the `edges` vector (which must contain monotonically non-decreasing values). `N` is a `length(edges)` vector containing these counts. `N(k)` will count the value `x(i)` if `edges(k) >= x(i) > edges(k+1)`. The last bin will count any values of `x` that match `edges(end)`. Values outside the values in `edges` are not counted. Use `-inf` and `inf` in `edges` to include all non-`NaN` values.
`[N,xc] = hist(...)`	Also returns the position of the bin centers in `xc`.
`hist (...)`	Without output arguments produces a histogram bar plot of the results.

For example, the following commands produce the 25-bin histograms shown in Figure 7.4, operating on the random data vectors `data1` and `data2`.

```
% Histogram plots
%
subplot(2,1,1),hist(data1,25),...
  title('Histogram of data1'),...
  xlabel('x'),...
  ylabel('N'),...
subplot(2,1,2),hist(data2,25),...
  title('Histogram of data2'),...
  xlabel('x'),...
  ylabel('N')
```

Note the differences in the nature of the two distributions. Random data `data1` is called a **uniform distribution**, as it has roughly equal, or uniform, frequency in all bins. Random data `data2` has a *bell-shaped* distribution, called a **Gaussian distribution** or **Normal distribution**.

Relative frequency histogram: The histogram value in bin k is normalized by the total number, N, of data samples: $N(k)/N$.

The `hist` function is limited in its ability to produce relative frequency histograms. It is better to generate such plots with the `bar` function, defined as follows:

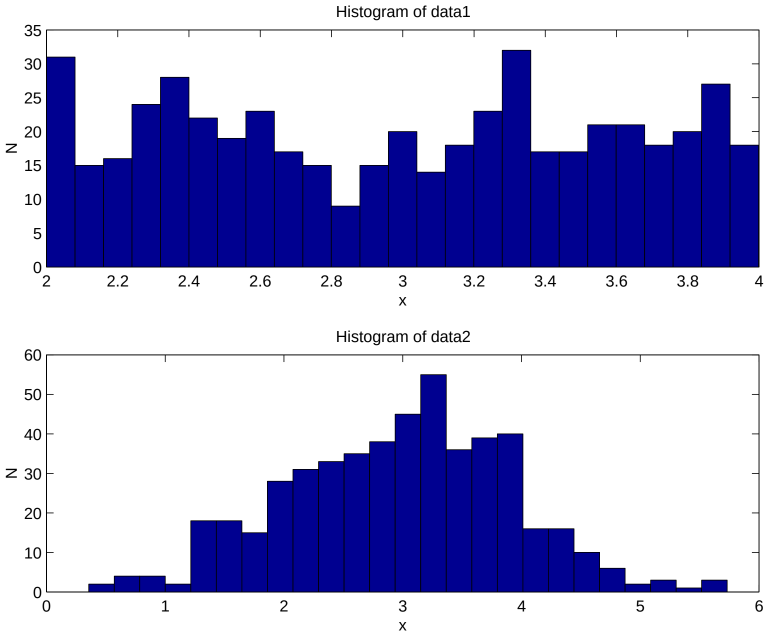


Figure 7.4: Absolute frequency histograms

bar(X,Y) Draws the columns of the M-by-N matrix Y as M groups of N vertical bars. The vector X must be monotonically increasing or decreasing.

bar(Y) Uses the default value of X=1:M.

bar(x,y For vector inputs, length(y) bars are drawn, with each bar centered on a value of x.

bar(X,Y,width) Specifies the width of the bars. Values of width ¿ 1 produce overlapped bars. The default value is width=0.8

For example, the following commands produce relative histogram plots shown in Figure 7.5 for the random data vectors data1 and data2.

```
% Relative frequency histogram plots
%
n1 = length(data1);
[freq1,x1] = hist(data1,25);
rfreq1 = freq1/n1;
n2 = length(data2);
[freq2,x2] = hist(data2,25);
rfreq2 = freq2/n2;

subplot(2,1,1),bar(x1,rfreq1),...
  title('Relative histogram of data1'), grid,...
  xlabel('x'), ylabel('relative frequency'),...
subplot(2,1,2),bar(x2,rfreq2),...
  title('Relative histogram of data2'), grid,...
  xlabel('x'), ylabel('relative frequency')
```

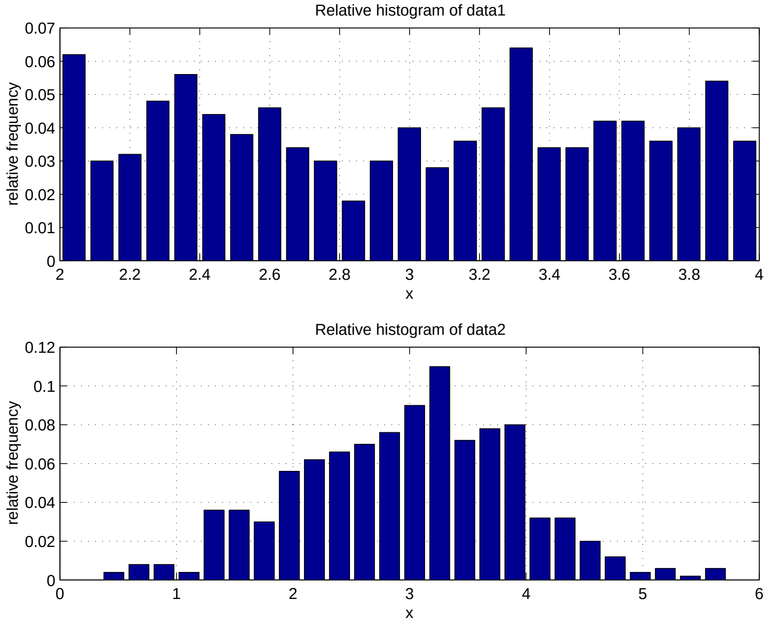


Figure 7.5: Relative frequency histograms

Measures of central tendency

The **mean** and **median** describe the middle, or center, of the range of the random variable.

The **sample mean** of vector **x** having N elements is $\bar{x}$

$$\bar{x} = \frac{1}{N}\sum_{n=1}^{N} x(m)$$

The **population mean** μ (mu) is the value of $\bar{x}$ for an infinite number of measurements, denoted by

$$\mu = \lim_{N\to\infty} \bar{x}$$

where the right hand side is read as "the limit of x bar as N approaches infinity."

Median: Middle value of a set of random samples, sorted by value (rank ordered). If there is an even number of samples, then the median is the average of the two middle sample values.

Command	Description
`mean(x)`	Returns the sample mean of the elements of the vector `x`. Returns a row vector of the sample means of the columns of matrix `x`.
`median(x)`	Returns the median of the elements of the vector `x`. Returns a row vector of the medians of the columns of matrix `x`.
`sort(x)`	Returns a vector with the values of vector `x` in ascending order. Returns a matrix with each column of matrix `x` in ascending order.

For example:

```
>> mean(data1)
ans =
    2.9940
>> sum(data1)/length(data1)
ans =
    2.9940
>> mean(data2)
ans =
    2.9768
>> median(data1)
ans =
    3.0285
>> median(data2)
ans =
    3.0574
```

These results can be confirmed by observing the plots of the data in Figure 7.1, in which the data values can be seen to be centered on 3.0. For the two data sets considered, the values of the mean and median are very close. This is not necessarily the case for other data sets. Also note that the results above show that `mean(data1) = sum(data1)/length(data1)`.

Measures of variation

Measures of variation: indicate the degree of deviation of random samples from the measure of central tendency.

Referring again to the plots of our random data sets `data1` and `data2` in Figure 7.1, observe that `data2` has greater variation from the mean.

The **sample standard deviation** of vector $\mathbf{x}$ having N elements is

$$s = \left[\frac{1}{N-1} \sum_{n=1}^{N} (x(n) - \bar{x})^2 \right]^{\frac{1}{2}}$$

The **sample variance** s^2 is the square of the standard deviation.

`std(x)` Returns the sample standard deviation of the elements of the vector `x`. Returns a row vector of the sample standard deviations of the columns of matrix `x`.

For example:

```
>> std(data1)
```

```
ans =
    0.5989
>> std(data2)
ans =
    0.9408
```

Thus, the variation of `data2` is greater than that of `data1`, as we concluded from observation of the plotted data values.

7.4 Random Number Generation

Many engineering problems require the use of **random numbers** in the development of a solution. In some cases, the random numbers are used to develop a simulation of a complex problem. The simulation can be tested over and over to analyze the results, with each test representing a repetition of the experiment. Random numbers also used to represent noise sequences, such as those heard on a radio.

Uniform Random Numbers

Random numbers are characterized by their frequency distributions. **Uniform random numbers** have a constant or uniform distribution over their range between minimum and maximum values.

The `rand` function in MATLAB generates uniform random numbers distributed over the interval [0,1]. A *state vector* is used to generate a random sequence of values. A given state vector always generates the same random sequence. Thus, the sequences are known more formally as **pseudorandom sequences**. This generator can generate all the floating point numbers in the closed interval [2^{-53}, $1 - 2^{-53}$]. Theoretically, it can generate over 2^{1492} values before repeating itself.

Command	Description
`rand(n)`	Returns an `n`×`n` matrix `x` of random numbers distributed uniformly in the interval [0,1].
`rand(m,n)`	Returns an `m`×`n` matrix `x` of random numbers distributed uniformly in the interval [0,1].
`rand`	Returns a scalar whose value changes each time it is referenced. `rand(size(A))` is the same size as A.
`s = rand('state')`	Returns a 35-element vector `s` containing the current state of the uniform generator.
`rand('state',s)`	Resets the state to `s`.
`rand('state',0)`	Resets the generator to its initial state.
`rand('state',J)`	Resets the generator to its `J`-th state for integer `J`.
`rand('state',sum(100*clock))`	Resets the generator to a different state each time.

The following commands generate and display two sets of ten random numbers uniformly distributed between 0 and 1; the difference between the two sets is caused by the different states.

```
rand('state',0)
```

```
set1 = rand(10,1);
rand('state',123)
set2 = rand(10,1);
[set1 set2]
```

The results displayed by these commands are the following, where the first column gives values of set1 and the second column gives values of set2:

```
    0.9501    0.0697
    0.2311    0.2332
    0.6068    0.7374
    0.4860    0.7585
    0.8913    0.6368
    0.7621    0.6129
    0.4565    0.3081
    0.0185    0.2856
    0.8214    0.0781
    0.4447    0.9532
```

Note that as desired, the two sequences are different.

To generate sequences having values in the interval $[x_{min}, x_{max}]$, first generate a random number r in the interval $[0, 1]$, multiply by $(x_{max} - x_{min})$, producing a number in the interval $[0, (x_{max} - x_{min}]$, then add x_{min}. To generate a row vector of 500 elements, the command needed is

```
x = (xmax - xmin)*rand(1,500) + xmin
```

The sequence data1, plotted in Figure 7.1, was generated with the command:

```
data1 = 2*rand(1,500) + 2;
```

Thus, $x_{max} - x_{min} = 2$ and $x_{min} = 2$, so $x_{max} = 4$ and the range of the data sequence is (2,4), which can be confirmed from Figure 7.1.

Example 7.4 *Flipping a coin*

When a fair coin is flipped, the probability of getting heads or tails is 0.5 (50%). If we want to represent tails by 0 and heads by 1, we can first generate a uniform random number in the range [0,2). The probability of a result in the range [0,1) is 0.5 (50%), which can be mapped to 0 by truncation of this result. Similarly, the probability of a result in the range [1,2) is 0.5 (50%), which can be mapped to 1 by truncation. The trunction function in Matlab is floor.

A script to simulate an experiment to flip a coin 50 times and display the results is the following:

```
% Coin flipping simulation
%
% Coin flip:
n = 50;                     % number of flips
coin = floor(2*rand(1,n))   % vector of n flips: 0: tails, 1:heads

% Histogram computation and display:
xc = [0 1];                 % histogram centers
y = hist(coin,xc);          % absolute frequency
bar(xc,y), ylabel('frequency'),...
   xlabel('0: tails, 1: heads'),...
   title('Absolute Frequency Histogram for 50 Coin Flips')
```

The output displayed by the script:

```
coin =
  Columns 1 through 12
     1     1     0     0     1     0     1     1     1     0     1     1
  Columns 13 through 24
     1     0     0     0     0     1     1     0     1     0     1     1
  Columns 25 through 36
     0     1     1     1     0     0     0     0     0     0     1     0
  Columns 37 through 48
     0     1     0     1     0     1     1     0     1     1     0     0
  Columns 49 through 50
     0     0
```

The histogram plot is shown in Figure 7.6. Note that in this trial, there were 26 tails and 24 heads, close to the expected result. Since this is a random process, repeated trials will give different results.

■

Example 7.5 *Rolling dice*

When a fair die (singular of dice) is rolled, the number uppermost is equally likely to be any integer from 1 to 6. The following statement will generate a vector of 10 random numbers in the range 1-6:

```
die = floor( 6 * rand(1,10) + 1 )
```

The following are the results of two such simulations:

```
     2     3     4     5     2     1     3     3     1     4
     5     2     2     5     5     6     3     6     3     5
```

■

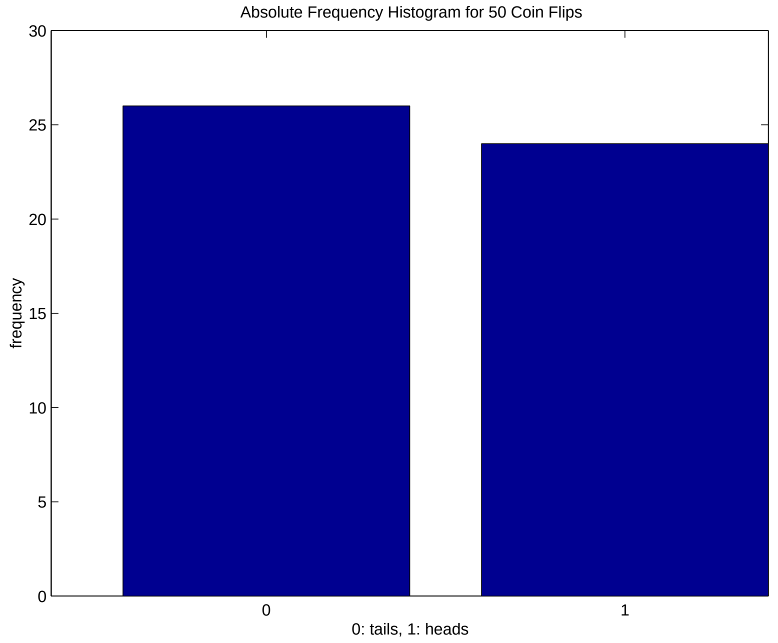


Figure 7.6: Histogram of 50 Coin Flips

Gaussian Random Numbers

A random distribution that is an appropriate model for data sets from many engineering problems is the **Gaussian** or **Normal** distribution. This is the distribution having a bell shape, with a high relative frequency at the mean, decreasing away from the mean, but extending indefinitely in both directions. The Gaussian distribution is specified by two parameters: (1) the population mean μ, and (2) the population standard deviation σ, with the distribution function given by

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/2\sigma^2}$$

This distribution is plotted in Figure 7.7, where it is observed to be symmetrical about the mean, dropping off rapidly away from the mean. The distribution can be normalized by performing a change of scale that converts the units of measurement into standard units

$$z = \frac{x-\mu}{\sigma}$$

as shown in Figure 7.7. It can be shown that approximately 68% of the values will fall within one standard deviation of the mean ($-1 < z < 1$), 95% will fall within two standard deviations of the mean ($-2 < z < 2$), and 99% will fall within three standard deviations of the mean ($-3 < z < 3$).

The Matlab functions for generating Gaussian random numbers are:

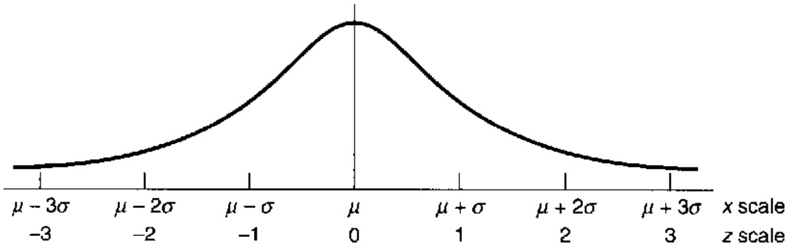


Figure 7.7: Normalized Gaussian distribution

Command	Description
`randn(n)`	Returns an `n`×`n` matrix `x` of Gaussian random numbers with a mean of 0 and a standard deviation of 1.
`randn(m,n)`	Returns an `m`×`n` matrix `x` of Gaussian random numbers with a mean of 0 and a standard deviation of 1.
`randn`	Returns a scalar whose value changes each time it is referenced. `rand(size(A))` is the same size as `A`.
`s = randn('state')`	Returns a 2-element vector `s` containing the current state of the uniform generator.
`randn('state',s)`	Resets the state to `s`.
`randn('state',0)`	Resets the generator to its initial state.
`randn('state',J)`	Resets the generator to its `J`-th state for integer `J`.
`randn('state',sum(100*clock))`	Resets the generator to a different state each time.

To generate a row vector `x` of 500 Gaussian random variables with mean m and standard deviation s:

```
x = m + s*randn(1,500);
```

The sequence `data2`, plotted in Figure 7.1, was generated with the command:

```
data2 = rand(1,500) + 3;
```

Thus, $m = 3$ and $s = 1$. The mean value can be confirmed from the plot in Figure 7.1 and the mean value computed in Section 7.3. The standard deviation can be confirmed by the value computed in Section 7.3. The computed values are not identical to the desired values because the computed values are due to a finite number of random samples. The computed and desired values become closer as the number of samples increases.

Example 7.6 *Noisy signal simulation*

Many sensors, which convert physical quantities into electrical signals, produce measurement errors, resulting in what is known as noisy signals. Additive Gaussian random numbers are often an accurate model of the errors. The signal model is

$$x = s + n$$

where s is the noise-free signal of interest, n is the Gaussian measurement error, and x is the noisy signal. A measure of the signal quality is the *signal to noise ratio* (SNR), defined by

$$\begin{aligned}\text{SNR} &= 10\log_{10}\frac{\sigma_s^2}{\sigma_n^2} \\ &= 20\log_{10}\frac{\sigma_s}{\sigma_n}\end{aligned}$$

where σ_s is the standard deviation of the signal and σ_n is the standard deviation of the noise.

Consider simulating a noisy signal consisting of a sine wave with additive Gaussian measurement noise. The variance of a sine wave can be shown to be

$$\sigma_s^2 = \frac{A}{2}$$

where A is the sinusoidal amplitude. For $A = 1$ and Gaussian noise standard deviation $\sigma_n = 0.1$, the SNR is

$$\text{SNR} = 10\log_{10}\frac{0.5}{0.1^2} = 10\log_{10} 50 = 17.0$$

The following script simulates this noisy signal. In the next section, methods to reduce the affect of the noise will be considered.

```
% Noisy signal generation and analysis
t = linspace(0,10,512);              % time base
s = sin(2*pi/5*t);                   % signal
n = 0.1*randn(size(t));              % noise
x = s + n;                           % noisy signal
disp('Signal to Noise Ratio (SNR), dB')
SNR = 20*log10(std(s)/std(n))        % signal to noise ratio, dB
plot(t,x), xlabel('Time (s)'),...
  ylabel('Signal amplitude'),...
  title('Noisy signal')
```

The computed SNR is

```
Signal to Noise Ratio (SNR), dB
SNR =
   16.9456
```

This is very close to the value computed above. The reason for the difference is the finite number of samples used to estimate the noise standard deviation. The resulting noise signal is shown in Figure 7.8.

■

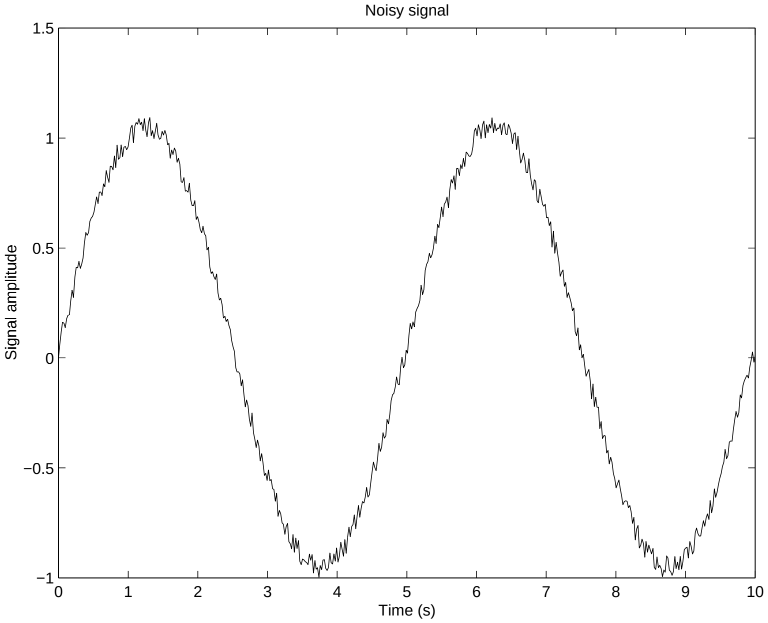


Figure 7.8: Simulated noisy sinusoidal signal

Section 8

Selection Programming

A **selection** statement allows a question to be asked or a condition to be tested to determine which steps are to be performed next. The question or condition is defined using **relational** and **logical** operators, which will be described prior to introducing the selection statements.

Outline

- Relational and logical operators
- Control flow
- Loops
- Selection statements in user-defined functions
- Update processes
- Applied problem solving: speech signal analysis

8.1 Relational and Logical Operators

For relational and logical expressions:

Inputs:	True is any nonzero number False is 0 (zero)
Outputs:	True is 1 (one) False is 0 (zero)

An output array variable assigned to a relational or logical expression is identified as **logical**. That is, the result contains numerical values 1 and 0, that can be used in mathematical statements, but also allow logical array addressing.

Relational Operators

For more information: `help ops`.

Relational Operator	Description
`<`	less than
`<=`	less than or equal
`>`	greater than
`>=`	greater than or equal
`==`	equal
∼=	not equal

Relational operators can be used to compare two arrays of the same size or to compare an array to a scalar. In the second case, the scalar is compared with all elements of the array and the result has the same size as the array.

Examples:

```
>> A=1:9, B=8-A
A =
     1     2     3     4     5     6     7     8     9
B =
     7     6     5     4     3     2     1     0    -1
>> tf1 = A <=4
tf1 =
     1     1     1     1     0     0     0     0     0
>> tf2 = A > B
tf2 =
     0     0     0     0     1     1     1     1     1
>> tf3 = (A==B)
tf3 =
     0     0     0     1     0     0     0     0     0
>> tf4 = B-(A>2)
tf4 =
     7     6     4     3     2     1     0    -1    -2
```

- `tf1` finds elements of `A` that are less than or equal to 4. Ones appear in the result where `A`≤ 4 and zeroes appear where `A`> 4.
- `tf2` finds elements of `A` that are greater than those in `B`.
- `tf3` finds elements of `A` that are equal to those in `B`. Note that `=` and `==` mean two different things: `==` compares two variables and returns ones where they are equal and zeros where they are not; `=`, on the other hand, is used to assign the output of an operation to a variable.
- `tf4` finds where `A>2` and subtracts the resulting vector from `B`. This shows that since the output of logical operations are numerical arrays of ones and zeros, they can be used in mathematical operations.

Note that = and == mean two different things: == compares two variables and returns ones where they are equal and zeros where they are not; on the other hand, = is used to assign the output of an operation to a variable.

Example 8.1 *Avoiding division by zero*

Consider the following:

```
>> x=(-3:3)/3
x =
   -1.0000   -0.6667   -0.3333         0    0.3333    0.6667    1.0000
>> sin(x)./x

Warning: Divide by zero.
ans =
    0.8415    0.9276    0.9816       NaN    0.9816    0.9276    0.8415
```

Computing the function $\sin(x)/x$ gives a warning because the fourth element in x is zero. Since $\sin(0)/0$ is undefined, the result for that element in the result is NaN (meaning Not a Number). The following will replace the zero in x with the special number eps, which is approximately 2.2×10^{-16}, resolving the divide-by-zero problem.

```
>> x = x + (x==0)*eps
x =
   -1.0000   -0.6667   -0.3333    0.0000    0.3333    0.6667    1.0000
>> sin(x)./x
ans =
    0.8415    0.9276    0.9816    1.0000    0.9816    0.9276    0.8415
```

■

Logical Operators

Logical operators provide a way to combine or negate relational expressions.

Logical Operator	Description
&	and
|	or
~	not

A fourth logical operator is implemented as a function:

xor(A,B) Exclusive or: Returns ones where either A or B is True (nonzero); returns False (zero) where both A and B are False (zero) or both are True (nonzero).

Definitions of the logical operators, with 0 representing False and 1 representing True:

`A`	`B`	`∼A`	`A|B`	`A&B`	`xor(A,B)`
0	0	1	0	0	0
0	1	1	1	0	1
1	0	0	1	0	1
1	1	0	1	1	0

The precedence from highest to lowest is relational operators, followed by logical operators ∼, `&`, and |. Parentheses can be used to change the precedence and should be used liberally to clarify the operations.

Examples:

```
>> A=1:9
A =
     1     2     3     4     5     6     7     8     9
>> tf1 = A>4
tf1 =
     0     0     0     0     1     1     1     1     1
>> tf2 = ~(A>4)
tf2 =
     1     1     1     1     0     0     0     0     0
>> tf3 = (A>2)&(A<6)
tf3 =
     0     0     1     1     1     0     0     0     0
>> tf4 = xor((A>2),(A<6))
tf4 =
     1     1     0     0     0     1     1     1     1
```

- `tf1` finds where `A` is greater than 4.
- `tf2` negates `tf1`, finding where `A` is not greater than 4.
- `tf3` finds where `A` is greater than 2 *and* less than 6.
- `tf4` find where the exclusive or of `(A>2)` and `(A<6)`, which is observed to be the same as the logical inverse of `tf3`.

Example 8.2 *Generating discontinuous signals*

The logical operators allow the generation of arrays representing signals with discontinuities or signals that are composed os segments of other signals. The basic idea is to multiply those values in a array that are to be retained with ones, and multiply all other values with zeros.

Consider the discontinuous signal:

$$x(t) = \begin{cases} \sin(t) & \sin(t) > 0 \\ 0 & \sin(t) < 0 \end{cases}$$

The following script computes and plots this signal over the range t = [0,10] s.

```
t = linspace(0,10,100);     % create x vector
x = sin(t);                 % compute sine vector
x = x.*(x>0);               % set negative values of sin(t) to zero
%
% Plot x and label
plot(t,x),xlabel('Time (s)'),ylabel('Amplitude'),...
     title('Discontinuous signal'), axis([0 10 -0.1 1.1])
```

The plot that is generated is shown in Figure 8.1.

■

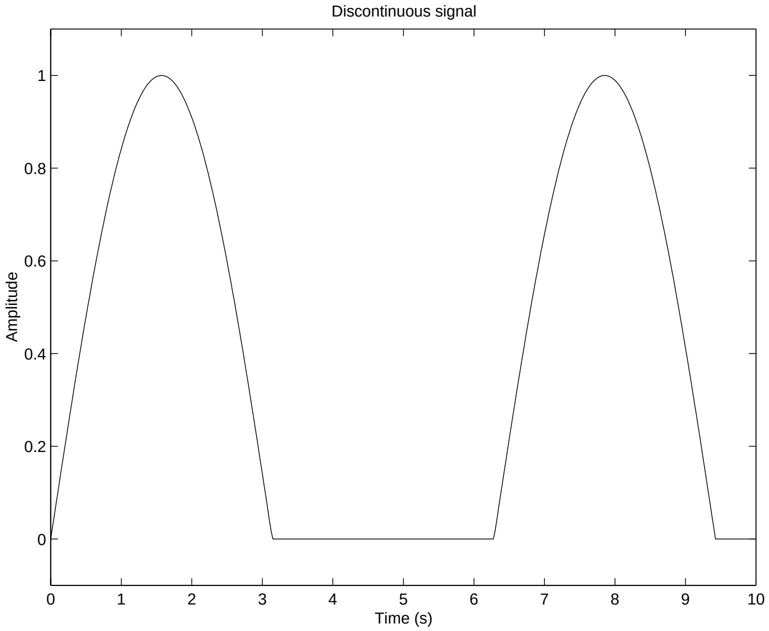


Figure 8.1: Plot of a discontinuous signal

Relational and Logical Functions

Matlab provides several useful *relational and local functions* that operate on scalars, vectors, and matrices. The following is a partial list of these functions.

Function	Description
`any(x)`	Returns a scalar that is 1 (true) if *any* element in the vector `x` is nonzero; otherwise, the scalar is 0 (false). Returns a row vector containing a 1 (true) in each element for which any element of the corresponding column of matrix `x` is nonzero, and a 0 (false) otherwise.
`all(x)`	Returns a scalar that is 1 (true) if *all* elements in the vector `x` are nonzero; otherwise, the scalar is 0 (false). Returns a row vector containing a 1 (true) in each element for which all elements of the corresponding column of matrix `x` are nonzero, and a 0 (false) otherwise.
`find(x)`	Returns a vector containing the indices of the nonzero elements of a vector `x`. Returns a vector containing the indices of the nonzero elements of `x(:)`, which is a single-column vector formed from the columns of matrix `x`.
`isnan(x)`	Returns an array with ones where the elements of `x` are `NaN` and zeros where they are not.
`isfinite(x)`	Returns an array with ones where the elements of `x` are finite and zeros where they are not. For example, `isfinite([pi NaN Inf -Inf])` is `[1 0 0 0]`.
`isinf(x)`	Returns an array with ones where the elements of `x` are `+Inf` or `-Inf` and zeros where they are not.
`isempty(x)`	Returns 1 if `x` is an empty array and 0 otherwise.

Example 8.3 *Height and speed of a projectile*

The height and speed of a projectile (such as a thrown ball) launched with a speed of v_0 at an angle θ to the horizontal are given by

$$h(t) = v_0 t \sin\theta - 0.5gt^2$$

$$v(t) = \sqrt{v_0^2 - 2v_0 gt \sin\theta + g^2t^2}$$

where g is the acceleration due to gravity. The projectile will strike the ground when $h(t) = 0$, which gives the time to return to the ground, $t_g = 2(v_0/g)\sin\theta$. For $\theta = 40°$, $v_0 = 20$ m/s, and $g = 9.81$ m/s^2, determine the times when the height is no less than 6m and the speed is simultaneously no greater than 16 m/s.

This problem can be solved by using the `find` command to determine the times at which the logical expression `(h >= 6 & (v <= 16)` is ture.

```
% Set the values for initial speed, gravity, and angle
v0 = 20; g = 9.81; theta = 40*pi/180;
% Compute the time to return to the ground
t_g = 2*v0*sin(theta)/g;
```

```
% Compute the arrays containing time, height, and speed.
t = [0:t_g/200:t_g];
h = v0*t*sin(theta) - 0.5*g*t.^2;
v = sqrt(v0^2 - 2*v0*g*sin(theta)*t + g^2*t.^2);
% Determine when the height is no less than 6,
% and the speed is no greater than 16.
u = find(h>6 & v <= 16);
% Compute the corresponding times
t_1 = t(u(1))
t_2 = t(u(end))
```

The beginning of the time interval, t_1 is the value of `t` indexed by the first element of `u` and the end of the time interval, t_2 is the value of `t` indexed by the last element of `u`.

The results are:

```
t_1 =
    0.8518
t_2 =
    1.7691
```

This problem could have been solved by plotting $h(t)$ and $v(t)$, but the accuracy of the results would be limited by our ability to pick points off the graph. In addition, the graphical approach is more time-consuming.

■

8.2 Flow Control

Selection statements that test the results of relational or logical functions or operators are the decision-making structures that allow the flow of command execution to be controlled.

For more information: `help lang`.

Simple `if` Statement

The general form of a simple `if` statement is:

```
if    logical expression
      commands
end
```

If the logical expression is true, the commands between the `if` statement and the `end` statement are executed. If the logical expression is false, the flow of execution jumps immediately to the `end` statement without executing the statements between the `if` statement and the `end` statement.

To help in reading and understanding an `if` structure, it is good style to indent the statements within the structure.

Example:

```
if d < 50
   count = count + 1;
   disp(d);
end
```

Assuming that `d` is a scalar, then if `d` is less than 50, `count` is incremented by 1 and the value of `d` is displayed; otherwise, these two statements are skipped. If `d` is not a scalar, then `count` is incremented by 1 and `d` is displayed only if every element of `d` is less than 50.

Nested `if` Statements

`if` statements may be nested, as shown in the following example:

```
if d < 50
   count = count + 1;
   disp(d);
   if b > d
      b = 0;
   end
end
```

Assuming first that `b` and `d` are scalars, then if `d<50`, `count` is incremented by 1 and `d` is displayed. In addition, if `b>d`, then `b` is set to 0. If `d` is not less than 50, we skip immediately to the second `end` statement.

`else` and `elseif` Clauses

`else` clause: allows one set of statements to be executed if a logical expression is true and a different set if the logical expression is false.

For example, if variable `interval` is less than one, set the value of `xinc` to `interval/10`; otherwise, set the value of `xinc` to 0.1.

```
if interval < 1
   xinc = interval/10;
else
   xinc = 0.1;
end
```

When several levels of `if-else` statements are nested, it may be difficult to determine which logical expressions must be true (or false) to execute each set of statement. In these cases, the `elseif` clause is often used to clarify the program logic, as shown in the example below.

```
if temperature > 100
   disp('Too hot - equipment malfunctioning.')
elseif temperature > 90
   disp('Normal operating range.')
elseif temperature > 50
   disp('Below desired operating range.')
else
   disp('Too cold - turn off equipment.')
end
```

In this example, temperature between 90 and 100 are in the normal operating range; temperatures outside this range generate an appropriate message.

Example 8.4 *Testing variable type*

The `if` command can be used to write a function M-file to take a single input argument and then report, in text, whether that argument is scalar, vector, or matrix. No argument is returned.

```
function testvar(x)
% Display text indicating whether x is a
% scalar, vector, or matrix
[m,n] = size(x);
if m==n & m==1
  disp('     Argument is a scalar')
elseif m==1 | n==1
  disp('     Argument is a vector')
else
  disp('     Argument is a matrix')
end
```

A test of this function:

```
>> a=2; b=[2 3]; c=[4 5; 6 7];
>> testvar(a)
      Argument is a scalar
>> testvar(b)
      Argument is a vector
>> testvar(c)
      Argument is a matrix
```

■

Switch Selection Structure

The `switch` selection structure provides an alternative to using the `if`, `elseif`, and `else` commands. Anything programmed using `if` structures can also be programmed using `switch` structures. The advantage of the `switch` structure is that in some situations, it yields code that is more readable.

The syntax is

```
switch expression
    case test expression 1
       commands
    case {test expression 2, test expression 3}
       commands
    .
    .
    .
    otherwise
       commands
end
```

The *expression* result is compared in turn to the result of each `case` *test expression*. If they are equal, then the commands following the `case` command are executed and processing continues with the command following the `end` statement. If *expression* is a character string, then a string comparison is made with the `case` *test expression*. Multiple *test expressions* can be listed, comma separated, enclosed in braces {}. Only the first matching `case` is executed, If no match occurs, the statements following the `otherwise` statement are executed. However, the `otherwise` statement is optional. If it is absent, execution continues with the command following the `end` statement if no match exists. Each `case` *test expression* statement must be on a single line.

Consider an example that was previously implemented with an `if`, `else` structure:

```
switch interval < 1
  case 1
     xinc = interval/10;
  case 0
     xinc = 0.1;
end
```

For this example, the resulting code is no more readable than in the previous implementation.

An example using string comparisons:

```
x = 6.1;
units = 'ft';
% convert x to meters
switch units
  case {'inch','in'}
```

```
     y = x*0.0254;
  case{'feet','ft'}
     y = x*0.3048;
  case{'meter','m'}
     y = x;
  case{'centimeter','cm'}
     y = x/100;
  case{'millimeter','mm'}
     y = x/1000;
  otherwise
     disp(['Unknown units: ' units])
     y = NaN;
end
```

Executing this example gives a final value of `y = 1.8593`.

8.3 Loops

A **loop** is a structure that allows a group of commands to be repeated.

`for` Loop

A `for` loop repeats a group of commands a fixed, predetermined number of times. A `for` loop has the following structure:

```
for  variable=expression
     commands
end
```

The commands between the `for` and `end` statements are executed once for every column in the expression, beginning with the first column and stepping through to the last column. At each step, known as an iteration, the appropriate column of the expression is assigned to the variable. Thus, on step n, column n of the expression is assigned to the variable, which then can be operated on by one of the commands in the loop.

Rules for writing and using a `for` loop include:

1. If the expression results in an empty matrix, the loop will not be executed. Control will pass directly to the statement following the `end` statement.

2. If the result of the expression is a scalar, the loop will be executed once, with the variable equal to the value of the scalar.

3. If the result of the expression is a vector, then each time through the loop, the variable will contain the next value in the vector.

4. A `for` loop cannot be terminated by reassigning the loop variable within the loop.
5. Upon completion of a `for` loop, the variable contains the last value used.
6. The colon operator can be used to define the expression using the following format
   `for` index = initial:increment:limit

Example 8.5 *Use of a* `for` *statement in a function*

Consider writing a user-defined function that searches a matrix input argument for the element with the largest value and returns the indices of that element.

```
function [r,c] = indmax(x)
% INDMAX returns the row and column indices of
% the maximum-valued element in x
[m n] = size(x);
xmax = x(1,1);
r=1; c=1;
for k=1:m
  for l=1:n
     if x(k,l)> xmax
        xmax = x(k,l);
        r=k;
        c=l;
     end
  end
end
```

Testing this function:

```
>> A=3; B=[2 4 3]; C=[3 2; 6 1];
>> [r c]=indmax(A)
r =
     1
c =
     1
>> [r c]=indmax(B)
r =
     1
c =
     2
>> [r c]=indmax(C)
r =
     2
c =
     1
```

■

Avoiding Loops

In general, loops should be avoided in MATLAB, as they can significantly increase the execution time of a program. MATLAB has the capability to increase the size of vectors and matrices dynamically, as can be required by a for loop. For example, a vector can be of length N at the Nth iteration of a loop, and at iteration $N+1$, MATLAB can increase the length of the vector from N to $N+1$. However, this dynamic storage allocation is not accomplished efficiently in terms of computation time.

For example, consider the following script for generating a sine wave:

```
% sinusoid of length 5000
for t=1:5000
    y(t) = sin(2*pi*t/10);
end
```

This results in a scalar t, with final value 5000, and a vector y, a sine wave with 10 samples per cycle, with 5000 elements. Each time the command inside the for loop is executed, the size of the variable y must be increased by one. The execution time on a certain PC is 7.91 seconds.

Then consider a second script:

```
% sinusoid of length 10000
for t=1:10000
    y(t) = sin(2*pi*t/10);
end
```

Clearing the MATLAB workspace and running this script to produce vector y having 10000 elements requires an execution time on the same PC of 28.56 seconds, nearly four times the execution time of the first script. Forcing MATLAB to allocate memory for y each time through the loop takes time, causing the execution time to grow geometrically with the vector length.

To maximize speed, arrays should be preallocated before a for loop is executed. To eliminate the need to increase the length of y each time through the loop, the script could be rewritten as:

```
% sinusoid of length 10000 with vector preallocation
y = zeros(1,10000);
for t=1:10000
    y = sin(2*pi*t/10);
end
```

In this case, y was arbitrarily defined to be a vector of length 10000, with values recomputed in the loop. The computation time in this case was 2.03 seconds.

Execution time can be decreased even further by "vectorizing" the algorithm, applying an operation to an entire vector instead of a single element at a time. Applying this approach to our example of sinusoid generation:

```
% better method of sinusoid generation
t = 1:10000;
y = sin(2*pi*t/10);
```

This script produces the vector t with 10,000 elements and then the vector y with 10,000 elements, which is the as same y from the second script above. However, again using the same PC, this script executes in 0.06 second, as opposed to 16.04 seconds for the second for loop script, or 1.92 seconds with a for loop and vector preallocation.

while Loop

A while loop repeats a group of commands as long as a specified condition is true. A while loop has the following structure:

```
while  expression
       commands
end
```

If **all** elements in expression are true, the commands between the while and end statements are executed. The expression is reevaluated and if all elements are still true, the commands are executed again. If any element in the expression is false, control skips to the statement following the end statement. The variables modified within the loop should include the variables in the expression, or the value of the expression will never change. If the expression is always true (or is a value that is nonzero), the loop becomes an **infinite loop**.

Example 8.6 *Computing the special value* eps

Consider the following example of one way to compute the special MATLAB value eps, which is the smallest number that can be added to 1 such that the result is greater than 1 using finite precision:

```
num = 0;
EPS = 1;
while (1+EPS)>1
    EPS=EPS/2;
    num = num + 1;
end
num = num - 1
EPS = 2*EPS
```

Uppercase EPS was used so that the MATLAB value eps was not overwritten. EPS starts at 1 and is continually divided in two, as long as (1+EPS)>1 is True (nonzero). EPS eventually gets so small that adding EPS to 1 is no longer greater than 1. The loop then terminates, and EPS is multiplied by 2 because the last division by 2 made it too small by a factor of two. The results:

```
num =
    52
EPS =
  2.2204e-016
```

∎

While Loops and Break

The `while` loop allows the execution of a set of commands an indefinite number of times. The `break` command is used to terminate the execution of the loop.

Example 8.7 *Use of a* `while` *loop to evaluate quadratic polynomials*

Consider writing a script to ask the user to input the scalar values for a, b, c, and x and then returns the value of $ax^2 + bx + c$. The program repeats this process until the user enters zero values for all four variables.

```
% Script to compute ax^2 +bx + c
disp(’Quadratic ax^2+bx+c evaluated’)
disp(’for user input a, b, c, and x’)
a=1; b=1; c=1; x=0;
while a~=0 | b~=0 | c~=0 | x~=0
  disp(’Enter a=b=c=x=0 to terminate’)
  a = input(’Enter value of a: ’);
  b = input(’Enter value of b: ’);
  c = input(’Enter value of c: ’);
  x = input(’Enter value of x: ’);
  if a==0 & b==0 & c==0 & x==0
     break
  end
  quadratic = a*x^2 + b*x + c;
  disp(’Quadratic result:’)
  disp(quadratic)
end
```

Note that this script will display multiple quadratic expression values if arrays are entered for `a`, `b`, `c`, or `x`. Consider modifying the script to check each input and breaking if any input is not a scalar.

∎

8.4 Selection Statements in User-Defined Functions

1. User-defined functions must appropriately handle any input data and produce output variables of the proper dimensions and values. This often requires the use of selection statements

to determine the dimensions and values of the input.

2. The function `error` displays a character string in the *Command* window, aborts function execution, and returns control to the keyboard. This function is useful for flagging improper command usage, as in the following portion of a function:

   ```
   if length(val)>1
      error('val must be a scalar.')
   end
   ```

3. The functions `nargin` and `nargout` can be used in functions to determine the number of input arguments and the number of output arguments used to call the function. This provides information for handling cases when the function is to use a variable number of arguments. For example, if a default value of 1 is to be returned for the output variable `err` if the function call includes no input arguments, the function can include the following:

   ```
   if nargin==0
      err=1;
   end
   ```

   If the function is written to use a fixed number of input or output arguments, but the wrong number has been supplied in the use of the funciton, MATLAB provides the needed error handling, without the need for the `nargin` and `nargout` functions.

4. Function M-files terminate execution and return when they reach the end of the M-file, or alternatively, when the command `return` is encountered.

Example 8.8 *Step signal*

The step signal is zero for negative time, "stepping up" to one for positive time

$$u(t) = \begin{cases} 0 & t < 0 \\ 1 & t \geq 0 \end{cases}$$

A user-defined function to implement the step signal:

```
function u = step(t)
% STEP  unit step function
% u = step(t)
% u = 0 for t<0
% u = 1 for t>=0
u = zeros(size(t));
u(find(t>=0))=1;
```

The output array `u` is first set to an array of zeros with the same dimensions at that of the input argument `t`. Then the elements of `u` corresponding to values of `t` greater than or equal to 0 are set to 1, with the use of the `find` function.

8.5 Update Processes

Many problems in science and engineering involve modelling a process where the main variable is updated over a period of time. In many situations, the updated value is a function of the current value. A comprehensive study of update processes is beyond the scope of this class, but a simple example can be considered.

A can of soda at temperature 25° C is placed in a refrigerator, where the ambient temperature F is 10° C. We want to determine how the temperature of the soda changes over a period of time. A standard way of approaching this type of problem is to subdivide the time interval into a number of small steps, each of duration Δt. If T_i is the temperature at the *beginning* of step i, the following model can be used to determine T_{i+1}:

$$T_{i+1} = T_i + K\Delta t(F - T_i)$$

where K is the *conduction coefficient*, a parameter that depends on the insulating properties of the can and the thermal properties of the soda. Assume that units are chosen so that time is in minutes and that an interval $\Delta t = 1$ minute provides sufficient accuracy. A script to compute, display, and plot this update process for $K = 0.05$:

```
% Define input values
K = 0.05;                          % Conduction coefficient
F = 10;                            % Refrigerator temperature (degrees C)

% Define vector variables
t = 0:100;                         % Time variable (min)
T = zeros(1,101);                  % Preallocate temperature vector
T(1) = 25;                         % Initial soda temperature (degrees C)

% Update to compute T
for i = 1:100;                     % Time in minutes
  T(i+1) = T(i) + K * (F - T(i));  % Compute T
end

% Display results every 10 minutes, plot every minute
disp([ t(1:10:101)' T(1:10:101)' ])
plot(t,T),grid,xlabel('Time (min)'),ylabel('Temperature (degrees C)'),...
  title('Cooling curve')
```

The statement `T = zeros(1,101)` preallocates a vector for the temperature of the soda. The script would work without this statement, but it would be much slower, as `T` would have to be *redimensioned* during each repeat of the `for` loop, in order to make space for a new element each time.

The `for` loop computes the values for `T(2),...,T(101)`, ensuring that temperature `T(i)` corresponds to `t(i)`.

The displayed output of the script is:

```
         0   25.0000
   10.0000   18.9811
   20.0000   15.3773
   30.0000   13.2196
   40.0000   11.9277
   50.0000   11.1542
   60.0000   10.6910
   70.0000   10.4138
   80.0000   10.2477
   90.0000   10.1483
  100.0000   10.0888
```

The resulting graph is shown in Figure 8.2. Note that the initial slope of the temperature is steep and that temperature of the soda changes rapidly. As the difference between the soda temperature and the refrigerator temperature becomes smaller, that change in soda temperature slows.

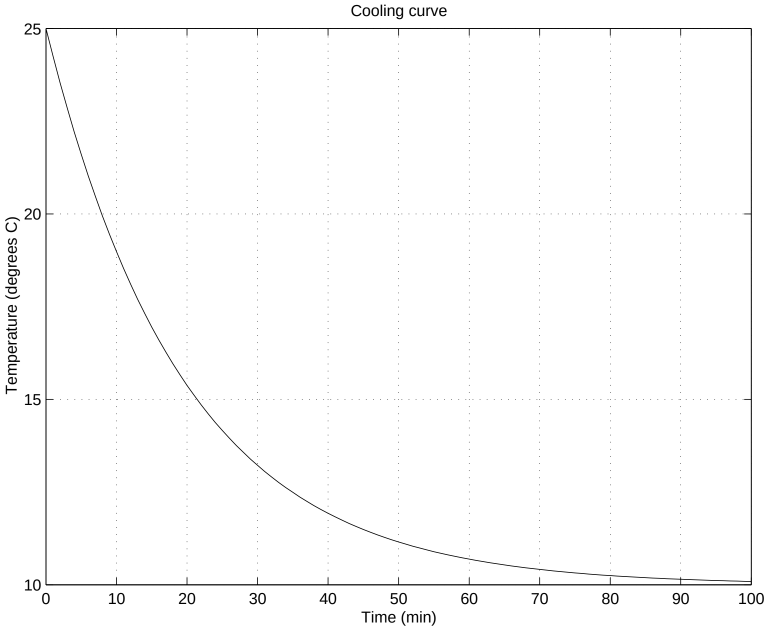


Figure 8.2: Soda cooling curve

Exact Solution

This cooling problem has an exact mathematical solution, which is determined by expressing the update relationship as a differential equation and then solving the resulting differential equation. The result is

$$T(t) = F + (T_0 - F)e^{-Kt}$$

where T_0 is the initial temperature. This is a decaying exponential, which you an confirm as having the same properties as the cooling curve shown in Figure 8.2.

8.5.1 Signal Filtering

Related to update processes are methods for filtering signals to reduce noise components. As described in Example 7.6, the noise is often modeled as a random measurement error. The noise can be reduced by averaging several successive signal samples. Denoting a noisy input signal sample by $x(k)$ and a filtered output signal sample by $y(k)$, a *3-point moving average filter* is represented by

$$y(k) = \frac{1}{3}\left[x(k) + x(k-1) + x(k-2)\right]$$

The output signal sample at index k is the average of the input signal samples at index k and the two previous indices.

To test this filtering method, consider the following script in which a noisy signal is simulated and filtered by a moving average.

```
% Noisy signal generation and filtering
t = linspace(0,10,512);            % time base
s = sin(2*pi/5*t);                 % signal
n = 0.1*randn(size(t));            % noise, std dev 0.1
x = s + n;                         % signal + noise
disp('Input Signal to Noise Ratio (SNR), dB')
SNRin = 20*log10(std(s)/std(n))    % input SNR, dB
y = zeros(size(t));                % initialize output signal

% filtering computation
y(1) = x(1);
y(2) = (x(2)+x(1))/2;
for k = 3:length(t);
  y(k) = (x(k)+x(k-1)+x(k-2))/3;
end

% analysis of filtered signal

disp('Output Signal to Noise Ratio (SNR), dB')
SNRout = 20*log10(std(s)/std(y-s))  % output SNR, dB

% plot signals
subplot(2,1,1),plot(t,x), xlabel('Time (s)'),...
  ylabel('Signal amplitude'),...
  title('Input signal')
subplot(2,1,2),plot(t,y), xlabel('Time (s)'),...
```

```
  ylabel('Signal amplitude'),...
  title('Output signal')
```

Note that in computing the first two filtered signal samples, two previous input signal samples are not available, so shortened averages are required. Signal quality is measured by signal to noise ratio (SNR), where the noise remaining in the filtered signal is the difference between the filtered signal and the original noise-free signal.

The improvement in the signal is apparent in Figure 8.3. The displayed output below shows that the SNR has been improved by about 4.3 dB.

```
Input Signal to Noise Ratio (SNR), dB
SNRin =
   17.5063
Output Signal to Noise Ratio (SNR), dB
SNRout =
   21.8317
```

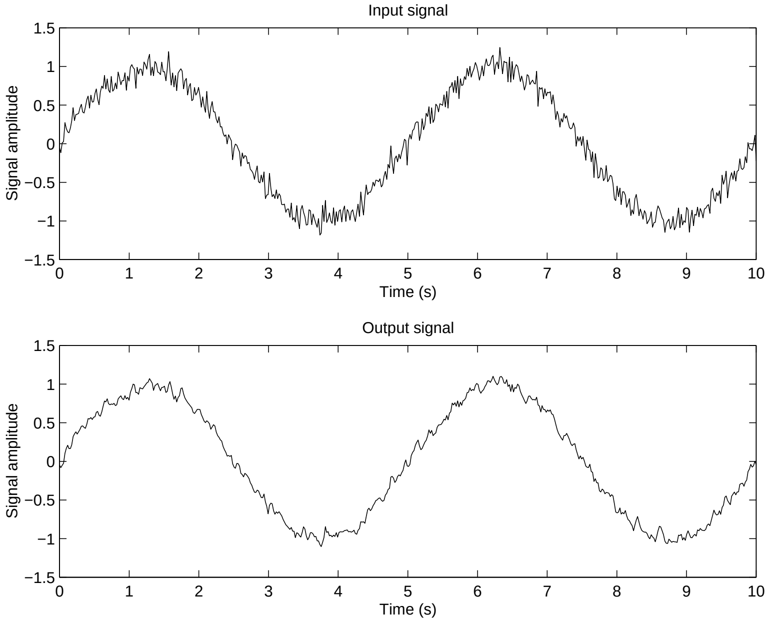


Figure 8.3: 3-point moving average signal filtering

If more input signal samples were averaged, the noise would be further reduced. However, this would also lead to distortion of the original signal. Thus, filtering functions must be designed to compromise between noise reduction and signal distortion.

More effective filtering functions and better implementations of filtering in Matlab are available, but these are beyond the scope of this presentation, which is intended as a short introduction to filtering.

8.6 Applied Problem Solving: Speech Signal Analysis

Consider the design of a system to recognize the spoken words for the ten digits: “zero,” “one,” “two,” ..., “nine.” A first step in the design is to analyze data sequences collected with a microphone to see if there are some statistical measurements that would allow recognition of the digits.

The statistical measurements should include the mean, variance, average magnitude, average power and number of zero crossings. If the data sequence is $x(n)$, $n = 1, \ldots, N$, these measurements are defined as follows:

$$\text{mean} = \mu = \frac{1}{N}\sum_{n=1}^{N} x(n)$$

$$\text{standard deviation} = \sigma = \left[\frac{1}{N}\sum_{n=1}^{N}(x(n)-\mu)^2\right]^{\frac{1}{2}}$$

$$\text{average magnitude} = \frac{1}{N}\sum_{n=1}^{N}|x(n)|$$

$$\text{average power} = \frac{1}{N}\sum_{n=1}^{N}(x(n))^2$$

The number of zero crossings is the number of times that $x(k)$ and $x(k+1)$ differ in sign, or the number of times that $x(k) \cdot x(k+1) < 0$.

Matlab functions for reading and writing Microsoft wave (“.wav”) sound files include:

Function	Description
`x = wavread(wavefile)`	Reads a wave sound file specified by the string `wavefile`, returning the sampled data in `x`. The “.wav” extension is appended if no extension is given. Amplitude values are in the range [-1,+1].
`[x,fs,bits]=wavread(wavefile)`	Returns the sample rate (`fs`) in Hertz and the number of bits per sample (`bits`) used to encode the data in the file.
`wavwrite(x,wavefile)`	Writes a wave sound file specified by the string `wavfile`. The data should be arranged with one channel per column. Amplitude values outside the range [-1,+1] are clipped prior to writing.
`wavwrite(x,fs,wavefile)`	Specifies the sample rate `fs` of the data in Hertz.

The following is a script (“digit.m”) to prompt the user to specify the name of a wave file to be read and then to compute and display the statistics and to plot the sound sequence and a histogram of the sequence. To compute the number of zero crossings, a vector `prod` is generated whose first element is x(1)*x(2), whose second element is x(2)*x(3), and so on, with the last value equal to the product of the next-to-the-last element and the last element. The find function is used to determine the locations of prod that are negative, and length is used to count the number of these negative products. The histogram has 51 bins between −1.0 and 1.0, with the bin centers computed using linspace.

```
% Script to read a wave sound file named "digit.wav"
% and to compute several speech statistics
%
file = input('Enter name of wave file as a string: ');
[x,fs,bits] = wavread(file);
n = length(x);
%
fprintf('\n')
fprintf('Digit Statistics \n\n')
fprintf('samples: %.0f \n',n)
fprintf('sampling frequency: %.1f \n',fs)
fprintf('bits per sample: %.0f \n',bits)
fprintf('mean: %.4f \n',mean(x))
fprintf('standard deviation: %.4f \n', std(x))
fprintf('average magnitude: %.4f \n', mean(abs(x)))
fprintf('average power: %.4f \n', mean(x.^2))
prod = x(1:n-1).*x(2:n);
crossings = length(find(prod<0));
fprintf('zero crossings: %.0f \n', crossings)
subplot(2,1,1),plot(x),...
  axis([1 n -1.0 1.0]),...
  title('Data sequence of spoken digit'),...
  xlabel('Index'), grid,...
subplot(2,1,2),hist(x,linspace(-1,1,51)),...
  axis([-0.6,0.6,0,5000]),...
  title('Histogram of data sequence'),...
  xlabel('Sound amplitude'),...
  ylabel('Number of samples'),grid
```

For a hand example to be used to test the script to be developed for this problem, consider the sequence:

$$[0.25 \quad 0.82 \quad -0.11 \quad -0.02 \quad 0.15]$$

Using a calculator, we compute the following:

$$\text{mean} = \mu = \frac{1}{5}(0.25 + 0.82 - 0.11 - 0.02 + 0.15) = 0.218$$

$$\text{standard deviation} = \sigma \quad = \quad \Bigg[\frac{1}{5}\Big((0.25 - \mu)^2 + (0.82 - \mu)^2 + (-0.11 - \mu)^2$$

$$+ (-0.02 - \mu)^2 + (0.15 - \mu)^2\Big)\Big]^{\frac{1}{2}} = 0.365$$

$$\text{average magnitude} = \frac{1}{5}\left(|0.25| + |0.82| + |-0.11| + |-0.02| + |0.15|\right) = 0.270$$

$$\text{average power} = \frac{1}{5}\left(0.25^2 + 0.82^2 + (-0.11)^2 + (-0.02)^2 + 0.15^2\right) = 0.154$$

$$\text{number of zero crossings} = 2$$

The MATLAB commands to compute and write a wave file `test.wav` containing the test sequence:

```
>> x = [0.25 0.82 -0.11 -0.02 0.15]
>> wavwrite(x,'test.wav')
```

Executing the script `digit.m` on the test data:

```
>> digit
Enter name of wave file as a string: 'test.wav'

Digit Statistics

samples: 5
sampling frequency: 8000.0
bits per sample: 16
mean: 0.2180
standard deviation: 0.3648
average magnitude: 0.2700
average power: 0.1540
zero crossings: 2
```

Observe that the results from the script agree with the hand-calculated values, giving us some confidence in the script.

Executing the script on a wave file `zero.wav` created using the "Sound Recorder" program on a PC to record the spoken word "zero" as a wave file:

```
>> digit
Enter name of wave file as a string: 'zero.wav'

Digit Statistics

samples: 13493
sampling frequency: 11025.0
```

```
bits per sample: 8
mean: -0.0155
standard deviation: 0.1000
average magnitude: 0.0690
average power: 0.0102
zero crossings: 271
```

The resulting plots are shown in Figure 8.4.

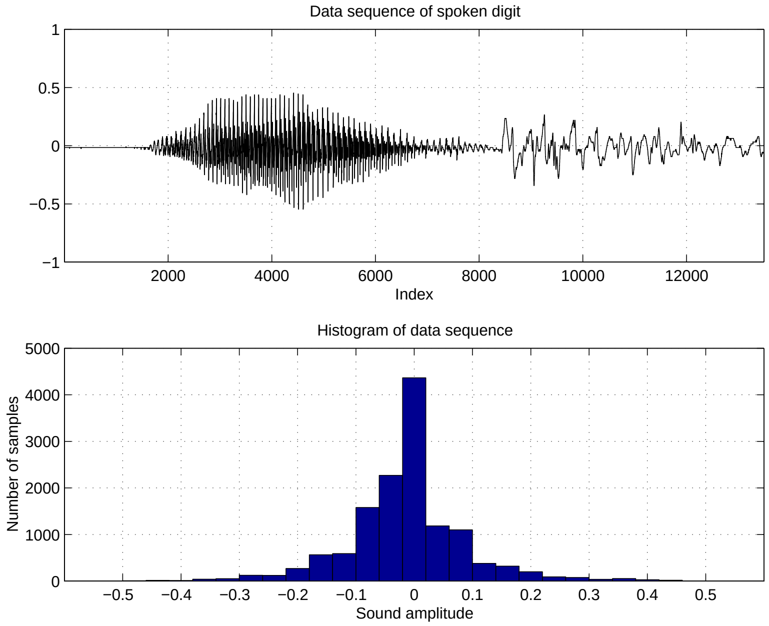


Figure 8.4: Data sequence and histogram for the spoken word ‘zero’

Observe that the characteristics of the change with time (index), as different sounds are being made to produce the word. An effective word recognition algorithm must capture this change in sounds, but the design of such an algorithm is beyond the material covered in this course. Note that the histogram has the appearance of a Gaussian or bell-shaped curve, with a mean near zero and a standard deviation of about 0.35, while the data sequence looks much different than the Gaussian random data sequence shown in Figure 7.1.

You can experiment with the PC Sound Recorder program to record wave files for other spoken digits and then process them with `digit`. In recording these files, you will need to make sure that the number of data samples in the recorded file is less than 16,384 if you are using the Student Version of Matlab. This requires using the “Telephone Quality” recording option and you may need to delete the silent sections at the beginning and end of the record.

You can also create wave files using Matlab and then listen to the signals in these files using PC Sound Recorder. For example, to compute and write a wave file containing 16000 samples in 2 seconds of an 800Hz sinusoid:

```
>> t = linspace(0,2,16000);
>> y = cos(2*pi*800*t);
>> wavwrite(y,'cosine.wav')
```

Section 9

Vectors, Matrices and Linear Algebra

A **matrix** is a two-dimensional array, as introduced in Section 7.2, where matrix addressing and methods of array creation were described. In Section 7.3, scalar-array mathematics and element-by-element array mathematics were described. In this chapter, we present a set of matrix operations and functions that apply to the matrix as a unit, as opposed to individual elements in the matrix.

Review of Matrix Definitions

1. **Scalar:** A matrix with one row and one column.

$$A = [3.5]$$

2. **Vector:** A matrix with one row (a **row vector**) or one column, (a **column vector**).

$$B_1 = \begin{bmatrix} 1.5 & 3.1 \end{bmatrix} \qquad B_2 = \begin{bmatrix} 2.5 \\ 3.1 \end{bmatrix}$$

3. **Matrix:** An example of a matrix with four rows and three columns:

$$C = \begin{bmatrix} -1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & -1 & 0 \\ 0 & 0 & -2 \end{bmatrix}$$

The elements in a matrix are denoted by c_{ij}, where i is the row index and j is the column index. For example c_{43} refers to the element in row 4 and column 3, and thus, $c_{43} = -2$ in the example above. If needed to avoid misinterpretation, the indexes can be separated by a comma, as in $c_{i,j}$ and $c_{4,3}$. In Matlab, subscripts are indicated by parentheses, as in `C(4,3)`.

9.1 Vectors

We usually think of vectors as column vectors, written as

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

To indicate that $\mathbf{x}$ is a vector of n real numbers, we write

$$\mathbf{x} \in \mathcal{R}^n$$

Geometrically, $\mathcal{R}^n$ is n-dimensional space, and the notation $\mathbf{x} \in \mathcal{R}^n$ means that $\mathbf{x}$ is a point in that space, specified by the n coordinates $x_1, x_2, \ldots x_n$. Figure 9.1 shows a vector in $\mathcal{R}^3$, drawn as an arrow from the origin to the point $\mathbf{x}$.

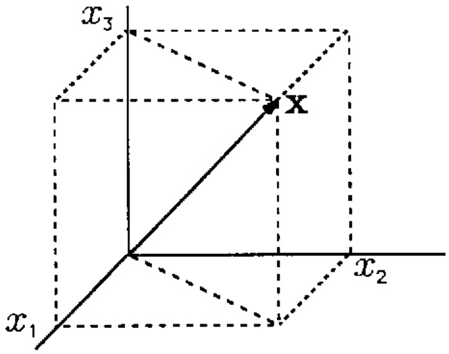


Figure 9.1: A vector in $\mathcal{R}^3$

Vector Addition

Vectors with the same number of elements can be added and subtracted in a very natural way:

$$\mathbf{x} + \mathbf{y} = \begin{bmatrix} x_1 + y_1 \\ x_2 + y_2 \\ x_3 + y_3 \\ \vdots \\ x_n + y_n \end{bmatrix}, \qquad \mathbf{x} - \mathbf{y} = \begin{bmatrix} x_1 - y_1 \\ x_2 - y_2 \\ x_3 - y_3 \\ \vdots \\ x_n - y_n \end{bmatrix}$$

Inner or Dot Product

The inner product $(\mathbf{x}, \mathbf{y})$, also referred to as the dot product $\mathbf{x} \cdot \mathbf{y}$, is a scalar sum of products

$$(\mathbf{x}, \mathbf{y}) = \mathbf{x} \cdot \mathbf{y} = x_1y_1 + x_2y_2 + x_3y_3 + \cdots + x_ny_n$$

For example, for

$$\mathbf{x} = \begin{bmatrix} 4 \\ -1 \\ 3 \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} -2 \\ 5 \\ 2 \end{bmatrix},$$

the inner product is

$$(\mathbf{x}, \mathbf{y}) = 4 \cdot (-2) + (-1) \cdot 5 + 3 \cdot 2 = -7$$

Properties:

1. $(\mathbf{x}, \mathbf{y}) = (\mathbf{y}, \mathbf{x})$
2. $(a\mathbf{x}, \mathbf{y}) = a(\mathbf{x}, \mathbf{y}) = (\mathbf{x}, a\mathbf{y})$
3. $(\mathbf{x}, \mathbf{y} + \mathbf{z}) = (\mathbf{x}, \mathbf{y}) + (\mathbf{x}, \mathbf{z})$

In MATLAB, the inner product is computed with the `dot` function:

`dot(x,y)` Returns the scalar dot product of the vectors `x` and `y`, which must be of the same length.

`dot(A,B)` Returns a row vector of the dot products for the corresponding columns of matrices `A` and `B`.

Example:

```
>> x = [4 -1 3]'
x =
     4
    -1
     3
>> y = [-2 5 2]'
y =
    -2
     5
     2
>> d = dot(x,y)
d =
    -7
```

Euclidean Norm

The length of a vector is called the **norm** of the vector. From Euclidean geometry, the distance between two points is the square root of the sum of the squares of the distances in each dimension. Thus, the notation and definition of the Euclidean norm is

$$||\mathbf{x}|| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$$

Note that the norm can be defined in terms of the inner product:

$$||\mathbf{x}|| = \sqrt{(\mathbf{x}, \mathbf{x})}$$

For example, the norm of **x** above is

$$||\mathbf{x}|| = \sqrt{4^2 + (-1)^2 + 3^2} = 5.1$$

In MATLAB, `norm(x)` returns the Euclidean norm of row vector or column vector `x`.

For example, for `x` and `y` as defined above:

```
>> nx = norm(x)
nx =
    5.0990
>> ny = norm(y)
ny =
    5.7446
```

Triangle Inequality

This inequality, also called the Cauchy-Bernoulli-Schwarz (CBS) inequality, states that the absolute value of the inner product of vectors **x** and **y** is less than or equal to the norm of **x** times the norm of **y**, with equality if and only if $\mathbf{y} = \alpha\mathbf{x}$

$$|(\mathbf{x}, \mathbf{y})| \leq ||\mathbf{x}||\ ||\mathbf{y}||$$

Confirming this using the examples above:

```
>> abs(dot(x,y))
ans =
     7
>> norm(x)*norm(y)
ans =
   29.2916
```

To confirm the equality condition, define $\mathbf{z} = 5\mathbf{x}$

```
>> z = 5*x
z =
    20
    -5
    15
>> abs(dot(x,z))
ans =
   130
>> norm(x)*norm(z)
ans =
   130
```

Unit Vectors

The unit vector $\mathbf{u}_x$ is a vector of length one pointing in the same direction as $\mathbf{x}$

$$\mathbf{u}_x = \frac{1}{||\mathbf{x}||}\mathbf{x}$$

```
>> ux = x/norm(x)
ux =
    0.7845
   -0.1961
    0.5883
>> norm(ux)
ans =
     1
```

Angle Between Vectors

The angle θ between two vectors $\mathbf{x}$ and $\mathbf{y}$ is

$$\cos\theta = \frac{(\mathbf{x},\mathbf{y})}{||\mathbf{x}||\,||\mathbf{y}||}$$

```
>> theta = acos(dot(x,y)/(norm(x)*norm(y)))
theta =
    1.8121
>> thetad = (180/pi)*theta
thetad =
  103.8261
```

Orthogonality

When the angle between two vectors is $\pi/2$ (90°), the vectors are said to be **orthogonal**. From the equation for the angle between two vectors

$$(\mathbf{x}, \mathbf{y}) = 0 \iff \mathbf{x} \text{ and } \mathbf{y} \text{ are orthogonal}$$

Projection

The projection of vector $\mathbf{x}$ onto $\mathbf{y}$ is found by dropping a perpendicular from the head of $\mathbf{x}$ onto the line representing $\mathbf{y}$, as shown in Figure 9.2. The point where the perpendicular intersects $\mathbf{y}$ (or an extension of $\mathbf{y}$) is the projection of $\mathbf{x}$ onto $\mathbf{y}$, or the component of $\mathbf{x}$ in the direction of $\mathbf{y}$. Denoting the projection vector by $\mathbf{z}$

$$\mathbf{z} = \frac{(\mathbf{x}, \mathbf{y})}{(\mathbf{y}, \mathbf{y})}\mathbf{y} = \frac{(\mathbf{x}, \mathbf{y})}{||\mathbf{y}||^2}\mathbf{y}$$

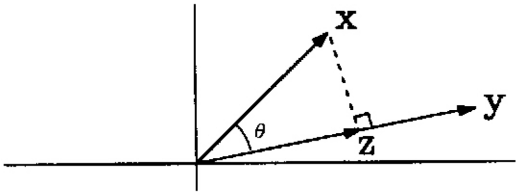


Figure 9.2: Projection of $\mathbf{x}$ onto $\mathbf{y}$

```
>> z = (dot(x,y)/norm(y)^2)*y
z =
    0.4242
   -1.0606
   -0.4242
```

Example 9.1 *Ship course*

Figure 9.1 shows a ship sailing on bearing 315° (NW) with a speed of 20 knots. The local current due to the tide is 2 knots in the direction 67.5° (ENE). The ship also drifts at 0.5 knots under wind in the direction 180°.

The following script is written to do the following:

- Represent the ship velocity as a vector `S`, the current velocity as a vector `C`, and the wind-drift velocity as a vector `W`. Calculate the true-velocity vector `T` (velocity over the ocean bottom). North represents the first component and East the second component of the vectors.

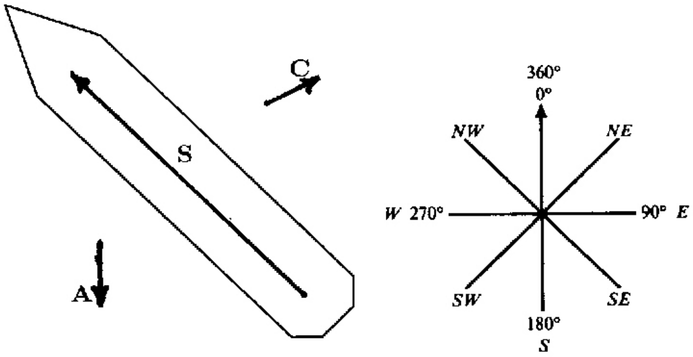


Figure 9.3: Ship course components

- Calculate the true ship speed, that is, the ship speed over the ocean bottom.
- Calculate the true ship course, that is, the direction of sailing over the ocean bottom, measured in degrees clockwise from north.

```
disp('Ship velocity vector:')
S = 20*[cos(315*pi/180) sin(315*pi/180)]     % ship vector
disp('Current velocity vector:')
C = 2*[cos(67.5*pi/180) sin(67.5*pi/180)]    % current vector
disp('Wind drift velocity vector:')
W = 0.5*[-1 0]                               % wind vector
disp('True velocity vector:')
T = S + C + W
disp('Ship speed (knots):')
speed = norm(T)
course = atan2(T(2),T(1))*180/pi;            % ship course, degrees
if course < 0
   course = 360 + course;                    % correct course if negative
end
disp('Ship course (degrees)')
course
```

The function atan2 was used to compute the course heading, as it provides a four-quadrant result. However, it produces results in the range $-\pi$ to π, which have to be converted to degrees in the range 0 to 360.

The displayed results:

```
Ship velocity vector:
S =
   14.1421  -14.1421
Current velocity vector:
```

```
C =
    0.7654    1.8478
Wind drift velocity vector:
W =
   -0.5000         0
True velocity vector:
T =
   14.4075  -12.2944
Ship speed (knots):
speed =
   18.9401
Ship course (degrees)
course =
  319.5248
```

■

9.2 Matrices

A matrix is denoted by a boldfaced capital letter having elements in the corresponding lower-case letter having double subscripts for the row and column. An $m \times n$ (read m by n) matrix, having m rows and n columns

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \dots & a_{mn} \end{bmatrix}$$

Scalar element a_{ij} is located in row i and column j. A **square matrix** has the same number of row and columns ($m = n$). The **main diagonal** of a matrix consists of the elements a_{ii} (the two subscripts are equal). The main diagonal runs diagonally down and away from the top left corner of the matrix, but it does not end in the lower right corner unless the matrix is square.

Transpose

The **transpose** of matrix **A** is another matrix **B** such that the element in row j and column i is $b_{ji} = a_{ij}$, for $1 \leq i \leq m$ and $1 \leq j \leq n$. The notation for transpose is $\mathbf{B} = \mathbf{A}^T = \mathbf{A}'$. In MATLAB, transpose is denoted by `A'`. A more intuitive way of describing the transpose operation is to say that it flips the matrix about its main diagonal so that rows become columns and columns become rows. For example:

```
>> A = [2 1; 5 4; 7 9]
```

```
A =
     2     1
     5     4
     7     9
>> B = A'
B =
     2     5     7
     1     4     9
```

Identity Matrix

An **identity matrix** is a matrix with ones on the main diagonal and zeros elsewhere. The following is an identity matrix with four rows and four columns

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To generate an identity matrix in Matlab:

`eye(n)`	Returns an `n` × `n` identity matrix.
`eye(m,n)`	Returns an `m` × `n` matrix with ones on the main diagonal and zeros elsewhere.
`eye(size(a))`	Returns a matrix with ones on the main diagonal and zeros elsewhere that is the same size as `A`.

```
>> I = eye(3)
I =
     1     0     0
     0     1     0
     0     0     1
```

The variable name `i` should not be used for an identity matrix in Matlab, because `i` will not represent $\sqrt{-1}$ in statements that follow.

Matrix Addition and Scalar Multiplication

Two matrices of the same dimensions may be added or subtracted in the same way as vectors, by adding or subtracting the corresponding elements. The equation $\mathbf{C} = \mathbf{A} \pm \mathbf{B}$ means that for each i and j, $c_{ij} = a_{ij} \pm b_{ij}$.

Scalar multiplication of a matrix multiplies each element of the matrix by the scalar:

$$\alpha\mathbf{A} = \begin{bmatrix} \alpha a_{11} & \alpha a_{12} & \alpha a_{13} & \dots & \alpha a_{1n} \\ \alpha a_{21} & \alpha a_{22} & \alpha a_{23} & \dots & \alpha a_{2n} \\ \alpha a_{31} & \alpha a_{32} & \alpha a_{33} & \dots & \alpha a_{3n} \\ \vdots & \vdots & \vdots & & \vdots \\ \alpha a_{m1} & \alpha a_{m2} & \alpha a_{m3} & \dots & \alpha a_{mn} \end{bmatrix}$$

Matrix Multiplication

The **matrix multiplication** of $m \times n$ matrix $\mathbf{A}$ and $n \times p$ matrix $\mathbf{B}$ yields $m \times p$ matrix $\mathbf{C}$, denoted by

$$\mathbf{C} = \mathbf{AB} \tag{9.1}$$

Element c_{ij} is the inner product of row i of $\mathbf{A}$ and column j of $\mathbf{B}$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \tag{9.2}$$

To better understand this operation, represent a matrix as a collection of vectors. For example, each row of $\mathbf{A}$ can be represented as being a vector

$$\mathbf{A} = \begin{bmatrix} - & \mathbf{a}_1^T & - \\ - & \mathbf{a}_2^T & - \\ & \vdots & \\ - & \mathbf{a}_m^T & - \end{bmatrix} = \begin{bmatrix} [a_{11} & a_{12} & \cdots & a_{1n}] \\ [a_{21} & a_{22} & \cdots & a_{2n}] \\ & & \vdots & \\ [a_{m1} & a_{m2} & \cdots & a_{mn}] \end{bmatrix} \tag{9.3}$$

Alternatively, each column of $\mathbf{B}$ can be represented as a vector

$$\mathbf{B} = \begin{bmatrix} | & | & & | \\ \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_p \\ | & | & & | \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} b_{11} \\ b_{21} \\ \vdots \\ b_{n1} \end{bmatrix} & \begin{bmatrix} b_{12} \\ b_{22} \\ \vdots \\ b_{n2} \end{bmatrix} & \cdots & \begin{bmatrix} b_{1p} \\ b_{2p} \\ \vdots \\ b_{np} \end{bmatrix} \end{bmatrix} \tag{9.4}$$

Matrix-matrix multiplication can now be defined in terms of inner products of these vectors. An n-element column vector $\mathbf{x}$ is an $n \times 1$ matrix, whose transpose $\mathbf{x}^T$ is a $1 \times n$ matrix, or n-element row vector

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}; \quad \mathbf{x}^T = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}$$

Consider the matrix multiplication $\mathbf{x}^T\mathbf{y}$, where $\mathbf{x}$ and $\mathbf{y}$ are n-element column vectors

$$\mathbf{x}^T\mathbf{y} = \left[\begin{array}{cccc} x_1 & x_2 & \cdots & x_n \end{array}\right] \left[\begin{array}{c} y_1 \\ y_2 \\ \vdots \\ y_n \end{array}\right]$$

Applying equation (9.1), let $\mathbf{A} = \mathbf{x}^T$, and $\mathbf{B} = \mathbf{y}$, with $m = p = 1$, and the result from equation (9.2) is the scalar

$$\mathbf{x}^T\mathbf{y} = \sum_{k=1}^{n} x_k y_k = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n$$

Observe that this is the inner product of $\mathbf{x}$ and $\mathbf{y}$

$$\mathbf{x}^T\mathbf{y} = (\mathbf{x}, \mathbf{y})$$

Applying this result to equation (9.1) and using representation of matrices $\mathbf{A}$ and $\mathbf{B}$ as the collections of vectors defined in equations (9.3) and (9.4)

$$c_{ij} = \mathbf{a}_i^T\mathbf{b}_j = (\mathbf{a}_i, \mathbf{b}_j)$$

Thus,

$$\mathbf{C} = \mathbf{AB} = \left[\begin{array}{cccc} \mathbf{a}_1^T\mathbf{b}_1 & \mathbf{a}_1^T\mathbf{b}_2 & \cdots & \mathbf{a}_1^T\mathbf{b}_p \\ \mathbf{a}_2^T\mathbf{b}_1 & \mathbf{a}_2^T\mathbf{b}_2 & \cdots & \mathbf{a}_2^T\mathbf{b}_p \\ \vdots & \vdots & & \vdots \\ \mathbf{a}_m^T\mathbf{b}_1 & \mathbf{a}_m^T\mathbf{b}_2 & \cdots & \mathbf{a}_m^T\mathbf{b}_p \end{array}\right]$$

For example, consider the following matrices

$$\mathbf{A} = \left[\begin{array}{rrr} 2 & 5 & 1 \\ 0 & 3 & -1 \end{array}\right] \qquad \mathbf{B} = \left[\begin{array}{rrr} 1 & 0 & 2 \\ -1 & 4 & -2 \\ 5 & 2 & 1 \end{array}\right]$$

where $m \times n = 2 \times 3$ and $n \times p = 3 \times 3$. Computing elements of $\mathbf{C}$:

$$c_{11} = \left[\begin{array}{ccc} 2 & 5 & 1 \end{array}\right] \left[\begin{array}{r} 1 \\ -1 \\ 5 \end{array}\right] = 2 \cdot 1 + 5 \cdot (-1) + 1 \cdot 5 = 2$$

$$c_{23} = \left[\begin{array}{ccc} 0 & 3 & -1 \end{array}\right] \left[\begin{array}{r} 2 \\ -2 \\ 1 \end{array}\right] = 0 \cdot 2 + 3 \cdot (-2) - 1 \cdot 1 = -7$$

Similarly, the rest of the elements can be computed to yield

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} 2 & 22 & -5 \\ -8 & 10 & -7 \end{bmatrix}$$

To determine if a matrix product exists and if so, the size of the resulting matrix, write the matrix sizes side-by-side

$$(m \times n) \cdot (n \times p) = (m \times p)$$

The two center sizes (columns of **A** and rows of **B**) must be equal for the product to exist. For the example above

$$(2 \times 3) \cdot (3 \times 3) = (2 \times 3)$$

Thus **AB** exists and is of size 2 × 3. If we wanted to compute **BA**, we again write the sizes side-by-side

$$(3 \times 3) \cdot (2 \times 3)$$

The two inner numbers are not the same, so **BA** does not exist. Note that this implies that matrix multiplication does not commute

$$\mathbf{AB} \neq \mathbf{BA}$$

In Matlab matrix multiplication is denoted by an asterisk.

```
>> A = [2,5,1; 0,3,-1]
A =
     2     5     1
     0     3    -1
>> B = [1,0,2; -1,4,-2; 5,2,1]
B =
     1     0     2
    -1     4    -2
     5     2     1
>> C = A*B
C =
     2    22    -5
    -8    10    -7
>> C = B*A
??? Error using ==> *
Inner matrix dimensions must agree.
```

Another matrix multiplication involving two vectors is the **outer product**, which is the product of a row vector and a column vector. If $\mathbf{x}$ and $\mathbf{y}$ are n-element column vectors, then the product $\mathbf{x}\mathbf{y}^T$ is a $n \times 1$ matrix multiplying an $1 \times n$ matrix, yielding an $n \times n$ result

$$\mathbf{x}\mathbf{y}^T = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \begin{bmatrix} y_1 & y_2 & \cdots & y_n \end{bmatrix} = \begin{bmatrix} x_1y_1 & x_1y_2 & \cdots & x_1y_n \\ x_2y_1 & x_2y_2 & \cdots & x_2y_n \\ \vdots & \vdots & & \vdots \\ x_my_1 & x_my_2 & \cdots & x_my_n \end{bmatrix}$$

In the outer product, the inner products that define its elements are between the one-dimensional row vectors of $\mathbf{x}$ and the one-dimensional column vectors of $\mathbf{y}^T$.

If $\mathbf{I}$ is an $n \times n$ identity matrix and $\mathbf{A}$ is an $n \times n$ matrix, then

$$\mathbf{IA} = \mathbf{AI} = \mathbf{A}$$

Confirming

```
>> A = [1 2 3;4 5 6;7 8 9]
A =
     1     2     3
     4     5     6
     7     8     9
>> I = eye(3)
I =
     1     0     0
     0     1     0
     0     0     1
>> I*A
ans =
     1     2     3
     4     5     6
     7     8     9
>> A*I
ans =
     1     2     3
     4     5     6
     7     8     9
```

Matrix Inverse

Two scalars are called inverses when their product is 1, such as 5 and 0.2. Two *square* matrices are *inverses* of one another if their product is the identity matrix

$$\mathbf{AB} = \mathbf{I} \quad \Longleftrightarrow \quad \mathbf{B} = \mathbf{A}^{-1}$$

Multiplying the first form by **B** from the left

$$\mathbf{B}(\mathbf{AB}) = \mathbf{BI} = \mathbf{B}$$

Multiplying in a different order

$$(\mathbf{BA})\mathbf{B} = \mathbf{B}$$

We can then conclude

$$\mathbf{BA} = \mathbf{I}$$

Thus, while matrix multiplication is not commutative in general, this is a special case where it is commutative. If **A** and **B** are inverses of each other, then

$$\mathbf{AB} = \mathbf{BA} = \mathbf{I}$$

The inverse for an **ill-conditioned** or **singular** matrix does not exist. The **rank** of a matrix is the number of independent equations represented by the rows of the of the matrix. If the rank of a matrix is equal to the number of its rows, the matrix is **nonsingular** and its inverse exists.

Matlab functions for matrix rank and inverse:

`rank(A)` Returns the rank of the matrix `A`, which is the number of independent rows of `A`.

`inv(A)` Returns the inverse of the matrix `A`, which must be square and have a rank equal to the number of rows. If the inverse does not exist, an error message is printed.

Matrix inverse can also be computed using the expression `A^(-1)`.

Example:

```
>> A = [2,1; 4,3]
A =
     2     1
     4     3
>> rank(A)
ans =
     2
>> B = inv(A)
B =
    1.5000   -0.5000
   -2.0000    1.0000
>> C = A*B
C =
     1     0
     0     1
>> D = A^(-1)
D =
    1.5000   -0.5000
   -2.0000    1.0000
```

The following commands determine the size and rank of a matrix **A** and then either compute the inverse of **A** or print a message indicating that the matrix inverse does not exist:

```
[nr,nc] = size(A);
if nr ~= nc
  disp('Inverse does not exist, matrix is not square')
elseif rank(A) ~= nr
  disp('Inverse does not exist, rank not equal to number of rows')
else
  disp('Matrix inverse:')
  disp(inv(A))
end
```

As shown in this example, it is good programming practice to avoid performing computations that would generate Matlab error messages by including error checks that provide more specific messages.

Matrix Powers

If `A` is a matrix, then `A.^2` is the expression that squares each element in the matrix. Thus,

$$\texttt{A.^2} = \begin{bmatrix} a_{11}^2 & a_{12}^2 & \cdots & a_{1n}^2 \\ a_{21}^2 & a_{22}^2 & \cdots & a_{2n}^2 \\ \vdots & \vdots & & \vdots \\ a_{m1}^2 & a_{m2}^2 & \cdots & a_{mn}^2 \end{bmatrix}$$

To square the matrix, that is, to compute `A*A`, the expression `A^2` can be used. Then `A^4` is equivalent to `A*A*A*A`. To perform a matrix multiplication between two matrices, the number of rows in the first matrix must be the same as the number of columns in the second matrix; therefore, to raise a matrix to a power, the matrix must be square.

Determinant

A **determinant** is a scalar computed from a square matrix. While there are many applications of determinants, a full explanation is beyond the intent of this presentation. For a 2 × 2 matrix **A**, the determinant is

$$|\mathbf{A}| = a_{11}a_{22} - a_{12}a_{21}$$

Thus, for

$$\mathbf{A} = \begin{bmatrix} 1 & 3 \\ -1 & 5 \end{bmatrix}$$

the determinant is

$$|\mathbf{A}| = 1 \cdot 5 - 3 \cdot (-1) = 8$$

For a 3 × 3 matrix **A**, the determinant is

$$\begin{aligned} |\mathbf{A}| &= a_{11}(a_{22}a_{33} - a_{23}a_{32}) - a_{12}(a_{21}a_{33} - a_{23}a_{31}) + a_{13}(a_{21}a_{32} - a_{22}a_{31}) \\ &= a_{11}a_{22}a_{33} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{13}a_{22}a_{31} \end{aligned}$$

Thus, for

$$\mathbf{B} = \begin{bmatrix} 1 & 3 & 0 \\ -1 & 5 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

the determinant is

$$|\mathbf{B}| = 1(5 - 4) - 3(-1 - 2) + 0(-1 - 5) = 1 + 9 = 10$$

If the determinant of a matrix is zero, the matrix is said to be **singular** and its inverse does not exist.

The MATLAB function `det(A)` computes the determinant of the square matrix `A`.

Examples:

```
>> A = [1 3; -1 5]
A =
     1     3
    -1     5
>> detA = det(A)
detA =
     8
>> B = [1 3 0; -1 5 2; 1 2 1]
B =
     1     3     0
    -1     5     2
     1     2     1
>> detB = det(B)
detB =
    10
```

9.3 Solutions to Systems of Linear Equations

The following is an example of a system of three linear equations with three unknowns (x_1, x_2, x_3):

$$\begin{array}{rrrcr} 3x_1 & +2x_2 & -x_3 & = & 10 \\ -x_1 & +3x_2 & +2x_3 & = & 5 \\ x_1 & -x_2 & -x_3 & = & -1 \end{array}$$

This system of equations can be rewritten in matrix form as

$$\begin{bmatrix} 3 & 2 & -1 \\ -1 & 3 & 2 \\ 1 & -1 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 10 \\ 5 \\ -1 \end{bmatrix}$$

or

$$\mathbf{Ax} = \mathbf{b}$$

where

$$\mathbf{A} = \begin{bmatrix} 3 & 2 & -1 \\ -1 & 3 & 2 \\ 1 & -1 & -1 \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 10 \\ 5 \\ -1 \end{bmatrix}$$

The system of equations can also be expressed using row vectors for **B** and **x**. For example, the set of equations above can be written as

$$\mathbf{xA} = \mathbf{b}$$

by defining

$$\mathbf{x} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \quad \mathbf{A} = \begin{bmatrix} 3 & -1 & 1 \\ 2 & 3 & -1 \\ -1 & 2 & -1 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 1- & 5 & -1 \end{bmatrix}$$

Note that the matrix **A** in this equation is the transpose of the matrix **A** in the original matrix equation.

The system of equations is nonsingular if the matrix **A** containing the coefficients of the equations is nonsingular. The rank of **A** must be equal to the number of rows of **A** and the determinant $|\mathbf{A}|$ must be nonzero for the system of equations to be nonsingular. To avoid errors, evaluate the rank of **A** (using the `rank` function) to determine that the system is nonsingular before attempting to compute the solution. Two methods for solving a nonsingular system are given below.

Solution by Matrix Inverse

Premultiply the system of equations

$$\mathbf{Ax} = \mathbf{b}$$

by $\mathbf{A}^{-1}$

$$\mathbf{A}^{-1}\mathbf{Ax} = \mathbf{A}^{-1}\mathbf{b}$$

Because $\mathbf{A}^{-1}\mathbf{A}$ is equal to the identity matrix $\mathbf{I}$

$$\mathbf{Ix} = \mathbf{A}^{-1}\mathbf{b}$$

or

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

Example:

```
>> A = [3 2 -1; -1 3 2; 1 -1 -1]
A =
     3     2    -1
    -1     3     2
     1    -1    -1
>> b = [10; 5; -1]
b =
    10
     5
    -1
>> x = inv(A)*b
x =
   -2.0000
    5.0000
   -6.0000
>> A*x
ans =
   10.0000
    5.0000
   -1.0000
```

If the system of equations is expressed in the form

$$\mathbf{xA} = \mathbf{b}$$

where $\mathbf{x}$ and $\mathbf{b}$ are row vectors and $\mathbf{A}$ is a matrix, the equation can be postmultiplied by $\mathbf{A}^{-1}$ to yield

$$\mathbf{x}\mathbf{A}\mathbf{A}^{-1} = \mathbf{b}\mathbf{A}^{-1}$$

Because $\mathbf{A}\mathbf{A}^{-1}$ is equal to the identity matrix $\mathbf{I}$,

$$\mathbf{x}\mathbf{I} = \mathbf{b}\mathbf{A}^{-1}$$

or

$$\mathbf{x} = \mathbf{b}\mathbf{A}^{-1}$$

This is computed in Matlab with the command

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

Solution by Matrix Division

In Matlab, a system of simultaneous equations can be solved by **matrix division**. The solution to the equation

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

can be computed using left division

```
x = A\b;
```

Example:

```
>> A = [3,2,-1; -1,3,2; 1,-1,-1]
A =
     3     2    -1
    -1     3     2
     1    -1    -1
>> b = [10;5;-1]
b =
    10
     5
    -1
>> x = A\b
x =
   -2.0000
    5.0000
   -6.0000
>> check = A*x
check =
   10.0000
    5.0000
   -1.0000
```

This is the preferred method of solving a system of simultaneous linear equations, as the numerical method used, known as Gauss elimination, has better performance than the method of matrix inverse.

If the system of equations is represented by the equation $\mathbf{xA} = \mathbf{b}$, where $\mathbf{x}$ and $\mathbf{b}$ are row vectors, the solution can be computed using matrix right division, as in $\mathbf{x} = \mathbf{b}/\mathbf{A}$.

Geometric Interpretations of Linear Equations in Two Unknowns

1. **Intersection**: The two equations

   $$\begin{aligned} 3x_1 - 4x_2 &= 5 \\ 6x_1 - 10x_2 &= 2 \end{aligned}$$

   can be solved for x_2 to yield

   $$\begin{aligned} x_2 &= (3x_1 - 5)/4 \\ x_2 &= (6x_1 - 2)/10 \end{aligned}$$

   These equations are plotted in Figure 9.4. The linear equations describe straight lines that intersect at $x_1 = 7$, $x_2 = 4$, which is the solution of the two equations. Confirming the solution:

   ```
   >> A = [3 -4; 6 -10];
   >> det(A)
   ans =
       -6
   >> b = [5; 2];
   >> x = A\b
   x =
         7
         4
   ```

   Note that since $|A| \neq 0$, a solution exists.

2. **One Line**: A set of two equations can describe the same line, which can be interpreted to have an infinite number of intersections and thus, no unique solution. The matrix A in this case can be shown to be singular.

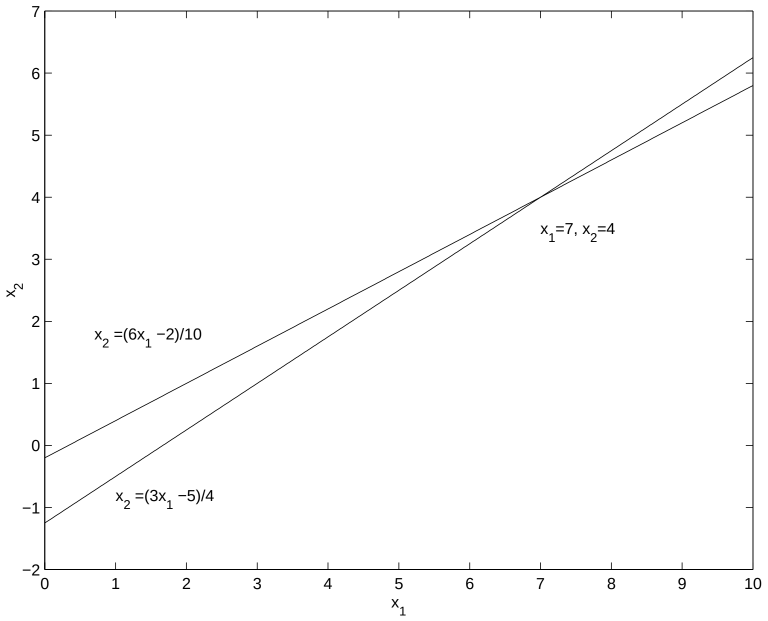


Figure 9.4: The graphs of two equations intersect at the solution

For example the two equations

$$\begin{aligned} 3x_1 - 4x_2 &= 5 \\ 6x_1 - 8x_2 &= 10 \end{aligned}$$

can be solved for x_2, with each yielding

$$x_2 = \frac{3}{4}x_1 - \frac{5}{4}$$

There is no unique solution because the second equation is simply the first equation multiplied by two. The graphs of these two equations are identical. All that can be said is that the solution must satisfy $x_2 = 3(x_1 - 5)/4$, which describes an infinite number of solutions. Confirming this with MATLAB:

```
>> A = [3 -4; 6 -8];
>> det(A)
ans =
     0
>> rank(A)
ans =
     1
```

The determinant is 0 and the rank is less that the dimension of the matrix, so the problem is singular and no unique solution exists.

3. **Parallel Lines**: When the two lines are parallel, there can be no intersection, and thus no solution exists.

   The set of equations

$$\begin{aligned} 3x_1 - 4x_2 &= 5 \\ 6x_1 - 8x_2 &= 3 \end{aligned}$$

   can be solved for x_2, giving

$$\begin{aligned} x_2 &= (3x_1 - 5)/4 \\ x_2 &= (6x_1 - 3)/8 \end{aligned}$$

   Note that slopes of the two lines are both equal to 3/4, so the lines are parallel, as shown in Figure 9.5. Since they do not intersect, no solution exists. The matrix A is the same as that in the one-line singular case above, where this matrix was found to be singular.

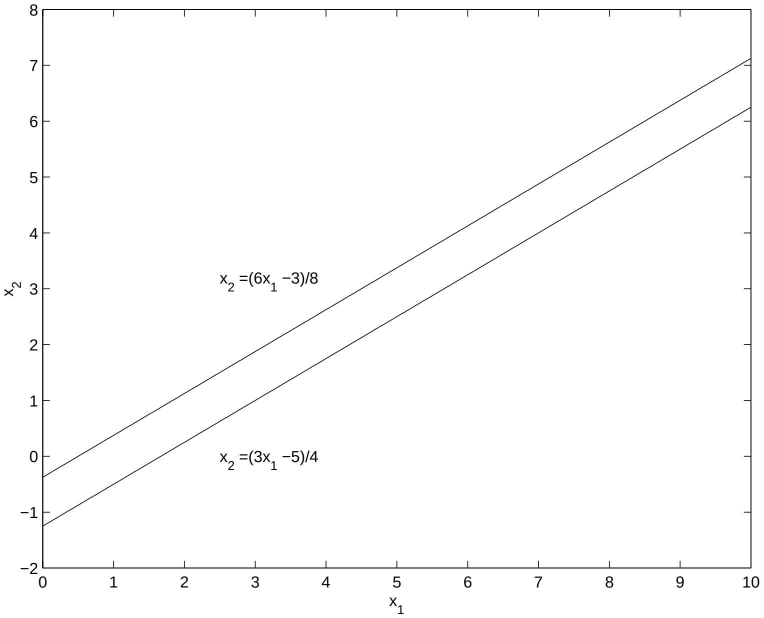


Figure 9.5: Parallel graphs indicate that no solution exists

An **ill-conditioned** set of equations is a set that is close to being singular (for example, two equations whose graphs are close to being parallel). The solution to an ill-conditioned set of equations depends on the accuracy with which the calculations are made. No computer can represent a number with infinitely many significant figures, and so a given set of equations can appear to be singular if the accuracy required to solve them is greater than the number of significant figures used by the software.

9.4 Applied Problem Solving: Robot Motion

Illustrated in Figure 9.6 is a robot arm having two "links" connected by two "joints": a shoulder, or base, joint, and an elbow joint. There is a motor at each joint and the joint angles are θ_1 and θ_2. The (x_1, x_2) coordinates of the hand at the end of the arm are given by

$$x_1 = L_1 \cos\theta_1 + L_2 \cos(\theta_1 + \theta_2)$$

$$x_2 = L_1 \sin\theta_1 + L_2 \sin(\theta_1 + \theta_2)$$

where L_1 and L_2 a

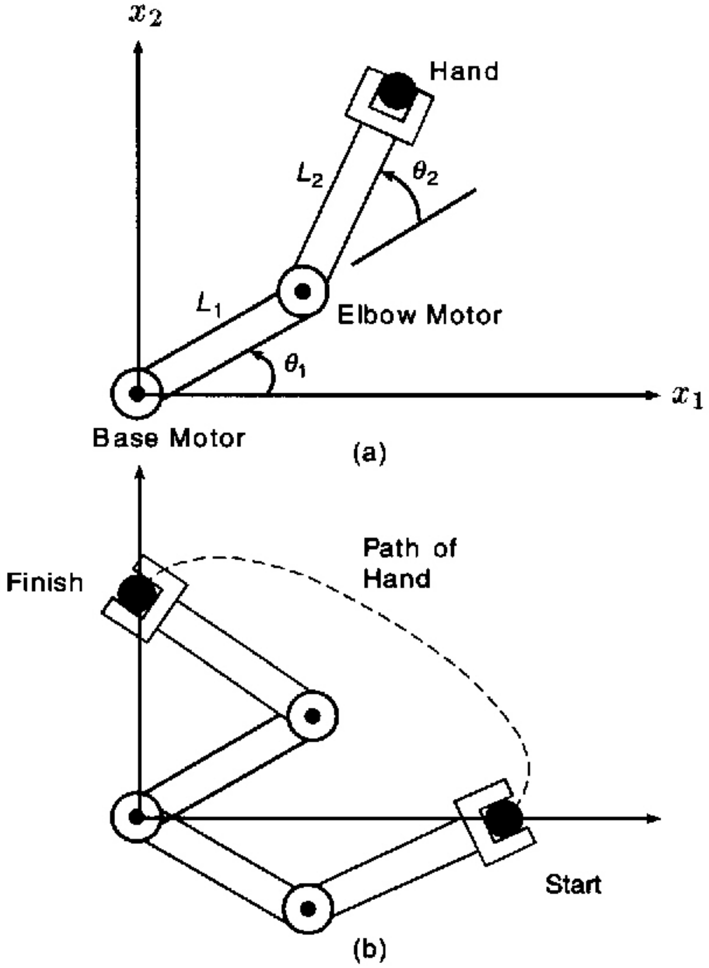


Figure 9.6: Robot arm

The problem is to determine how to control the joint angles by the motors to move the hand from one position to another. The arm is to start from rest at a known position and move to a desired position. It must start and stop with zero velocity and acceleration. The following polynomial expressions are to be used for controlling the motion by generating commands to be sent to the joint motor controllers

$$\theta_1(t) = \theta_1(0) + a_1t^5 + a_2t^4 + a_3t^3 + a_4t^2 + a_5t$$

$$\theta_2(t) = \theta_2(0) + b_1t^5 + b_2t^4 + b_3t^3 + b_4t^2 + b_5t$$

where $\theta_1(0)$ and $\theta_2(0)$ are the initial angles at time $t = 0$ and coefficient vectors $\mathbf{a} = [a_1\ a_2\ a_3\ a_4\ a_5]^T$ and $\mathbf{b} = [b_1\ b_2\ b_3\ b_4\ b_5]^T$ are to be determined to provide the desired motion. The choice of the degree of the polynomials will be explained as the equations of motion are described.

For desired initial coordinates (x_1, x_2) at time $t = 0$ and desired final coordinates at time $t = t_f$, the required values for angles $\theta_1(0)$, $\theta_2(0)$, $\theta_1(t_f)$, and $\theta_2(t_f)$ can be found using trigonometry.

For given values of $\theta_1(0)$, $\theta_1(t_f)$, and t_f, matrix equations are to be set up and solved for coefficient vector $\mathbf{a}$. Similarly, for given values of $\theta_2(0)$, $\theta_2(t_f)$, and t_f, matrix equations are to be set up and solved for coefficient vector $\mathbf{b}$. These results are to be used to plot the path of the hand.

The remaining constraints in the arm motion are that the velocity and acceleration of the links are to be zero at the known starting location and the desired final location. This implies that the angular velocity and angular acceleration of the two angles are to be zero at time $t = 0$ and $t = t_f$. The angular velocity of the first link at time t is the derivative of the angle $\theta_1(t)$ with respect to time

$$\theta_1'(t) = 5a_1t^4 + 4a_2t^3 + 3a_3t^2 + 2a_4t + a_5$$

The velocity at $t = 0$ is

$$\theta_1'(0) = a_5 = 0$$

and thus, coefficient $a_5 = 0$. The angular acceleration is the second derivative of the angle $\theta_1(t)$ with respect to time

$$\theta_1''(t) = 20a_1t^3 + 12a_2t^2 + 6a_3t + 2a_4$$

The acceleration at $t = 0$ is

$$\theta_1''(0) = 2a_4 = 0$$

and thus, coefficient $a_4 = 0$. Writing the three constraints on $\theta_1(t)$ and its derivatives at time $t = t_f$ in matrix form:

$$\begin{bmatrix} t_f^5 & t_f^4 & t_f^3 \\ 5t_f^4 & 4t_f^3 & 3t_f^2 \\ 20t_f^3 & 12t_f^2 & 6t_f \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} \theta_1(t_f) - \theta_1(0) \\ 0 \\ 0 \end{bmatrix}$$

Similarly, for $\theta_2(t)$:

$$\begin{bmatrix} t_f^5 & t_f^4 & t_f^3 \\ 5t_f^4 & 4t_f^3 & 3t_f^2 \\ 20t_f^3 & 12t_f^2 & 6t_f \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} \theta_2(t_f) - \theta_2(0) \\ 0 \\ 0 \end{bmatrix}$$

Note that there are three equations and three unknowns for each angle and its two derivatives. This is the reason for choosing a fifth-degree polynomial to control the motion. The lower degree three terms (t^2, t^1, t^0) have coefficients with value zero to meet the constraints at $t = 0$. This leaves the higher degree three terms (t^5, t^4, t^3) and corresponding three unknown coefficients. Adding additional higher degree terms would require additional constraint equations to be able to compute the values of the corresponding coefficients.

Assuming the following initial and final values and link lengths:

$t_f = 2s$
$\theta_1(0) = -19°$
$\theta_2(0) = 44°$
$\theta_1(t_f) = 43°$
$\theta_2(t_f) = 151°$
$L_1 = 4$ feet
$L_2 = 3$ feet

which correspond to a starting hand location of $(6.5, 0)$ and a destination location of $(0, 2)$, a script to compute the motion control coefficients and to compute and plot the path of the hand is:

```
% Robot arm motion script
%
% Initial values, angles in degrees
tf = 2;
theta10 = -19*pi/180;
theta1tf = 43*pi/180;
theta20 = 44*pi/180;
theta2tf = 151*pi/180;
%
% Equations for a coefficients
T = [   tf^5    tf^4   tf^3
      5*tf^4  4*tf^3 3*tf^2
     20*tf^3 12*tf^2 6*tf   ];
c = [ theta1tf-theta10; 0; 0 ];
disp('Coefficients for theta1 motion:')
a = T\c
%
% Equations for b coefficients
d  = [ theta2tf-theta20; 0; 0 ];
disp('Coefficients for theta2 motion:')
```

```
b  = T\d
%
% Equations of motion
L1 = 4;
L2 = 3;
t = linspace(0,2,401);
tq = [ t.^5;  t.^4; t.^3 ];
theta1 = theta10 + a'*tq;
theta2 = theta20 + b'*tq;
x1 = L1*cos(theta1) + L2*cos(theta1 + theta2);
x2 = L1*sin(theta1) + L2*sin(theta1 + theta2);
%
% Plot path of hand
plot(x1,x2),...
  xlabel('x_1'),...
  ylabel('x_2'),...
  title('Path of robot hand'),...
  text(4.3,0,'t=0s: (x_1,x_2) = (6.5,0)'),...
  text(0.2,2,'t=2s: (x_1,x_2) = (0,2)')
```

Executing the script:

```
>> robot
Coefficients for theta1 motion:
a =
    0.2029
   -1.0145
    1.3526
Coefficients for theta2 motion:
b =
    0.3502
   -1.7508
    2.3344
```

The plot of the hand motion is shown in Figure 9.7.

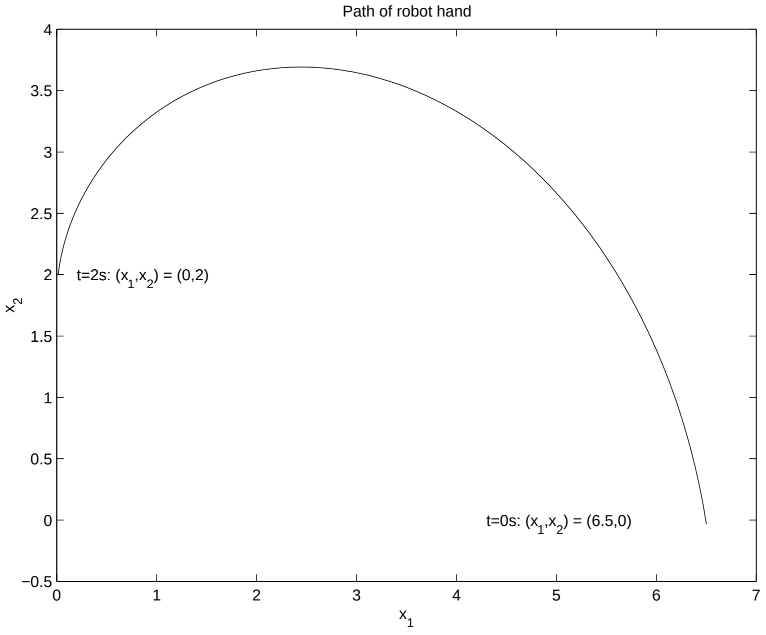


Figure 9.7: Calculated path of robot hand

Section 10

Curve Fitting and Interpolation

Curve fitting: Finding a function (curve) $y = f(x)$ that best "fits" a set of measured x values and corresponding y values.

Interpolating: Estimating the value of y_i corresponding to a chosen x_i having value between the measured x values.

10.1 Minimum Mean-Square Error Curve Fitting

The commonly-used measure of "fit" in curve fitting is **mean-squared error (MSE)**." In minimum mean-squared error (MMSE) curve fitting (also called least-square curve fitting), the parameters of a selected function are chosen to minimize the MSE between the curve and the measured values.

Polynomial regression is a form of MMSE curve fitting in which the selected function is a polynomial and the parameters to be chosen are the polynomial coefficients.

Linear regression is a form of polynomial regression in which the polynomial is of first degree. The two parameters of a linear equation, representing a straight line curve, are chosen to minimize the average of the squared distances between the line and the measured data values.

Linear Regression

To illustrate linear regression, consider the following set of temperature measurements:

Time (s)	Temperature (deg F)
0	0
1	20
2	60
3	68
4	77
5	110

Observing the plot of these data points shown in Figure 10.1, it can be seen that a good estimate of a line passing through the points is $\hat{y} = 20x$. The script used to generate this plot:

```
x = 0:5;
y = [0 20 60 68 77 110];
yhat = 20*x
err = yhat-y
MSE = mean(err.^2)
RMSE = sqrt(MSE)
plot( x,y,’o’, x,yhat),title(’Linear Estimate’),...
  xlabel(’time, s’),ylabel(’Temperature, degrees F’),...
  grid,axis([-1,6,-2,120]), legend(’measured’, ’estimated’,4)
```

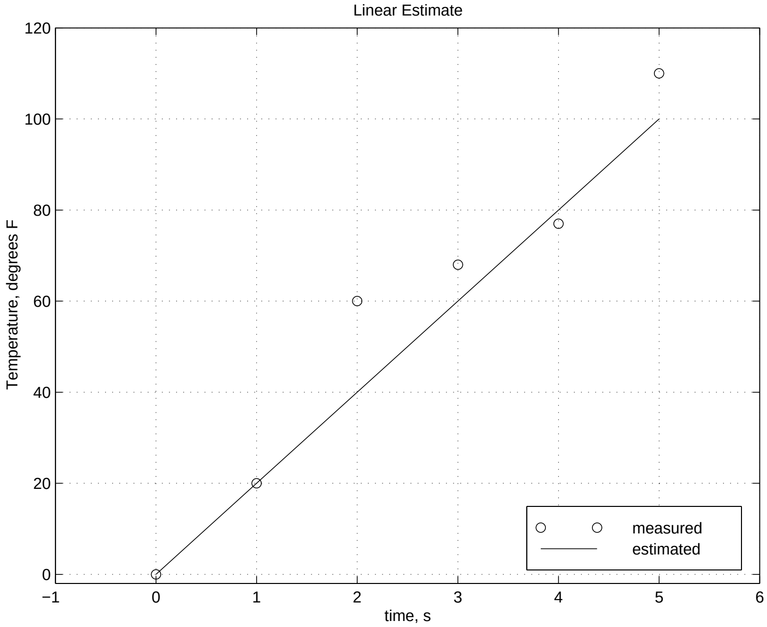


Figure 10.1: A linear estimate

A measure of the quality of the fit of the linear estimate to the data is the mean squared error (MSE) or the root mean squared error (RMSE)

$$\text{MSE} = \frac{1}{N} \sum_{k=1}^{N} (\hat{y}_k - y_k)^2$$

$$\text{RMSE} = \sqrt{\text{MSE}}$$

where N is the number of measured data values (x_k, y_k), with estimated values $\hat{y}_k = 20x_k$. Note that the units of MSE are the square of the units of the measured quantity y_k. The units of RMSE are the same as those of the measured quantity.

For the plotted data

```
>> MSE = mean((yhat-y).^2)
MSE =
   95.5000
>> RMSE = sqrt(MSE)
RMSE =
    9.7724
```

Note that the absolute values of the errors in the estimates range from zero to 20, so an RMSE value of about 10 seems reasonable.

The best fit is the line $\hat{y} = a_1 x + a_2$ having coefficients a_1 and a_2 that produce the smallest mean squared error (MSE). MSE is expressed as

$$\begin{aligned}
\text{MSE} &= \frac{1}{N}\sum_{k=1}^{N}(\hat{y}_k - y_k)^2 \\
&= \frac{1}{N}\sum_{k=1}^{N}(a_1 x_k + a_2 - y_k)^2
\end{aligned}$$

The MSE is minimum when its partial derivatives with respect to each of the coefficients are zero.

$$\begin{aligned}
\frac{\partial \text{MSE}}{\partial a_1} &= \frac{1}{N}\sum_{k=1}^{N} 2(a_1 x_k + a_2 - y_k)x_k \\
&= \frac{2}{N}\sum_{k=1}^{N} a_1 x_k^2 + a_2 x_k - x_k y_k \\
&= \frac{2}{N}\left[\left(\sum_{k=1}^{N} x_k^2\right) a_1 + \left(\sum_{k=1}^{N} x_k\right) a_2 - \left(\sum_{k=1}^{N} x_k y_k\right)\right] \\
&= 0
\end{aligned}$$

$$\begin{aligned}
\frac{\partial \text{MSE}}{\partial a_2} &= \frac{1}{N}\sum_{k=1}^{N} 2(a_1 x_k + a_2 - y_k) \\
&= \frac{2}{N}\left[\left(\sum_{k=1}^{N} x_k\right) a_1 + N a_2 - \left(\sum_{k=1}^{N} y_k\right)\right] \\
&= 0
\end{aligned}$$

Ignoring the constant factors $2/N$ and writing the two equations above in matrix form with the terms in the unknown coefficients a_1 and a_2 on the left hand side

$$\begin{bmatrix} \sum_{k=1}^{N} x_k^2 & \sum_{k=1}^{N} x_k \\ \sum_{k=1}^{N} x_k & N \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} \sum_{k=1}^{N} x_k y_k \\ \sum_{k=1}^{N} y_k \end{bmatrix}$$

This is a pair of linear equations in two unknowns that can be solved using the methods discussed in the previous section of these notes. The values of a_1 and a_2 determined in this way represent the straight line with the minimum mean squared error.

The Matlab command to compute the best linear fit to a set of data is `polyfit(x,y,1)`, which returns a coefficient vector of the straight line fitting the data. For the data above:

```
x = 0:5;
y = [0 20 60 68 77 110];
a = polyfit(x,y,1)
yhat = polyval(a,x);
err = yhat - y
MSE = mean(err.^2)
RMSE = sqrt(MSE)
plot(x,y,’o’,x,yhat),title(’Linear Regression’),...
  xlabel(’time, s’),ylabel(’Temperature, degrees F’),...
  grid,axis([-1,6,-2,120]),legend(’measured’,’estimated’,4)
```

Running this script:

```
a =
   20.8286    3.7619
err =
    3.7619    4.5905  -14.5810   -1.7524   10.0762   -2.0952
MSE =
   59.4698
RMSE =
    7.7117
```

Thus, the best linear fit is

$$\hat{y} = 20.8286x + 3.7619$$

and the root mean squared error is 7.71, lower than that of the previous estimate. The resulting plot is shown in Figure 10.2.

The linear regression curve can also be used to **interpolate** the measured data to estimate values of y corresponding to values of x between the measured values. For example:

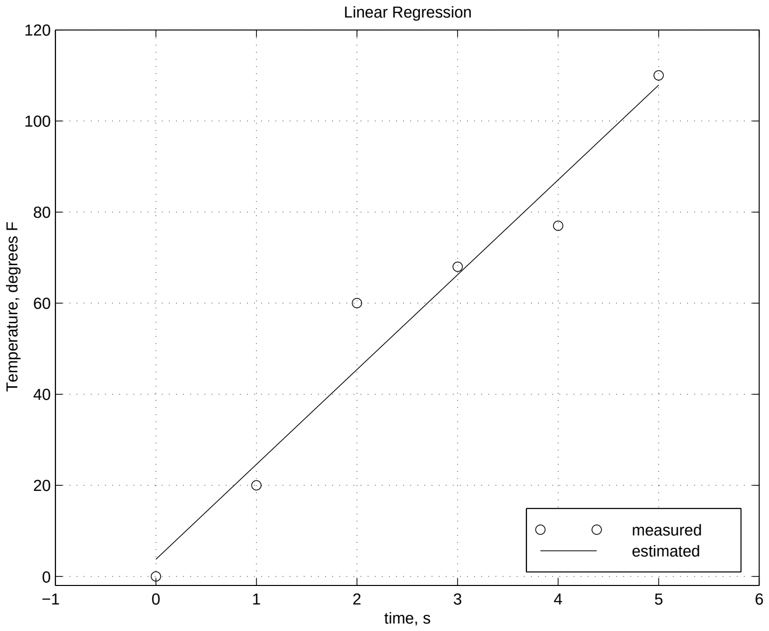


Figure 10.2: Linear regression curve fit

```
>> yi = polyval(a,2.5)
yi =
   55.8333
```

Polynomial Regression

The method discussed above for linear regression can be extended to an n-th degree polynomial curve fit, using a method known as polynomial regression to find the minimum mean squared error values of the polynomial coefficients. The n-th degree polynomial in one variable is

$$f(x) = a_1x^n + a_2x^{n-1} + a_3x^{n-2} + \cdots + a_nx + a_{n+1}$$

The MATLAB command to compute a polynomial regression is

```
a = polyfit(x,y,n}
```

This provides an `n`-th degree least-squares polynomial curve fit to the vectors `x` and `y`, which must be of the same length. It returns a coefficient vector `a` of length `n+1`. As the degree increases, the MSE decreases and the number of points that fall on the curve increases. If a set of $n + 1$ points is used to determine the n-th degree polynomial, all $n + 1$ points will fall on the polynomial curve. Regression analysis cannot determine a unique solution if the number of points is equal to or less than the degree of the polynomial model.

Consider a set of 11 data points and find the best 2-nd and 10-th degree polynomial curve fits to this data, using the following script:

```
x = 0:0.1:1;
y = [-0.447 1.978 3.28 6.16 7.08 7.34 7.66 9.56 9.48 9.30 11.2];
a2 = polyfit(x,y,2);
disp('a2 coefficients:')
disp(a2')
a10 = polyfit(x,y,10);
disp('a10 coefficients:')
format short e
disp(a10')
xi = linspace(0,1,101);
yi2 = polyval(a2,xi);
yi10 = polyval(a10,xi);
plot(x,y,'o',xi,yi2,'--',xi,yi10),...
  xlabel('x'), ylabel('y'),...
  title('2nd and 10th Degree Polynomial Curve Fit'),...
  legend('Measured','2nd Degree','10th Degree',4)
```

Executing this script:

```
a2 coefficients:
   -9.8108
   20.1293
   -0.0317
a10 coefficients:
 -4.6436e+005
  2.2965e+006
 -4.8773e+006
  5.8233e+006
 -4.2948e+006
  2.0211e+006
 -6.0322e+005
  1.0896e+005
 -1.0626e+004
  4.3599e+002
 -4.4700e-001
```

The plot produced is shown in Figure 10.3.

Thus, the best quadratic ($n = 2$) curve fit is

$$\hat{y} = -9.8108x^2 + 20.1293x - 0.0317$$

As the polynomial degree increases, the approximation becomes less smooth since higher-order polynomials can be differentiated more times before then become zero. Note the values of the 10-th degree polynomial coefficients `a10` compared to those of the quadratic fit. Note also the seven orders of magnitude difference between the smallest (`-4.4700e-001`) and the largest (`5.8233e+006`)

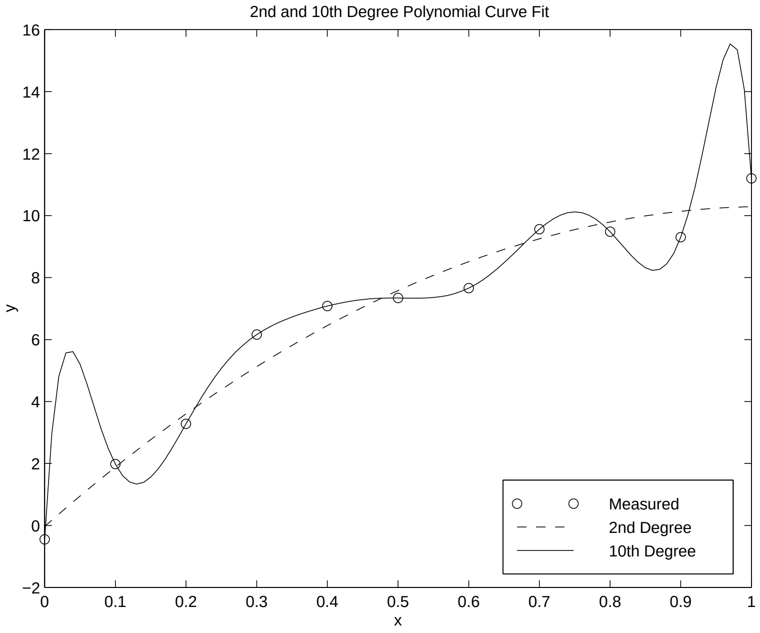


Figure 10.3: Polynomial Curve Fitting

coefficients and the alternating signs on the coefficients. This example clearly demonstrates the difficulties with higher-degree polynomials.

10.2 Applied Problem Solving: Hydraulic Engineering

The following application is from *Matlab for Engineering Applications* by William J. Palm III (McGraw-Hill, 1999).

Torricelli's principle of hydraulic resistance states that the volume flow rate f of a liquid through a restriction, such as an opening or a valve, is proportional to the square root of the pressure drop p across the restriction

$$f = c\sqrt{p}$$

where c is a constant. In many applications the weight of liquid in a tank above the valve causes the pressure drop. In this case, the flow rate is proportional to the square root of the the volume V in the tank

$$f = r\sqrt{V}$$

where r is a constant. This expression can be generalized to

$$f = rV^e$$

and an attempt can be made to experimentally verify that $e = 0.5$.

Consider applying this principle to the flow of coffee out of a coffee pot. To gather some experimental measurements, place a 15-cup coffee pot under a water faucet and fill to the 15-cup line. With the outlet valve open, adjust the flow rate through the faucet so that the water level remains constant at 15 cups, and measure the time for one cup to flow out of the pot. Repeat this measurement with the pot filled to the levels shown in the following table:

Volume V (cups)	Fill time t (s)
15	6
12	7
9	8
6	9

Use this data to verify Torricelli's principle for the coffee pot and obtain a relation between the flow rate and the number of cups in the pot.

Torricelli's principle is a power function and if we apply the base 10 logarithm to both sides, we obtain

$$\log_{10} f = e \log_{10} V + \log_{10} r$$

This equation has the form

$$y = a_1 x + a_2$$

where $y = \log_{10} f$, $x = \log_{10} V$, $a_1 = e$, and $a_2 = \log_{10} r$. Thus, if we plot $\log_{10} f$ versus $\log_{10} V$, we should obtain a straight line. The values for f are obtained from the reciprocals of the given data for t. That is, $f = 1/t$ cups per second. We can find the power function that fits the data with the command

```
a = polyfit(log10(V), log10(f), 1)
```

The first element a_1 of vector `a` will be e and the second element a_2 will be $\log_{10} r$. We can find r from $r = 10^{a_2}$.

The MATLAB script is:

```
% Modeling of hydraulic resistance in a coffee pot
%
% Measured data
vol = [6, 9, 12, 15];                    % coffee pot volume (cups)
time = [9, 8, 7, 6];                     % time to fill one cup (seconds)
flow = 1./time;                          % flow rate (cups/seconds)
%
% Fit a straight line to log transformed data
a = polyfit(log10(vol),log10(flow),1);
```

```
disp('Exponent e')
e = a(1)
disp('Constant r')
r = 10^a(2)
%
% Plot the data and the fitted curve on loglog and linear plots
x = [6:0.01:40];
y = r*x.^e;
subplot(2,1,1),loglog(x,y,vol,flow,'o'),grid,...
   xlabel('Volume (cups - log scale)'),...
   ylabel('Flow Rate (cups/sec - log scale)'),...
   axis([5 15 .1 .3]), legend('Model','Experimental')
subplot(2,1,2),plot(x,y,vol,flow,'o'),grid,xlabel('Volume (cups)'),...
   ylabel('Flow Rate (cups/sec)'),axis([5 15 .1 .3]),...
   legend('Model','Experimental')
```

The displayed output is:

```
Exponent e
e =
    0.4331
Constant r
r =
    0.0499
```

The derived relation is

$$f = 0.0499V^{0.433}$$

Because the exponent is 0.433, not 0.5, our model does not agree exactly with Torricelli's principle, but it is close. Note that the plot in Figure 10.4 shows that the data points do not lie exactly on the fitted straight line. In this application it is difficult to measure the time to fill one cup with an accuracy greater than an integer second, so this inaccuracy could have caused our result to disagree with that predicted by Torricelli.

10.3 Interpolation

In **interpolation**, data points $(x(k), y(k))$ for $k = 1,\ 2, \ldots, N$ have been acquired, with the x values in ascending order so that $x(k) < x(k+1)$. For a chosen value, an estimate is sought for y_i corresponding to a chosen x_i value that lies within the data range $(x(1) \leq x_i \leq x(N))$. Thus, the objective is not to fit the entire range of data with a single curve, as the case for curve fitting, but rather to use the data to estimate a y_i value corresponding to x_i. The estimate is "local" in the sense that it is oomputed only from data points having $x(k)$ values near that of x_i.

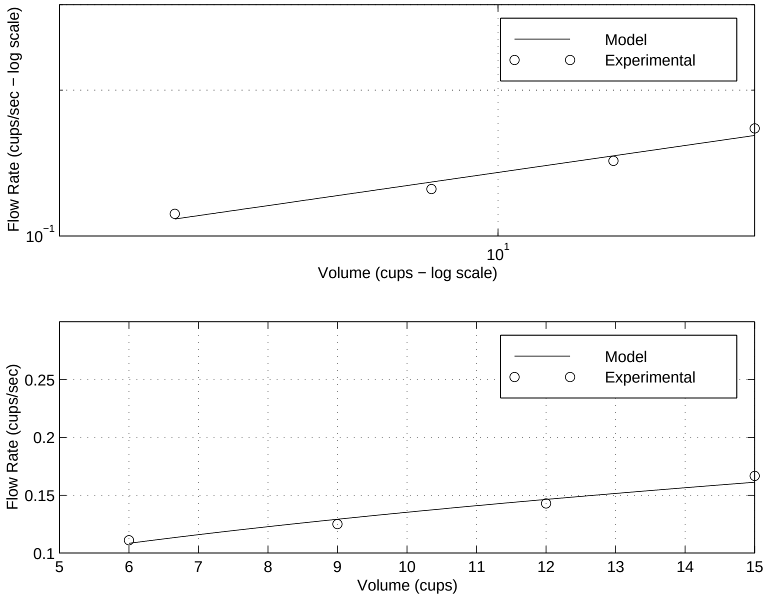


Figure 10.4: Flow rate and volume for a coffee pot

In **linear interpolation**, the estimate is based on a straight line connecting neighboring pairs of data points. In **cubic-spline interpolation**, the estimate is based on joining neighboring data points with a cubic (third-degree) polynomial.

Linear Interpolation

The method of linear interpolation is illustrated in Figure 10.5, which shows measured data points $(x(k), y(k))$ and $(x(k+1), y(k+1))$ for which the chosen x_i lies in the range $x(k) \leq x_i \leq x(k+1)$. The equation of the straight line between these data points is

$$y_i = y(k) + m(x_i - x(k))$$

where m is the slope of the line

$$m = \frac{y(k+1) - y(k)}{x(k+1) - x(k)}$$

Substituting this slope into the equation of the line, the estimated or interpolated value is

$$y_i = y(k) + \frac{y(k+1) - y(k)}{x(k+1) - x(k)}(x_i - x(k))$$

Given a set of data points, it is conceptually straightforward to interpolate for a new point between two of the given points. However, the interpolation takes several steps, because it is first necessary

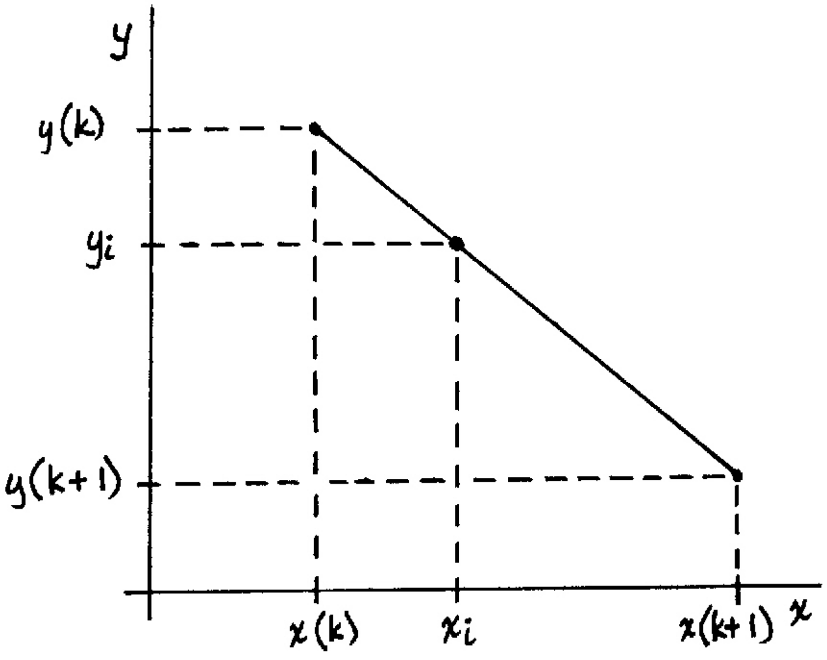


Figure 10.5: Linear interpolation

to find the two values in the data between which the desired point falls. When these two values are found, the interpolation equation above can be applied. These steps are performed by the MATLAB interpolation function `interp1`.

`yi = interp1(x,y,xi)` Returns vector `yi` of the length of `xi`, containing the interpolated `y` values corresponding to `xi` using linear interpolation. The vector `x`, which must have values in ascending order, specifies the points at which the data `y` is given. Vectors `x` and `y` must be of the same length. The values of `xi` must be within the range of the `x` values.

To illustrate use of linear interpolation, consider again the temperature data from the previous section. The range of x in this data is 0.0 to 5.0, so interpolated values can be found for data in this range. Computing linearly interpolated values for $x = 2.6$ and $x = 4.9$:

```
>> x = 0:5;
>> y = [0 20 60 68 77 110];
>> yi = interp1(x,y,[2.6 4.9])
yi =
   64.8000  106.7000
```

If the second argument of the `interp1` function is a matrix, the function returns a row vector with the same number of columns, and each value returned will be interpolated from its corresponding column of data. Assume the following data has been acquired:

Time, s	Temp 1	Temp 2	Temp 3
0	0	0	0
1	20	25	52
2	60	62	90
3	68	67	91
4	77	82	93
5	110	103	96

Storing this data in a matrix and interpolating at the time 2.6 seconds:

```
>> x = (0:5)';
>> y(:,1) = [0,20,60,68,77,110]';
>> y(:,2) = [0,25,62,67,82,103]';
>> y(:,3) = [0,52,90,91,93,96]';
>> temps = interp1(x,y,2.6)
temps =
   64.8000   65.0000   90.6000
```

Cubic-Spline Interpolation

A **cubic spline** is a third-degree polynomial computed to provide a smooth curve between the two data points bounding the point for which an interpolated value is to be determine and to provide a smooth transition from the third-degree polynomial between the previous two points.

The interpolating equation is

$$y_i = a_1(x_i - x(k))^3 + a_2(x_i - x(k))^2 + a_3(x_i - x(k)) + a_4$$

for which $x(k) \leq x_i \leq x(k+1)$. The coefficients $a_1,\ a_2,\ a_3$ and a_4 are determined so that the following three conditions are met:

1. The polynomial must pass through the data points at its end points $x(k)$ and $x(k+1)$. For $x_i = x(k)$, this requires that $y_i = y(k)$, so that $a_4 = y(k)$.
2. The slopes (first derivatives) of cubic polynomials in adjacent data intervals must be equal at their common data point.
3. The curvatures of adjacent polynomials must be equal at their common data point.

The corresponding Matlab function is:

`yi = spline(x,y,xi)` Returns vector `yi` of the length of `xi`, containing the interpolated `y` values corresponding to `xi` using cubic-spline interpolation. The vector `x`, which must have values in ascending order, specifies the points at which the data `y` is given. Vectors `x` and `y` must be of the same length. The values of `xi` must be within the range of the `x` values.

To illustrate, apply cubic-spline interpolation to the temperature data considered above.

```
>> x = 0:5;
>> y = [0,20,60,68,77,110];
>> temp = spline(x,y,[2.6,4.9])
temp =
   67.3013  105.2020
```

To compute and plot linear and cubic-spline interpolation curves over a range of values, an `xi` vector with the desired resolution of the curve can be generated and used as the third parameter in the `interp1` function. For example:

```
% Comparison of linear and cubic spline interpolation
x = (0:5);
y = [0,20,60,68,77,110];
xi = 0:0.1:5;
ylin = interp1(x,y,xi);
ycub = spline(x,y,xi);
plot(xi,ylin,':',xi,ycub,x,y,'o'),...
  legend('Linear','Cubic','Measured',4),...
  title('Linear and cubic spline interpolation'),...
  xlabel('x'),...
  axis([-1,6,-20,120]),...
  grid
```

The resulting plot is shown in Figure 10.6.

10.4 Applied Problem Solving: Human Hearing

To illustrate one-dimensional interpolation, consider the following example: The threshold of audibility (i.e. the lowest perceptible sound level) of the human ear varies with frequency. Plotting frequency in Hz on a log scale against sound pressure level in dB, normalized so that 0 dB appears at 1000 Hz is done with the following script:

```
f = [20:10:100 200:100:1000 1500 2000:1000:10000]; % frequency in Hz
spl = [76 66 59 54 49 46 43 40 38 22 ... % sound pressure level in dB
       14 9 6 3.5 2.5 1.4 0.7 0 -1 -3 ...
       -8 -7 -2 2 7 9 11 12];
semilogx(f,spl,'-o'),...
  xlabel('Frequency, Hz'),...
  ylabel('Relative Sound Pressure Level, dB'),...
  title('Threshold of human hearing'),grid on
```

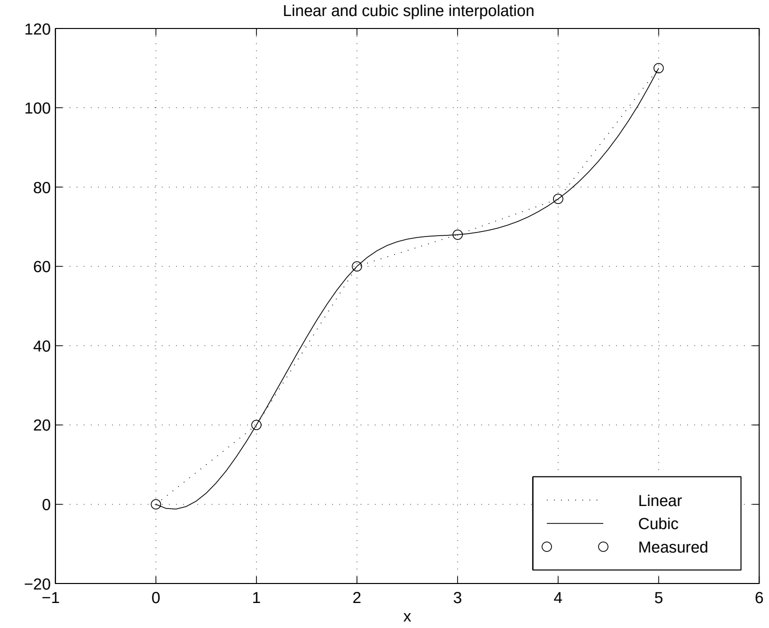


Figure 10.6: Comparison of linear and cubic spline interpolation

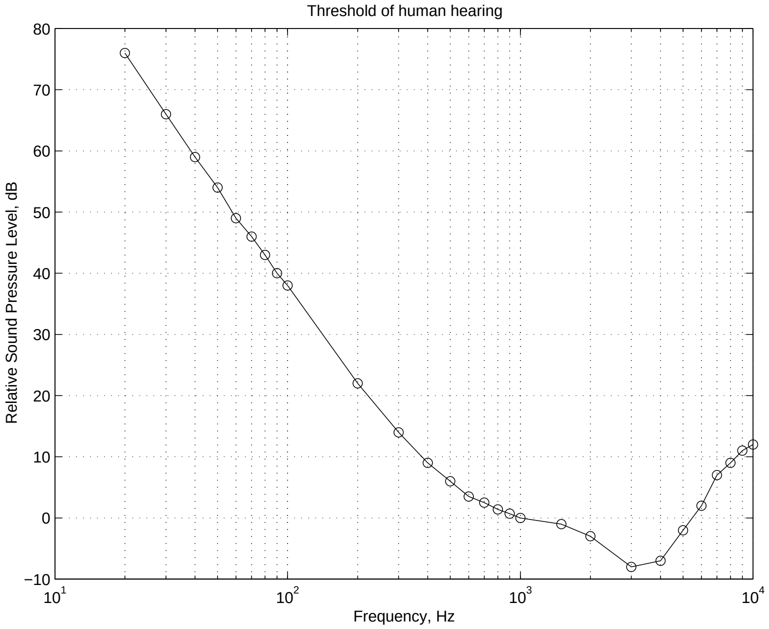


Figure 10.7: Plot of Threshold of Human Hearing

The resulting plot is shown in Figure 10.7, where by default, the data points have been connected with straight lines.

Based on this plot, the human ear is most sensitive to tones around 3 kHz. The function `interp1` can be used to estimate the sound pressure level in several different ways at a frequency of 2.5 kHz.

```
>> slin = interp1(f,spl,2500,'linear')  % linear interpolation
slin =
   -5.5000
>> scub = interp1(f,spl,2500,'spline')  % cubic spline interpolation
scub =
   -5.6641
>> snn = interp1(f,spl,2500,'nearest')  % nearest neighbor interpolation
snn =
    -8
```

where **nearest neighbor** is another interpolation method, in which, as implied by its name, interpolates with the nearest sample in the original data. Note the differences in these results. The first result returns exactly what is shown in the figure at 2.5 kHz since MATLAB linearly interpolates between data points on plots. Cubic spline interpolation fits a cubic polynomial to each data interval, so it returns a slightly different result. The poorest interpolation in this case is the nearest neighbor.

How is an interpolation method to be chosen for a given problem? In many cases, linear interpolation is sufficient, which is why it is the default method. While nearest neighbor produced poor results here, it is often used when speed is important or the data set is large. The most time-consuming method is spline, but it often produces the most desirable results.

Now use cubic interpolation to investigate the data at a finer interval near the minimum.

```
fi = linspace(2500,5000);
spli = interp1(f,spl,fi,'cubic');    % interpolate near minumum
k = find(f>=2000 & f<=5000);          % find indices near minumum
semilogx(f(k),spl(k),'--o',fi,spli),...  % plot orig & cubic data
  legend('Linear','Cubic'),...
  xlabel('Frequency, Hz'),...
  ylabel('Relative Sound Pressure Level, dB'),...
  title('Threshold of human hearing'),grid on
```

The resulting plot is shown in Figure 10.8. By specifying a finer resolution on the frequency axis and using cubic convolution, a smoother estimate of the sound pressure level is generated. Note how the slope of the cubic solution does not change abruptly at the data points.

With the cubic spline interpolation, a better estimate can be made of the frequency of greatest sensitivity.

```
>> [splmin,kmin] = min(spli)      % minimum and index of minimum
```

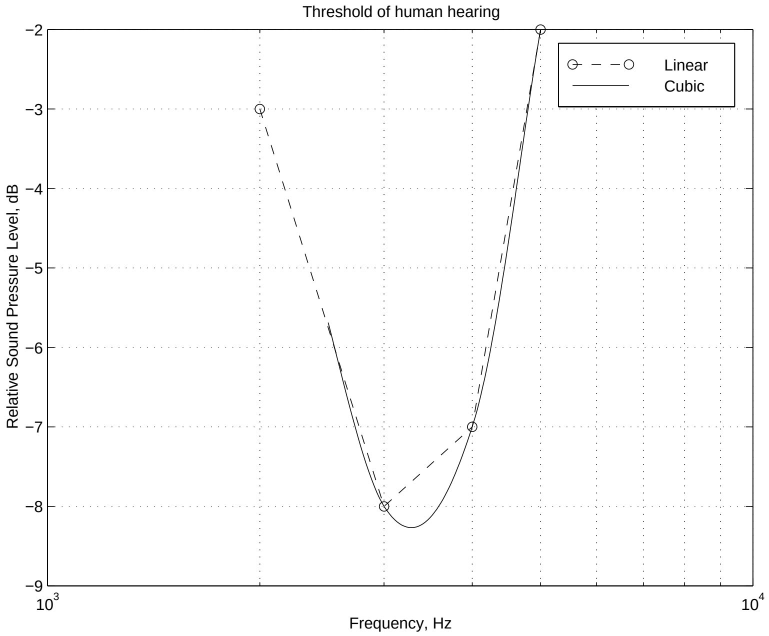


Figure 10.8: Plot of threshold of human hearing near minimum

```
splmin =
   -8.2683
kmin =
    32
>> fmin = fi(kmin)      % frequency at minimum
fmin =
  3.2828e+003
```

Thus, by this analysis, the human ear is most sensitive to tones near 3.3 kHz.

Section 11

Integration and Differentiation

Integration and differentiation are the key concepts presented in the first two calculus courses and they are fundamental to solving many engineering and science problems. While many of these problems can be solved analytically, there are also many problems that require numerical integration or numerical differentiation techniques.

11.1 Numerical Integration

The **integral** of a function $f(x)$ for $a \leq x \leq b$ can be interpreted as the area under the curve of $f(x)$ between $x = a$ and $x = b$, as shown in Figure 11.1. Denoting this area as A, the integral is written as

$$A = \int_a^b f(x) dx$$

Definitions:

- Integrand: $f(x)$
- Lower limit of integration: a
- Upper limit of integration: b
- Variable of integration: x

Numerical integration, called **quadrature**, involves methods for estimating the value of a definite integral. In these methods, the function $f(x)$ is estimated or approximated by another function $\hat{f}(x)$, chosen so that the area under $\hat{f}(x)$ is easy to compute. The better the estimate of $f(x)$ by $\hat{f}(x)$, the better the estimate of the integral of $f(x)$. Two of the most common numerical integration techniques estimate $f(x)$ with a set of piecewise linear functions or with a set of piecewise parabolic functions. If the function is estimated with piecewise linear functions, the area of the trapezoids that compose the area under the piecewise linear functions is the approximation to the desired

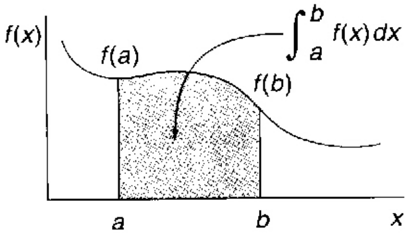


Figure 11.1: Integral of $f(x)$ from a to b

integral, and the method is known as the **trapezoidal rule**. If the function is estimated with piecewise quadratic functions, the technique is called **Simpson's rule**.

Trapezoidal Rule

In the trapezoidal rule for integration, the interval $[a, b]$ is divided into n equal subintervals and the curve $f(x)$ is approximated by $\hat{f}(x)$ on each subinterval as a straight line connecting the values of $f(x)$ at the ends of each subinterval. The integral A is the sum of the approximate integrals on each subinterval. The width of each subinterval is

$$\Delta x = \frac{b-a}{n}$$

The range of values of x on subinterval i is

$$[x_i, x_{i+1}] = [x_i, x_i + \Delta x] = [a + i\Delta x, a + (i+1)\Delta x], \quad i = 0, \ldots, n-1$$

The approximation of $f(x)$ on subinterval i is shown in Figure 11.2. The approximating curve $\hat{f}(x)$ is represented by the dashed line. The approximate area A_i of $f(x)$ over this subinterval is that of the trapezoid under $\hat{f}(x)$

$$A_i \approx (x_{i+1} - x_i)\,\frac{f(x_i) + f(x_{i+1})}{2} = \frac{\Delta x}{2}\,(f(x_i) + f(x_{i+1}))$$

The full integral A is then approximated by

$$\begin{aligned} A_T &= \sum_{i=0}^{n} \frac{\Delta x}{2}\,(f(x_i) + f(x_{i+1})) \\ &= \frac{\Delta x}{2}\,(f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-1}) + f(x_n)) \end{aligned}$$

The MATLAB function for trapezoidal rule integration is

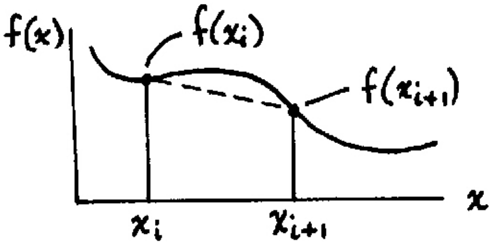


Figure 11.2: Approximation of $f(x)$ from x_i to x_{i+1}

`z = trapz(x,y)` Integral of `y` with respect to `x` by the trapezoidal rule. `x` and `y` must be vectors of the same length, or `x` must be a column vector and `y` an array whose first non-singleton dimension is `length(x)`. `trapz` operates along this dimension.

`z = trapz(x,y,dim)` Integrates across dimension `dim` of `y`. The length of `x` must be the same as `size(y,dim))`.

To illustrate integration, consider the function `humps(x)`, a demonstration function provided by MATLAB that has strong peaks near $x = 0.3$ and $x = 0.9$, shown in Figure 11.3.

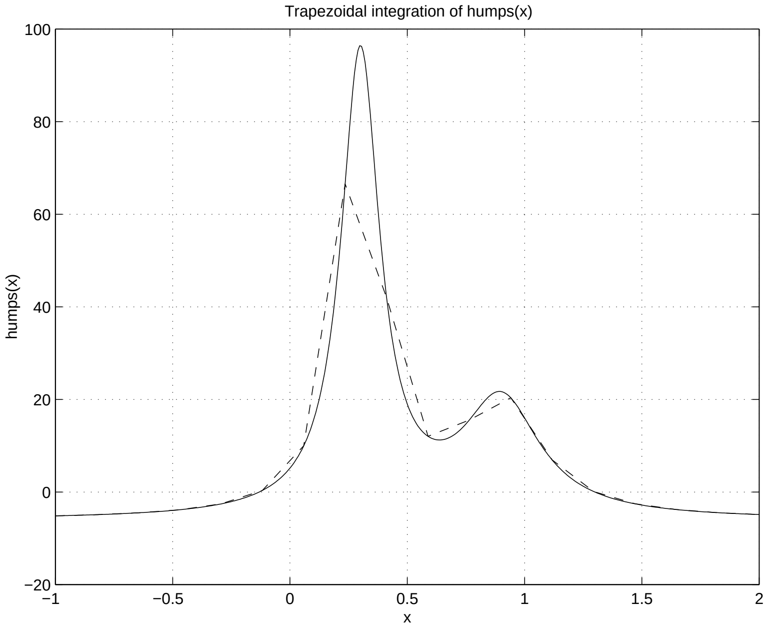


Figure 11.3: Trapezoidal integration of humps(x)

Using computed samples of the `humps` function, the function `trapz` approximates the area using the trapezoidal approximation. For $n = 18$ subintervals, the integral approximation is given by:

```
>> x = linspace(-1,2,18)
>> y = humps(x);
>> area = trapz(x,y)
area =
   25.1406
```

Based on the figure, this is probably not a very accurate estimate of the area. However, if a finer discretization is used by increasing the number of subintervals to 401, better accuracy is achieved:

```
>> x = linspace(-1,2,401);
>> y = humps(x);
>> area = trapz(x,y)
area =
   26.3449
```

This area agrees with the analytic integral to five significant digits.

Another integral of interest is that for which the upper limit is the variable x:

$$\int_a^x f(u)du$$

where the variable of integration has been changed to u to avoid confusion with the upper limit x. The definite integral from the lower limit of integration, a, to any point x is found by evaluating the integral at x. Using the trapezoidal rule, tabulated values of the cumulative integral are computed using the function `cumtrapz`:

`z = cumtrapz(x,y)`	Computes the cumulative integral of `y` with respect to `x` using trapezoidal integration. `x` and `y` must be vectors of the same length, or `x` must be a column vector and `y` an array whose first non-singleton dimension is `length(x)`. `cumtrapz` operates across this dimension.
`z = cumtrapz(x,y,dim)`	Integrates along dimension `dim` of `y`. The length of `x` must be the same as `size(y,dim))`.

For example, consider integration of `humps(x)`

```
x = linspace(-1,2,201);
y = humps(x);
z = cumtrapz(x,y);
  plot(x,y,x,z,'--'),...
  title('Cumulative integral of humps(x)'),...
  xlabel('x'),ylabel('humps(x) and integral of humps(x)'),...
  grid,legend('humps(x)','integral of humps(x)')
```

The resulting plot is shown in Figure 11.4.

Example 11.1 *Velocity from an accelerometer*

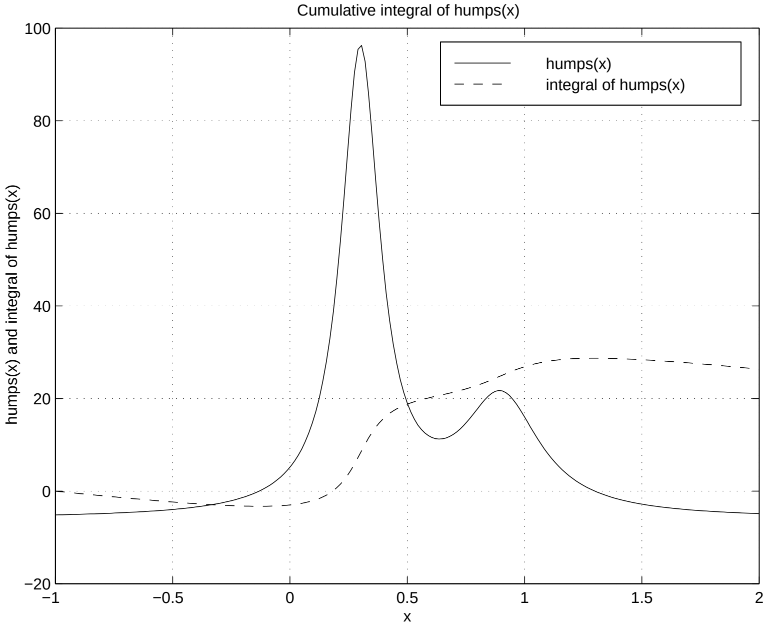


Figure 11.4: Cumulative integral of humps(x)

An *accelerometer* measures acceleration and the resulting measurements can be used to estimate velocity and displacement. The acceleration is integrated to estimate velocity and the velocity in turn is integrated to estimate displacement. Suppose a vehicle starts from rest at time $t = 0$ and that its measured acceleration is given by the following table:

Time (s)	Accel (m/s^2)
0	0
1	2
2	4
3	7
4	11
5	17
6	24
7	32
8	41
9	48
10	51

A script to estimate the velocity:

```
t = [0:10];
a = [0,2,4,7,11,17,24,32,41,48,51];
v = cumtrapz(t,a);
disp(['     time     accel  velocity'])
disp([t',a',v'])
```

The displayed results:

```
     time     accel  velocity
        0         0         0
   1.0000    2.0000    1.0000
   2.0000    4.0000    4.0000
   3.0000    7.0000    9.5000
   4.0000   11.0000   18.5000
   5.0000   17.0000   32.5000
   6.0000   24.0000   53.0000
   7.0000   32.0000   81.0000
   8.0000   41.0000  117.5000
   9.0000   48.0000  162.0000
  10.0000   51.0000  211.5000
```

■

Simpson's Rule

If the area under a curve is approximated by areas under quadratic curve, with the interval $[a, b]$ divided into $2n$ equal subintervals, then the area can be approximated by Simpson's rule:

$$A_S = \frac{\Delta x}{3}\left(f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \cdots + 2f(x_{2n-2}) + 4f(x_{2n-1}) + f(x_{2n})\right)$$

where the x_k values represent the endpoints of the subintervals and where $x_0 = a$, $x_{2n} = b$, and $\Delta x = (b - a)/(2n)$.

If the piecewise components of the approximating function are higher-degree functions, the integration techniques are referred to as Newton-Cotes integration techniques. The estimate of an integral improves as more components are used to approximate the area under a curve. An integral approximation can be made to meet a desired relative error by increasing the number of components until the error specification is met.

If the function to be integrated contains a **singularity** (a point at which the function or its derivatives are infinity or are not defined), a satisfactory result may not be provided by numerical techniques.

Matlab provides two quadrature functions for performing numerical function integration.

`quad('function',a,b)` Returns the approximation of the integral of the function `f(x)`, whose name is contained in the string `'function'`, from `a` to `b` to within a relative error of 0.001 using an adaptive recursive Simpson's rule. Function `f` must return a vector of output values if given a vector of input values. `Inf` is returned if an excessive recursion level is reached, indicating a possibly singular integral.

`quad8('function',a,b)` Returns the approximation of the integral of the function between `a` and `b` using an adaptive Newton-Cotes 8-panel rule. This function is better than the `quad` function at handling some functions with singularities.

For example, consider computing the integral of the `humps` function again:

```
>> quad('humps',-1,2)
ans =
  26.34497558341242
>> quad8('humps',-1,2)
ans =
  26.34496024631924
```

Here the `quad` solution achieves six significant digit accuracy and the `quad8` solution achieves eight significant digit accuracy.

Two-Dimensional Integration

Two-dimensional integration is evaluated using the function `dblquad`, which approximates the integral

$$\int_{y_{min}}^{y_{max}} \int_{x_{min}}^{x_{max}} f(x,y)dxdy$$

`q = dblquad('function',xmin,xmax,ymin,ymax)`: Returns the evaluation of the double integral `f(x,y)` using the `quad` quadrature function where `x` is the inner variable ranging from `xmin` to `xmax`, and `y` is the outer variable ranging from `ymin` to `ymax`. The first argument `'function'` is a string representing the integrand function. This function must be a function of two variables of the form `z =f(x,y)`. The function must take a vector `x` and a scalar `y` and return a vector `z` that is the function evaluated at `y` and each value of `x`.

To use `dblquad`, one must first create a function that evaluates $f(x,y)$. Consider, for example, the function `func.m`:

```
function z = func(x,y)
% FUNC(X,Y) computes an example function of two variables
z = sin(x).*cos(y) + 1;
```

Using the commands listed below, this function can be plotted as shown in Figure 11.5:

```
x = linspace(0,pi,20);
y = linspace(-pi,pi,20);
[xx,yy] = meshgrid(x,y);
zz = func(xx,yy);
mesh(xx,yy,zz),xlabel('x'),ylabel('y')
```

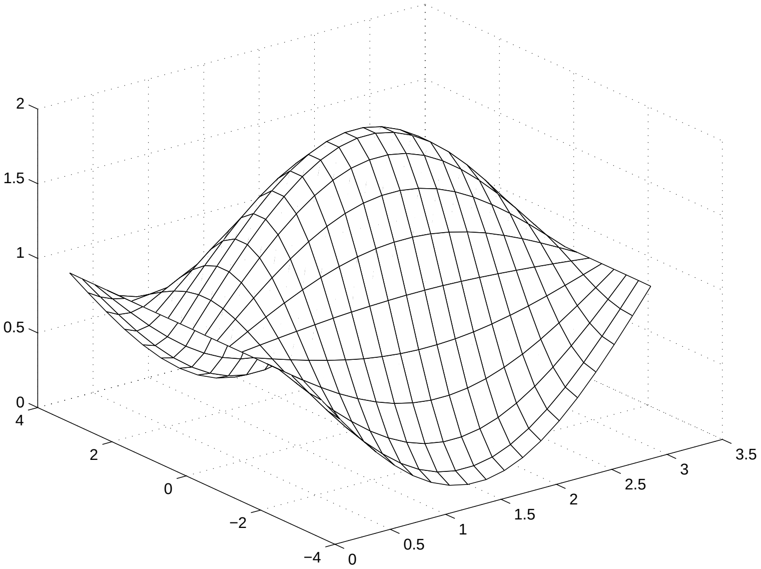


Figure 11.5: Mesh plot of `func(x,y)`

The integral of this function over limits $0 \leq x \leq \pi$, $-\pi \leq y \leq \pi$ is given by

```
>> dblquad('func',0,pi,-pi,pi)
ans =
  19.73921476256606
```

11.2 Numerical Differentiation

The **derivative** of a function $f(x)$ is defined as a function $f'(x)$ that is equal to the rate of change of $f(x)$ with respect to x

$$f'(x) = \frac{df(x)}{dx}$$

Differentiation and integration are complementary operations. For example, if

$$f(x) = \int g(x)dx$$

then $g(x)$ is the derivative of $f(x)$ with respect to x. This relationship is written as

$$g(x) = \frac{df(x)}{dx} = f'(x)$$

The derivative $f'(x)$ can be interpreted geometrically as the slope of the function at $(x, f(x))$, where the slope of $f(x)$ is the slope of the tangent line to the function at x, as shown in Figure 11.6.

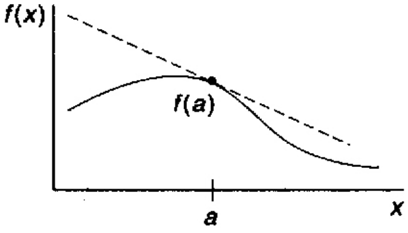


Figure 11.6: Derivative of $f(x)$ at $x = a$.

Formally, the derivative of f at x is

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{(x + \Delta x) - x} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

Difference Expressions

Numerical differentiation techniques estimate the derivative of a function at a point x_k by approximating the slope of the tangent line at x_k using values of the function at points near x_k. The approximation of the slope of the tangent line can be done in several ways, as shown in Figure 11.7.

The **backward difference** approximation of the derivative at x_k is the slope of the line between $f(x_{k-1})$ and $f(x_k)$, as shown in Figure 11.7a

$$f'(x_k) = \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}$$

If the samples of x are uniformly spaced so that $x_k - x_{k-1} = \Delta x$

$$f'(x_k) = \frac{f(x_k) - f(x_{k-1})}{\Delta x}$$

The **forward difference** approximation of the derivative at x_k is the slope of the line between $f(x_k)$ and $f(x_{k+1})$, as shown in Figure 11.7b

$$f'(x_k) = \frac{f(x_{k+1}) - f(x_k)}{x_{k+1} - x_k}$$

For uniform spacing Δx

$$f'(x_k) = \frac{f(x_{k+1}) - f(x_k)}{\Delta x}$$

Note that the forward difference at x_k is the same as the backward difference at x_{k+1}.

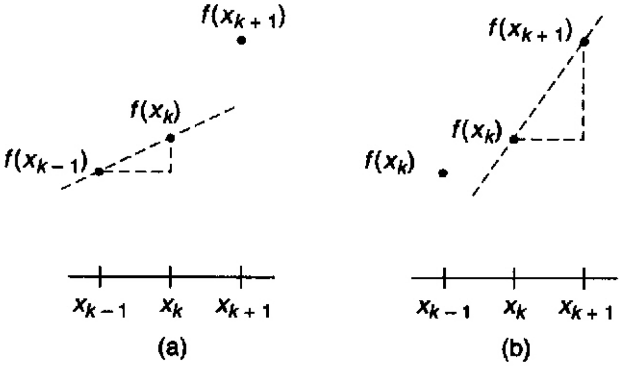


Figure 11.7: Techniques for approximating $f'(x_k)$.

The quality of both of these derivative approximations depends heavily on two factors: the spacing of the samples and the scatter in the data due to measurement error. The greater the spacing, the more difficult it is to estimate the derivative. The approximation improves as the spacing between the two points decreases.

The **central difference** is the average of the forward and backward differences. For uniform spacing

$$f'(x_k) = \frac{1}{2}\left(\frac{f(x_{k+1}) - f(x_k)}{\Delta x} + \frac{f(x_k) - f(x_{k-1})}{\Delta x}\right) = \frac{f(x_{k+1}) - f(x_{k-1})}{2\Delta x}$$

This average tends to cancel out the effects of measurement error.

The second derivative of a function $f(x)$ is the derivative of the first derivative of the function

$$f''(x_k) = \frac{df'(x)}{dx}$$

This function can be approximated using slopes of the first derivative. Using backward differences and assuming uniform spacing Δx

$$\begin{aligned} f''(x_k) &= \frac{f'(x_k) - f'(x_{k-1})}{\Delta x} \\ &= \frac{1}{\Delta x}\left[\frac{f(x_k) - f(x_{k-1})}{\Delta x} - \frac{f(x_{k-1}) - f(x_{k-2})}{\Delta x}\right] \\ &= \frac{f(x_k) - 2f(x_{k-1}) + f(x_{k-2})}{(\Delta x)^2} \end{aligned}$$

`diff` Function

The `diff` function computes differences between adjacent values in a vector, generating a new vector with one less value.

`diff(x)` For vector `x`, returns a vector of differences between adjacent values in `x`: `[x(2)-x(1) x(3)-x(2) ... x(n)-x(n-1)]`, where `n` is `length(x)`. For a matrix `x`, returns the matrix of column differences, `[x(2:n,:) - x(1:n-1,:)]`.

Example:

```
>> x = [0,1,2,3,4,5];
>> y = [2,3,1,5,8,10];
>> dx = diff(x)
dx =
     1     1     1     1     1
>> dy = diff(y)
dy =
     1    -2     4     3     2
```

An approximate derivative $f'(x)$ is computed using `diff` by dividing a change in $y = f(x)$ by a change in x.

```
>> df = diff(y)./diff(x)
df =
     1    -2     4     3     2
```

Obviously, in this example, since samples of `x` are spaced by 1, `dy` and `df` are the same.

Note that the values of `df` are correct for both the forward-difference and the backward-difference approximation to the derivative, as explained above. The distinction between the two methods of approximating the derivative is determined by the values of `x` that correspond to the derivative `dy`. Since `dy` has length one less than that of `x`, it must be related to a vector `xd` that is the vector `x` truncated by one sample at either the beginning or the end. The backward-difference approximation of the derivative is related to `x` truncated at the beginning:

```
>> xd = x(2:end)
xd =
     1     2     3     4     5
```

The forward-difference approximation of the derivative is related to `x` truncated at the end:

```
>> xd = x(1:end-1)
xd =
     0     1     2     3     4
```

The backward difference and central difference methods can be compared by considering a sinusoidal signal that is measured 51 times during one half-period. The measurements are in error by a Gaussian distributed error with standard deviation of 0.025. Figure 11.2 shows the measured data and the underlying sine curve. The following script implements the two methods. The results are shown in Figure 11.2. Clearly the central difference method provides better results in this example.

```
% Comparison of numerical derivative algorithms
x = [0:pi/50:pi];
n = length(x);
% Sine signal with Gaussian random error
yn = sin(x)+0.025*randn(1,n);
% Derivative of noiseless sine signal
td = cos(x);
% Backward difference estimate noisy sine signal
dynb = diff(yn)./diff(x);
subplot(2,1,1),plot(x(2:n),td(2:n),x(2:n),dynb,'o'),xlabel('x'),...
  ylabel('Derivative'),axis([0 pi -2 2]),...
  legend('True derivative','Backward difference')
% Central difference
dync = (yn(3:n)-yn(1:n-2))./(x(3:n)-x(1:n-2));
subplot(2,1,2),plot(x(2:n-1),td(2:n-1),x(2:n-1),dync,'o'),xlabel('x'),...
  ylabel('Derivative'),axis([0 pi -2 2]),...
  legend('True derivative','Central difference')
```

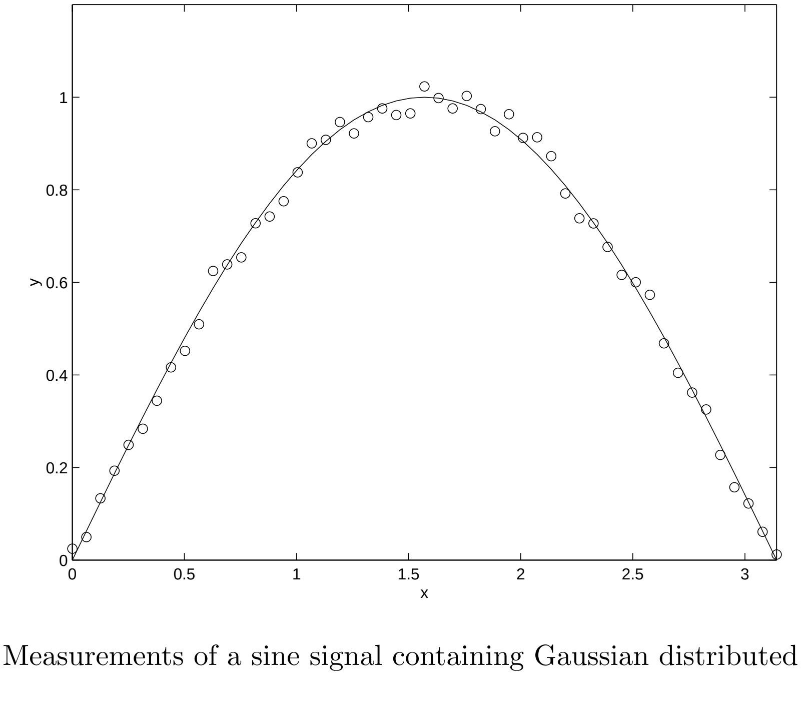


Figure 11.8: Measurements of a sine signal containing Gaussian distributed random errors

Example 11.2 *Numerical differentiation of a polynomial function*

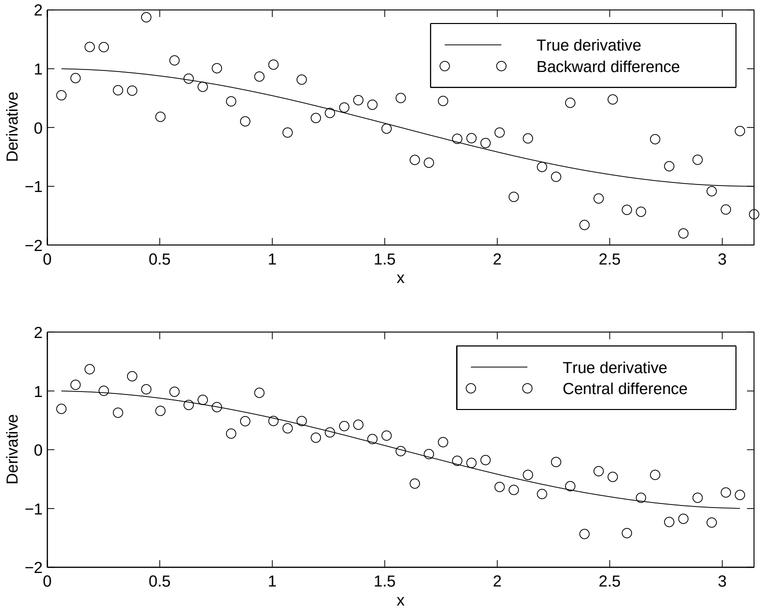


Figure 11.9: Comparison of backward difference and central difference methods

Determine the linear acceleration of an object whose speed is defined by $s(t) = t^3 - 2t^2 + 2$ m/s, where t is in seconds, over the interval 0 to 5. Determine the specific acceleration at $t = 2.5$s.

Consider computing the derivative function at three different resolutions to determine the effect of increasing the resolution. Begin with a time resolution of $\Delta t = 0.1$ second, compute the speed at each of the time values and compute the acceleration by approximating the derivative of the speed with respect to time:

```
>> t = 0:0.1:5;
>> s = t.^3 - 2*t.^2 + 2;
>> ds = diff(s)./diff(t);
```

To determine the acceleration at $t = 2.5$s, it is first necessary to find the time index corresponding to this time. Relating t to find the index k and the time interval Δt:

$$t = \Delta t \cdot (k - 1)$$

Thus

$$k = \frac{t}{\Delta t} + 1$$

or

$$k = \frac{2.5}{0.1} + 1 = 26$$

and

```
>> ds(26)
ans =
    9.3100
```

Thus, the acceleration is 9.31 meters/second^2.

Repeating the computations for $\Delta t = 0.01$ second:

```
>> t = 0:0.01:5;
>> s = t.^3 - 2*t.^2 + 2;
>> ds = diff(s)./diff(t);
```

The time index k for $t = 2.5$s:

$$k = \frac{2.5}{0.01} + 1 = 251$$

The computed acceleration:

```
>> ds(251)
ans =
    8.8051
```

Repeating again for a resolution $\Delta t = 0.001$s, where we are interested in the acceleration at index

$$k = \frac{2.5}{0.001} + 1 = 2501$$

```
>> t = 0:0.001:5;
>> s = t.^3 - 2*t.^2 + 2;
>> ds = diff(s)./diff(t);
>> ds (2501)
ans =
    8.7555
```

Note that the result decreases with each decrease in the time interval. To check the answer, analytically differentiate $s(t)$ to obtain:

$$s'(t) = 3t^2 - 4t$$

At $t = 2.5$s:

$$s'(2.5) = 3(2.5)^2 - 4 \cdot 2.4 = 8.75\text{m/s}^2$$

Thus, we observe that the approximation is converging to the correct result.

Since the Matlab diff function returns only an approximate derivative, it is necessary to use the resolution required to achieve the accuracy desired.

■

Differentiation Error Sensitivity

Differentiation is sensitive to minor changes in the shape of a function, as any small change in the function can easily create large changes in its slope in the neighborhood of the change.

Because of this inherent sensitivity in differentiation, numerical differentiation is avoided whenever possible, especially if the data to be differentiated are obtained experimentally. In this case, it is best to perform a least squares curve fit to the data and then differentiate the resulting polynomial. For example, reconsider the example from the section on curve fitting (Section 10).

In the script below, the x and y data values determined by assignment statements and a second degree curve is fit to the data and plotted as the first subplot. The derivative is approximated from the data and then from the second degree curve fit and these curves are plotted as the second subplot. Since diff computes the difference between elements of a vector, the resulting vector contains one less element than the original vector. Thus, to plot the derivative, one element of the x vector is discarded to form the vector xd. The last element was discarded, so yd is a forward difference approximation.

```
% Generate noisy data
x = 0:0.1:1;
y = [-0.447 1.978 3.28 6.16 7.08 7.34 7.66 9.56 9.48 9.30 11.2];
% 2nd degree curve fit
a = polyfit(x,y,2);
xi = linspace(0,1,101);
yi = polyval(a,xi);
subplot(2,1,1),plot(x,y,'--o',xi,yi),...
  xlabel('x'), ylabel('y'),...
  title('Noisy data and 2nd degree curve fit'),...
  legend('Noisy data','2nd Degree Fit',4)
% Differentiate noise data and fitted curve
yd = diff(y)./diff(x);
ad = polyder(a);
yid = polyval(ad,xi);
xd = x(1:end-1);
subplot(2,1,2),plot(xd,yd,'--o',xi,yid),...
  xlabel('x'), ylabel('dy/dx'),...
  title('Derivative approximations'),...
  legend('Noisy data','2nd Degree Fit',3)
```

The resulting plot is shown in Figure 11.10. Observing the approximation to the derivative shown in dashed lines in the bottom subplot, it is overwhelmingly apparent that approximating the derivative by finite differences can lead to poor results. Note that the derivative of the second order curve fit shown in the solid curve in the bottom subplot does not show the large fluctuations of the approximation.

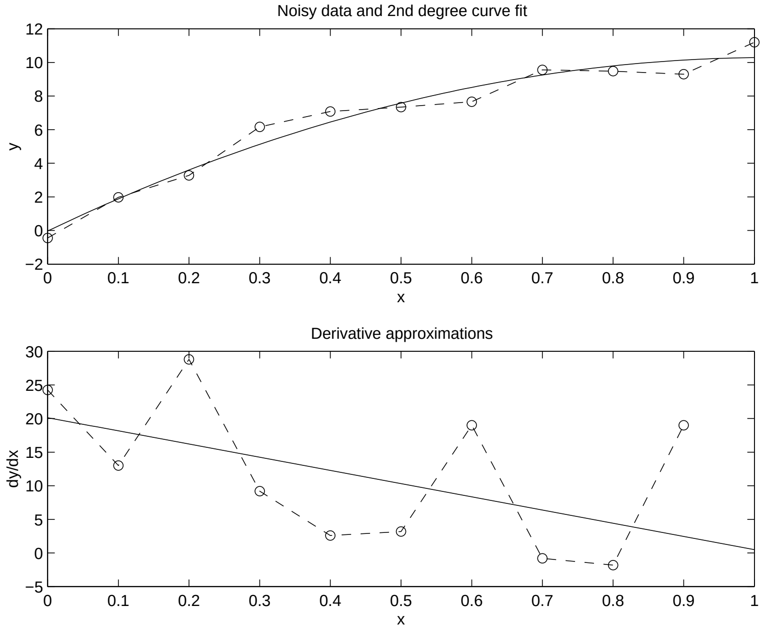


Figure 11.10: Curve fitting and derivative approximation

Section 12

Strings, Time, Base Conversion and Bit Operations

12.1 Character Strings

While MATLAB is primarily intended for number crunching, there are times when it is desirable to manipulate text, as is needed in placing labels and titles on plots. In MATLAB, text is referred to as character strings or simply strings.

String Construction

Character strings in MATLAB are special numerical arrays of ASCII values that are displayed as their character string representation. For example:

```
>> text = ’This is a character string’
text =
This is a character string
>> size(text)
ans =
     1    26
>> whos
  Name       Size         Bytes  Class

  ans        1x2             16  double array
  text       1x26            52  char array

Grand total is 28 elements using 68 bytes
```

ASCII Codes

Each character in a string is one element in an array that requires two bytes per character for storage, using the ASCII code. This differs from the eight bytes per element required for numerical or double arrays. The ASCII codes for the letters ‘A’ to ‘Z’ are the consecutive integers from 65 to 90, while the codes for ‘a’ to ‘z’ are 97 to 122. The function `abs` returns the ASCII codes for a string.

```
>> text = ’This is a character string’
text =
This is a character string
>> d = abs(text)
d =
  Columns 1 through 12
    84   104   105   115    32   105   115    32    97    32    99   104
  Columns 13 through 24
    97   114    97    99   116   101   114    32   115   116   114   105
  Columns 25 through 26
   110   103
```

The function `char` performs the inverse transformation, from ASCII codes to a string:

```
>> char(d)
ans =
This is a character string
```

The relationship between strings and their ASCII codes allow you to do the following:

```
>> alpha = abs(’a’):abs(’z’);
>> disp(char(alpha))
abcdefghijklmnopqrstuvwxyz
```

Strings are Arrays

Since strings are arrays, they can be manipulated with array manipulation tools:

```
>> text = ’This is a character string’;
>> u = text(11:19)
u =
character
```

As with matrices, character strings can have multiple rows, **but each row must have an equal number of columns.** Therefore, blanks are explicitly required to make all rows the same length. For example:

```
>> v = [’Character strings having more than’
        ’one row must have the same number ’
        ’of columns - just like matrices   ’]
v =
Character strings having more than
one row must have the same number
of columns - just like matrices
>> size(v)
ans =
     3    34
```

Concatenation of Strings

Because strings are arrays, they may be *concatenated* (joined) with square brackets. For example:

```
>> today = ’May’;
>> today = [today, ’ 18’]
today =
May 18
```

String Conversions

`char(x)` Converts the array `x` that contains positive integers representing character codes into a character array (the first 127 codes are ASCII). The result for any elements of `x` outside the range from 0 to 65535 is not defined.

`int2str(x)` Rounds the elements of the matrix `x` to integers and converts the result into a string matrix.

`num2str(x)` Converts the matrix `x` into a string representation with about 4 digits and an exponent if required. This is useful for labeling plots with the `title`, `xlabel`, `ylabel`, and `text` commands.

`str2num(s)` Converts the string `s`, which should be an ASCII character representation of a numeric value, to numeric representation. The string may contain digits, a decimal point, a leading + or - sign, an ’e’ preceeding a power of 10 scale factor, and an ’i’ for a complex unit.

The `num2str` function can be used to convert numerical results into strings for use in formating displayed results with `disp`. For example, consider the following portion of a script:

```
tg = 2.2774;
xg = 144.6364;
disp([’time of flight:  ’ num2str(tg) ’ s’])
disp([’distance traveled  : ’ num2str(xg) ’ ft’])
```

The arguments for each of the `disp` commands are vectors of strings, with the first element being a label, the second being a number converted to a string, and the third being the units of the quantity. The label, the numerical results, and the units are displayed on one line, which was not possible with other forms of the use of `disp`:

```
time of flight:  2.2774 s
distance traveled:  144.6364 ft
```

String Functions

`blanks(n)` Returns a string of `n` blanks. Used with `disp`, eg. `disp ([’xxx’ blanks(20) ’yyy’])`. `disp(blanks(n)’)` moves the cursor down `n` lines.

`deblank(s)` Removes trailing blanks from string `s`.

`eval(s)` Execute the string `s` as a Matlab expression or statement.

`eval(s1,s2)` Provides the ability to catch errors. Executes string `s1` and returns if the operation was successful. If the operation generates an error, string `s2` is evaluated before returning. This can be thought of as `eval(’try’,’catch’)`.

`findstr(s1,s2)` Find one string within another. Returns the starting indices of any occurrences of the shorter of the two strings in the longer.

`ischar(s)` Returns 1 if `s` is a character array and 0 otherwise.

`isletter(s)` Returns 1 for each element of character array `s` containing letters of the alphabet and 0 otherwise.

`isspace(s)` Returns 1 for each element of character `s` containing white space characters and 0 otherwise. White space characters are spaces, newlines, carriage returns, tabs, vertical tabs, and formfeeds.

`lasterr` Returns a string containing the last error message issued. `lasterr` is usually used in conjunction with the two argument form of `eval`: `eval(’try’,’catch’)`. The ’catch’ action can examine the `lasterr` string to determine the cause of the error and take appropriate action.

`lower(s)` Converts any uppercase characters in string `s` to the corresponding lowercase character and leaves all other characters unchanged.

`strcat(s1,s2,s3,...)` Horizontally concatenates corresponding rows of the character arrays `s1`, `s2`, `s3` etc. The trailing padding is ignored. All the inputs must have the same number of rows (or any can be a single string). When the inputs are all character arrays, the output is also a character array.

`strcmp(s1,s2)` Returns 1 if strings `s1` and `s2` are identical and 0 otherwise.

`strjust(s)` Returns a right justified version of the character array `s`.

`strmatch(str,strs)` Searches through the rows of the character array of strings `strs` to find strings that begin with string `str`, returning the matching row indices.

`strncmp(s1,s2,n)` Returns 1 if the first `n` characters of the strings `s1` and `s2` are identical and 0 otherwise.

`strrep(s1,s2,s3)` Replaces all occurrences of the string `s2` in string `s1` with the string `s3`. The new string is returned.

`upper(s)` Converts any lower case characters in `s` to the corresponding upper case character and leaves all other characters unchanged.

12.2 Time Computations

Current Date and Time

Three formats are supported for dates:

`clock` Returns a six element date vector vector containing the current time and date in decimal form: `[year month day hour minute seconds]`. The first five elements are integers. The seconds element is accurate to several digits beyond the decimal point.

`date` Returns a string containing the date in dd-mmm-yyyy format.

`now` Returns the current date and time as a serial date number.

Examples:

```
>> t = clock
t =
       1998          6         10         10         18      59.57
>> date
ans =
10-Jun-1998
>> format long
>> d = now
d =
    7.299164376449074e+005
```

The date number can be converted to a string with the `datestr` function:

`datestr(d,dateform)`: Converts a data number `d` (such as that returned by `now`) into a date string. The string is formatted according to the format number `dateform` (see table below). By default, `dateform` is 1, 16, or 0 depending on whether `d` contains dates, times or both.

`dateform`	Date format	Example
0	`’dd-mmm-yyyy HH:MM:SS’`	`01-Mar-1995 15:45:17`
2	`’mm/dd/yy’`	`03/01/95`
3	`’mmm’`	`Mar`
4	`’m’`	`M`
5	`’mm’`	`3`
6	`’mm/dd’`	`03/01`
7	`’dd’`	`1`
8	`’ddd’`	`Wed`
9	`’d’`	`W`
10	`’yyyy’`	`1995`
11	`’yy’`	`95`
12	`’mmmyy’`	`Mar95`
13	`’HH:MM:SS’`	`15:45:17`
14	`’HH:MM:SS PM’`	`3:45:17 PM`
15	`’HH:MM’`	`15:45`
16	`’HH:MM PM’`	`3:45 PM`
17	`’QQ-YY’`	`Q1-96`
18	`’QQ’`	`Q1`

Examples:

```
>> ds = datestr(d)
ds =
10-Jun-1998 10:30:13
>> datestr(d,14)
ans =
10:30:13 AM
```

The function `datenum` is used to compute a date number. It has three forms:

`datenum(s)`: Returns the date number corresponding to the date string `s`.

`datenum(year,month,day)`: Returns the date number corresponding to the specified `year`, `month`, and `day`.

`datenum(year,month,day,hour minute,second)`: Returns the date number corresponding to the specified `year`, `month`, `day`, `hour`, `minute`, and `second`.

Examples:

```
>> datenum(ds)
ans =
    7.299164376504630e+005
>> datenum(1998,6,13)
ans =
      729919
```

```
>> datenum(1998,6,16,10,30,00)
ans =
    7.299224375000000e+005
```

Date Functions

The day of the week may be found from a date string or a date number using `weekday`, using the convention that Sunday = 1 and Saturday = 7.

```
>> [d s] = weekday('9/9/90')
d =
     1
s =
Sun
```

A calendar can be generated for a desired month, for display in the *Command* window or to be placed in a 6-by-7 array.

```
>> calendar('9/9/90')
                   Sep 1990
     S     M    Tu     W    Th     F     S
     0     0     0     0     0     0     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     0     0     0     0     0     0
>> a = calendar(1978,6)
a =
     0     0     0     0     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     0
     0     0     0     0     0     0     0
```

Timing Functions

The functions `tic` and `toc` are used to time a sequence of commands. `tic` starts the timer; `toc` stops the timer and displays the elapsed time in seconds.

Example:

```
tic
```

```
for t=1:5000
    y(t)=sin(2*pi*t/10);
end
toc
```

Executing:

```
elapsed_time =
    4.5100
```

cputime returns the amount of central processing unit (CPU) time in seconds since the current session was started. Computing processing times at various points in a script and taking differences can be used to determine the CPU time required for segments of the script, possibly locating portions of the code that could be rewritten to decrease the total computation time.

12.3 Base Conversions and Bit Operations

Base Conversion

MATLAB provides functions for converting decimal numbers to other bases in the form of character strings. These conversion functions include:

dec2bin(d)	Returns the binary representation of d as a string. d must be a non-negative integer smaller than 2^{52}.
dec2bin(d,N)	Produces a binary representation with at least N bits.
bin2dec(b)	Interprets the binary string b and returns the equivalent decimal number.
dec2hex(d)	Returns the hexadecimal representation of decimal integer d as a string. d must be a non-negative integer smaller than 2^{52}.
hex2dec(h)	Interprets the hexadecimal string h and returns the equivalent decimal number. If h is a character array, each row is interpreted as a hexadecimal string.
dec2base(d,b)	Returns the representation of d as a string in base b. d must be a non-negative integer smaller than 2^{52} and b must be an integer between 2 and 36.
dec2base(d,b,N)	Produces a representation with at least N digits.

Examples:

```
>> a = dec2bin(18)    % find binary representation of 18
a =
10010
>> bin2dec(a)         % convert a back to decimal
ans =
    18
```

```
>> b = dec2hex(30)    % hex representation of 30
b =
1E
>> hex2dec(b)         % convert b back to decimal
ans =
    30
>> c = dec2base(30,4) % 30 in base 4
c =
132
>> base2dec(c,4)      % convert c back to decimal
ans =
    30
```

Bit Operations

Matlab provides functions to implement logical operations on the individual bits of floating-point integers. The Matlab bitwise functions operate on integers between 0 and `bitmax`, which is $2^{53} - 1 = 9.0072 \times 10^{15}$:

`c = bitand(a,b)` Returns the bit-wise AND of the two arguments `a` and `b`.

`c = bitor(a,b)` Returns the bit-wise OR of the two arguments `a` and `b`.

`c = bitxor(a,b)` Returns the bit-wise exclusive OR of the two arguments `a` and `b`.

`c = bitcmp(a,N)` Returns the bit-wise complement of `a` as an `N`-bit non-negative integer.

`c = bitset(a,bit,v)` Sets the bit at position `bit` to the value `v`, where `v` must be either 0 or 1.

`c = bitget(a,bit)` Returns the value of the bit at position `bit` in `a`. `a` must contain non-negative integers and `bit` must be a number between 1 and the number of bits in a floating point integer (52 for IEEE machines).

`c = bitshift(a,N)` Returns the value of `a` shifted by `N` bits. If `N > 0`, this is same as a multiplication by 2^N (left shift). If `N < 0`, this is the same as a division by `2^(-N)` (right shift).

Examples of the use of these functions are best understood by displaying the results of each operation by `dec2bin`:

```
>> a = 6;
>> b = 10;
>> dec2bin(a,4)
ans =
0110
>> dec2bin(b,4)
ans =
1010
>> dec2bin(bitand(a,b),4)
ans =
0010
```

```
>> dec2bin(bitor(a,b),4)
ans =
1110
>> dec2bin(bitxor(a,b),4)
ans =
1100
>> dec2bin(bitcmp(a,4),4)
ans =
1001
>> dec2bin(bitset(a,4,1),4)
ans =
1110
>> bitget(a,2)
ans =
     1
>> dec2bin(bitshift(a,1))
ans =
1100
```

Section 13

Symbolic Processing

We have focused on the use of Matlab to perform numerical operations, involving numerical data represented by double precision floating point numbers. We also given some consideration to the manipulation of text, represented by character strings. In this section, we introduce the use of Matlab to perform *symbolic processing* to manipulate mathematical expressions, in much the way that we do with pencil and paper.

The objective of symbolic processing is to obtain what are known as *closed form* solutions, expressions that don't need to be iterated or updated in order to obtain answers. An understanding of these solutions often provides better physical and mathematical insight into the problem under investigation.

For more information, type `help symbolic` in Matlab. A tutorial demonstration is available with the command `symintro`.

The following notes represent a short introduction to the symbolic processing capabilities of Matlab.

13.1 Symbolic Expressions and Algebra

To introduce symbolic processing, first consider the handling of symbolic expressions and the manipulation of algebra.

Declaring Symbolic Variables and Constants

To enable symbolic processing, the variables and constants involved must first be declared as *symbolic objects.*

For example, to create the symbolic variables with names `x` and `y`:

```
>> syms x y
```

If x and y are to be assumed to be real variables, they are created with the command:

```
>> syms x y real
```

To declare symbolic constants, the sym function is used. Its argument is a string containing the name of a special variable, a numeric expression, or a function evaluation. It is used in an assignment statement which serves as a declaration of a symbolic variable for the assigned variable. Examples include:

```
>> pi = sym('pi');
>> delta = sym('1/10');
>> sqroot2 = sym('sqrt(2)');
```

If the symbolic constant pi is created this way, it replaces the special variable pi in the workspace. The advantage of using symbolic constants is that they maintain full accuracy until a numeric evaluation is required.

Symbolic variables and constants are represented by the data type *symbolic object*. For example, as displayed by the function whos for the symbolic variables and constants declared in the commands above:

```
>> whos
  Name           Size         Bytes  Class

  delta          1x1            132  sym object
  pi             1x1            128  sym object
  sqroot2        1x1            138  sym object
  x              1x1            126  sym object
  y              1x1            126  sym object

Grand total is 20 elements using 650 bytes
```

Symbolic Expressions

Symbolic variables can be used in expressions and as arguments of functions in much the same way as numeric variables have been used. The operators + - * / ^ and the built-in functions can also be used in the same way as they have been used in numeric calculations. For example:

```
>> syms s t A
>> f = s^2 + 4*s + 5
f =
s^2+4*s+5
>> g = s + 2
g =
s+2
```

```
>> h = f*g
h =
(s^2+4*s+5)*(s+2)
>> z = exp(-s*t)
z =
exp(-s*t)
>> y = A*exp(-s*t)
y =
A*exp(-s*t)
```

The symbolic variables `s`, `t`, and `A` are first declared, then used in symbolic expressions to create the symbolic variables `f`, `g`, `h`, `z`, and `y`. The displayed results show that the created variables remain as symbolic objects, rather than being evaluated numerically. Symbolic processing doesn't seem to obey the `format compact` command, as the displayed output is always double-spaced. These blank lines have been removed from these notes to conserve paper.

The variable `x` is the *default* independent variable, but as can be seen with the expressions above, other variables can be specified to be the independent variable. It is important to know which variable is the independent variable in an expression. The command to find the independent variable is:

`findsym(S)` Finds the symbolic variables in a symbolic expression or matrix `S` by returning a string containing all of the symbolic variables appearing in `S`. The variables are returned in alphabetical order and are separated by commas. If no symbolic variables are found, `findsym` returns the empty string.

Examples based on the declarations and expressions in the examples above are:

```
>> findsym(f)
ans =
s
>> findsym(z)
ans =
A, s, t
```

The vector and matrix notation used in MATLAB also applies to symbolic variables. For example, the symbolic matrix `B` can be created with the commands:

```
>> n = 3;
>> syms x
>> B = x.^((0:n)'*(0:n))
B =
[   1,   1,   1,   1]
[   1,   x, x^2, x^3]
[   1, x^2, x^4, x^6]
[   1, x^3, x^6, x^9]
```

Manipulating Polynomial Expressions

In the examples above, symbolic variables were declared and were used in symbolic expressions to create polynomials. We now wish to manipulate these polynomial expressions algebraically.

The Matlab commands for this purpose include:

`expand(S)` Expands each element of a symbolic expression `S` as a product of its factors. `expand` is most often used on polynomials, but also expands trigonometric, exponential and logarithmic functions.

`factor(S)` Factors each element of the symbolic matrix `S`.

`simplify(S)` Simplifies each element of the symbolic matrix `S`.

`[n,d] = numden(S)` Returns two symbolic expressions that represent the numerator expression `num` and the denominator expression `den` for the rational representation of the symbolic expression `S`.

`subs(S,old,new)` Symbolic substitution, replacing symbolic variable `old` with symbolic variable `new` in the symbolic expression `S`.

These commands can be used to implement symbolic polynomial operations that were previously considered as numeric operations in Section 7.2 of these notes.

- **Addition:**

  ```
  >> syms s
  >> A = s^4 -3*s^3 -s +2;
  >> B = 4*s^3 -2*s^2 +5*s -16;
  >> C = A + B
  C =
  s^4+s^3+4*s-14-2*s^2
  ```

  Note that the result is correct, although it is not in form the we prefer, as the terms are not ordered in decreasing powers of `s`.

- **Scalar multiple:**

  ```
  >> syms s
  >> A = s^4 -3*s^3 -s +2;
  >> C = 3*A
  C =
  3*s^4-9*s^3-3*s+6
  ```

- **Multiplication:**

  ```
  >> syms s
  >> A = s+2;
  >> B = s+3;
  >> C = A*B
  ```

```
C =
(s+2)*(s+3)
>> C = expand(C)
C =
s^2+5*s+6
```

- **Factoring:**

```
>> syms s
>> D = s^2 + 6*s + 9;
>> D = factor(D)
D =
(s+3)^2
>> P = s^3 - 2*s^2 -3*s + 10;
>> P = factor(P)
P =
(s+2)*(s^2-4*s+5)
```

- **Common denominator:** Consider the expression

$$H(s) = -\frac{1/6}{s+3} - \frac{1/2}{s+1} + \frac{2/3}{s}$$

This can be expressed as a ratio of polynomials by finding the common denominator for the three terms, as follows:

```
>> syms s
>> H = -(1/6)/(s+3) -(1/2)/(s+1) +(2/3)/s;
>> [N,D] = numden(H)
N =
s+2
D =
(s+3)*(s+1)*s
>> D = expand(D)
D =
s^3+4*s^2+3*s
```

Thus, $H(s)$ can be expressed in the form

$$H(s) = \frac{s+2}{s^3+4s^2+3s}$$

As a second example, consider

$$G(s) = s + 4 + \frac{2}{s+4} + \frac{3}{s+2}$$

Manipulating with Matlab:

```
>> syms s
>> G = s+4 + 2/(s+4) + 3/(s+2);
>> [N,D] = numden(G)
N =
s^3+10*s^2+37*s+48

D =
(s+4)*(s+2)
>> D = expand(D)
D =
s^2+6*s+8
```

Thus, $G(s)$ can also be expressed in the form:

$$G(s) = \frac{s^3 + 10s^2 + 37s + 48}{s^2 + 6s + 8}$$

In this example, $G(s)$ is an improper rational function.

- **Cancellation of terms**: For a ratio of polynomials, MATLAB can be applied to see if any terms cancel. For example

  $$H(s) = \frac{s^3 + 2s^2 + 5s + 10}{s^2 + 5}$$

  Applying MATLAB:

  ```
  >> syms s
  >> H = (s^3 +2*s^2 +5*s +10)/(s^2 + 5);
  >> H = simplify(H)
  H =
  s+2
  ```

  Factoring the denominator shows why the cancellation occurs:

  ```
  >> factor(s^3 +2*s^2 +5*s +10)
  ans =
  (s+2)*(s^2+5)
  ```

  Thus,

  $$H(s) = s + 2$$

- **Variable substitution:** Consider the ratio of polynomials

  $$H(s) = \frac{s+3}{s^2 + 6s + 8}$$

  Define a second expression

  $$G(s) = H(s)|_{s=s+2}$$

  Evaluating $G(s)$ with MATLAB:

```
>> syms s
>> H = (s+3)/(s^2 +6*s + 8);
>> G = subs(H,s,s+2)
G =
(s+5)/((s+2)^2+6*s+20)
>> G = collect(G)
G =
(s+5)/(s^2+10*s+24)
```

Thus

$$G(s) = \frac{s+5}{s^2+10s+24}$$

Commands are also provided to convert between the numeric representation of a polynomial as the vector of its coefficients and the symbolic representation.

`sym2poly(P)` Converts from a symbolic polynomial `P` to a row vector containing the polynomial coefficients.

`poly2sym(p)` Converts from a polynomial coefficient vector `p` to a symbolic polynomial in the variable `x`. `poly2sym(p,v)` uses the symbolic the variable `v`.

For example, consider the polynomial

$$A(s) = s^3 + 4s^2 - 7s - 10$$

In Matlab:

```
>> a = [1 4 -7 -10];
>> A = poly2sym(a,s)
A =
s^3+4*s^2-7*s-10
```

For the polynomial

$$B(s) = 4s^3 - 2s^2 + 5s - 16$$

```
>> syms s
>> B = 4*s^3 -2*s^2 +5*s -16;
>> b = sym2poly(B)
b =
     4    -2     5   -16
```

Forms of Expressions

As we have seen in some of the examples above, Matlab does not always arrange expressions in the form that we would prefer. For example, Matlab expresses results in the form `1/a*b`, while we would prefer `b/a`. For example:

```
>> syms s
>> H = s^2 +6*s + 8;
>> G = -H/3
G =
-1/3*s^2-2*s-8/3
```

The result is $G = -(1/3)s^2 - 2s - 8/3$, while we would prefer $G = -(s^2 + 6s + 8)/3$.

13.2 Manipulating Trigonometric Expressions

Trigonometric expressions can also be manipulated symbolically in MATLAB, primarily with the use of the expand function. For example:

```
>> syms theta phi
>> A = sin(theta + phi)
A =
sin(theta+phi)
>> A = expand(A)
A =
sin(theta)*cos(phi)+cos(theta)*sin(phi)
>> B = cos(2*theta)
B =
cos(2*theta)
>> B = expand(B)
B =
2*cos(theta)^2-1
>> C = 6*((sin(theta))^2 + (cos(theta))^2)
C =
6*sin(theta)^2+6*cos(theta)^2
>> C = expand(C)
C =
6*sin(theta)^2+6*cos(theta)^2
```

Thus, MATLAB was able to apply trigonometric identities to expressions A and B, but it was not successful with C, as we know

$$C = 6(\sin^2(\theta) + \cos^2(\theta)) = 6$$

MATLAB can also manipulate expressions involving complex exponential functions. For example:

```
>> syms theta real
>> A = real(exp(j*theta))
A =
1/2*exp(i*theta)+1/2*exp(-i*theta)
```

```
>> A = simplify(A)
A =
cos(theta)
```

13.3 Evaluating and Plotting Symbolic Expressions

In many applications, we eventually want to obtain numerical results or a plot from a symbolic expression. The function `double` produces numerical results:

`double(S)` Converts the symbolic matrix expression `S` to a matrix of double precision floating point numbers. `S` must not contain any symbolic variables, except possibly `eps`.

Since the symbolic expression cannot contain any symbolic variables, it is necessary to use `subs` to substitute numerical values for the symbolic variables prior to applying `double`. For example:

```
>> E = s^3 -14*s^2 +65*s -100;
>> F = subs(E,s,7.1)
F =
13671/1000
>> G = double(F)
G =
   13.6710
```

The symbolic form is `F` and the numeric quantity is `G`, as confirmed by the display from `whos`:

```
>> whos
  Name      Size         Bytes  Class

  E         1x1            162  sym object
  F         1x1            144  sym object
  G         1x1              8  double array
  s         1x1            126  sym object

Grand total is 34 elements using 440 bytes
```

Symbolic expressions can be plotted with the MATLAB function `ezplot`:

`ezplot(f)` Plots a graph of `f(x)` where `f` is a string or a symbolic expression representing a mathematical expression involving a single symbolic variable, say `x`. The default range of the x-axis is $[-2\pi, 2\pi]$

`ezplot(f,xmin,xmax)` Plots the graph using the specified x-range instead of the default range.

For example consider plotting the polynomial function

$$A(s) = s^3 + 4s^2 - 7s - 10$$

over the range $[-1, 3]$:

```
syms s
a = [1 4 -7 -10];
A = poly2sym(a,s)
ezplot(A,-1,3), ylabel('A(s)')
```

The resulting plot is shown in Figure 13.3. Note that the expression plotted is automatically placed at the top of the plot and that the axis label for the independent variable is automatically placed. A `ylabel` command was used to label the dependent variable.

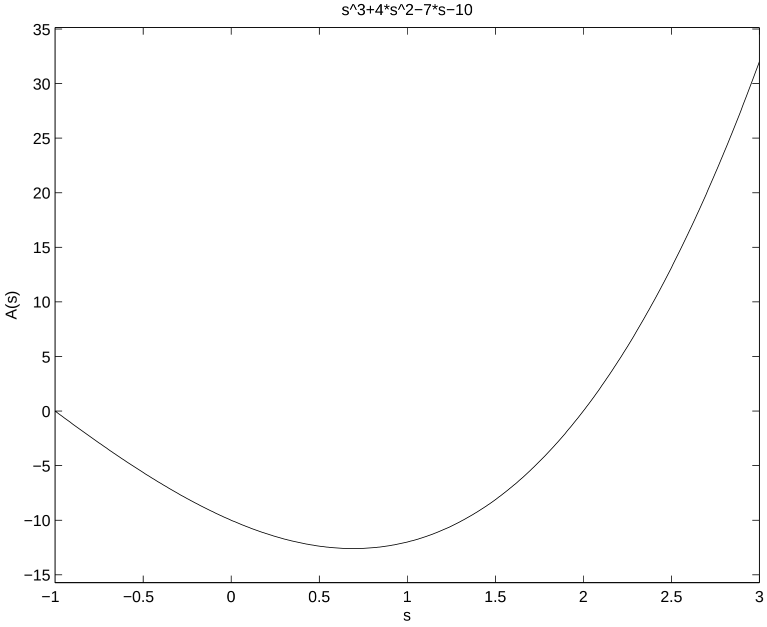


Figure 13.1: Plot of a polynomial function using `ezplot`

13.4 Solving Algebraic and Transcendental Equations

The symbolic math toolbox can be used to solve algebraic and transcendental equations, as well as systems of such equations. A *transcendental* equation is one that contains one or more transcendental functions, such as $\cos x$, e^x, or $\ln x$.

The function used in solving these equations is `solve`. There are several forms of `solve`, but only the following forms will be presented in these notes:

```
solve(E1, E2,...,EN)
solve(E1, E2,...,EN, var1, var2,...,varN)
```

where `E1, E2,...,EN` are the names of symbolic expressions and `var1, var2,..., varN` are variables in the expressions that have been declared to be symbolic. The solutions obtained are the roots of the expressions; that is, symbolic expressions for the variables under the conditions `E1 = 0`, `E2 = 0`, ... `EN = 0`.

For one equation and one variable, the resulting output solution is returned as a single symbolic variable.

For example:

```
>> syms s
>> E = s+2;
>> s = solve(E)
s =
-2
```

For `N` equations, the `N` solutions are returned as a symbolic vector.

For example:

```
>> syms s
>> D = s^2 +6*s +9;
>> s = solve(D)
s =
[ -3]
[ -3]
```

Thus, the solution is the symbolic representation of the repeated real roots of the quadratic polynomial, providing the same results as those obtained earlier in numeric representation using the function `roots`. Complex roots can also be obtained, as shown in the following example:

```
>> syms s
>> P = s^3 -2*s^2 -3*s + 10;
>> s = solve(P)
s =
[  -2]
[ 2+i]
[ 2-i]
```

Similar results can be obtained in the solution of transcendental equations. An example in trigonometry:

```
>> syms theta x z
>> E = z*cos(theta) - x;
>> theta = solve(E,theta)
theta =
acos(x/z)
```

For an example involving e^x, consider the solution to $e^{2x} + 4e^x = 32$:

```
>> syms x
>> E = exp(2*x) + 4*exp(x) -32;
>> x = solve(E)
x =
[ log(-8)]
[  log(4)]
>> log(-8)
ans =
   2.0794+ 3.1416i
>> log(4)
ans =
    1.3863
```

Note that two solutions are provided, with the numeric results showing that the first solution $\log(-8) = 2.0794 + 3.1416i$ is complex, while the second solution $\log(4) = 1.3863$ is real. The issue as to whether both of these solutions are meaningful would depend on the application that led to the original equation.

Equations containing periodic functions can have an infinite number of solutions. In such cases, `solve` restricts the search for solutions to the region near 0. For example, to solve the equation $\cos(2\theta) - \sin(\theta) = 0$:

```
>> E = cos(2*theta)-sin(theta);
>> solve(E)
ans =
[ -1/2*pi]
[  1/6*pi]
[  5/6*pi]
```

Example 13.1 *Positioning a robot arm*

Consider again the application to robot motion that was presented in Section 10.4. The robot arm has two joints and two links. The (x, y) coordinates of the hand are given by

$$x_1 = L_1 \cos\theta_1 + L_2 \cos(\theta_1 + \theta_2)$$

$$x_2 = L_1 \sin\theta_1 + L_2 \sin(\theta_1 + \theta_2)$$

where θ_1 and θ_2 are the joint angles and $L_1 = 4$ feet and $L_2 = 3$ feet are the link lengths. A part of the previous solution that was not determined was the joint angles needed to position the hand at a given set of coordinates. For the initial hand position of $(x, y) = (6.5, 0)$, the following commands determine the required angles:

```
>> syms theta1 theta2
>> E1 = 4*cos(theta1)+3*cos(theta1+theta2)-6.5;
>> E2 = 4*sin(theta1)+3*sin(theta1+theta2);
>> [theta1, theta2] = solve(E1,E2)
theta1 =
[  atan(9/197*55^(1/2))]
[ atan(-9/197*55^(1/2))]

theta2 =
[  -atan(3/23*55^(1/2))]
[ -atan(-3/23*55^(1/2))]
>> theta1 = double(theta1*(180/pi))
theta1 =
   18.7170
  -18.7170
>> theta2 = double(theta2*(180/pi))
theta2 =
  -44.0486
   44.0486
```

There are two solutions, given first in symbolic form, then converted into numeric form using `double`. The first is $\theta_1 = 18.717°$, $\theta_2 = -44.0486°$, which is the "elbow up" solution. The second is $\theta_1 = -18.717°$, $\theta_2 = 44.0486°$, the "elbow down" solution.

■

13.5 Calculus

Symbolic expressions can be differentiated and integrated to obtain closed form results.

Differentiation

The `diff` function, when applied to a symbolic expression, provides a symbolic derivative.

`diff(E)`	Differentiates a symbolic expression `E` with respect to its free variable as determined by `findsym`.
`diff(E,v)`	Differentiates `E` with respect to symbolic variable `v`.
`diff(E,n)`	Differentiates `E` `n` times for positive integer `n`.
`diff(S,v,n)`	Differentiates `E` `n` times with respect to symbolic variable `v`.

Examples of derivatives of polynomial functions:

```
>> syms s n
>> p = s^3 + 4*s^2 -7*s -10;
>> d = diff(p)
d =
3*s^2+8*s-7
>> e = diff(p,2)
e =
6*s+8
>> f = diff(p,3)
f =
6
>> g = s^n;
>> h = diff(g)
h =
s^n*n/s
>> h = simplify(h)
h =
s^(n-1)*n
```

Examples of derivatives of transcendental functions:

```
>> syms x
>> f1 = log(x);
>> df1 = diff(f1)
df1 =
1/x
>> f2 = (cos(x))^2;
>> df2 = diff(f2)
df2 =
-2*cos(x)*sin(x)
>> f3 = sin(x^2);
>> df3 = diff(f3)
df3 =
2*cos(x^2)*x
>> df3 = simplify(df3)
df3 =
2*cos(x^2)*x
>> f4 = cos(2*x);
>> df4 = diff(f4)
df4 =
-2*sin(2*x)
>> f5 = exp(-(x^2)/2);
>> df5 = diff(f5)
df5 =
-x*exp(-1/2*x^2)
```

Min-Max Problems

The derivative can be used to find the maximum or minimum of a continuous function, say, $f(x)$, over an interval $a \leq x \leq b$. A *local* maximum or local minimum (one that does not occur at one of the boundaries $x = a$ or $x = b$) can occur only at a *critical point*, which is a point where either $df/dx = 0$ or df/dx does not exist.

Example 13.2 *Minimum cost tank design*

Consider again the tank design problem that was solved numerically in Example 7.1. In this problem, the tank radius is R, the height is H and the tank volume is such that

$$500 = \pi R^2 H + \frac{2}{3}\pi R^3$$

The cost of the tank, a function of surface area, is

$$C = 300(2\pi RH) + 400(2\pi R^2)$$

The problem is to solve for R and H providing the minimum cost tank providing the specified volume. The symbolic approach is to solve the volume equation for H as a function of R, express cost C symbolically, then differentiate C with respect to R and solve the resulting equation for R.

```
>> syms R H
>> V = pi*R^2*H + (2/3)*pi*R^3 -500;      % Equation for volume
>> H = solve(V,H)                         % Solve volume for height H
H =
-2/3*(pi*R^3-750)/pi/R^2
>> C = 300*(2*pi*R*H) + 400*(2*pi*R^2);   % Equation for cost
>> dCdR = diff(C,R)                       % Derivative of cost wrt R
dCdR =
400/R^2*(pi*R^3-750)+400*pi*R
>> Rmins = solve(dCdR,R)                  % Solve dC/dR for R: Rmin
Rmins =
[                                  5/pi*3^(1/3)*(pi^2)^(1/3)]
[ -5/2/pi*3^(1/3)*(pi^2)^(1/3)+5/2*i*3^(5/6)/pi*(pi^2)^(1/3)]
[ -5/2/pi*3^(1/3)*(pi^2)^(1/3)-5/2*i*3^(5/6)/pi*(pi^2)^(1/3)]
>> Rmins = double(Rmins)
Rmin =
   4.9237
  -2.4619+ 4.2641i
  -2.4619- 4.2641i
>> Rmin = Rmins(1)
Rmin =
    4.9237
```

```
>> Hmin = double(subs(H,R,Rmin))
Hmin =
    3.2825
>> Cmin = double(subs(C,{R,H},{Rmin,Hmin}))
Cmin =
  9.1394e+004
```

Note that there are three symbolic solutions for R to provide minimum cost (`Rmins`). Converting these solutions to numeric quantities with `double`, we see that the second and third solutions are complex, which are not physically meaningful. Thus, we choose `Rmin` to be `Rmins(1)` and we compute `Hmin` and `Cmin` from this value. These symbolic results obtained here are more accurate than those determined previously in Example 7.1, as there has been no need to consider samples of R and H at a limited resolution. However, note that the results determined by the two methods are very close.

■

Integration

The `int` function, when applied to a symbolic expression, provides a symbolic integration.

`int(E)`	Indefinite integral of symbolic expression `E` with respect to its symbolic variable as defined by `findsym`. If `E` is a constant, the integral is with respect to `x`.
`int(E,v)`	Indefinite integral of `E` with respect to scalar symbolic variable `v`.
`int(E,a,b)`	Definite integral of `E` with respect to its symbolic variable from `a` to `b`, where `a` and `b` are each double or symbolic scalars.
`int(E,v,a,b)`	Definite integral of `E` with respect to `v` from `a` to `b`.

Examples of integrals of polynomials:

```
>> syms x n a b t
>> int(x^n)
ans =
x^(n+1)/(n+1)
>> int(x^3 +4*x^2 + 7*x + 10)
ans =
1/4*x^4+4/3*x^3+7/2*x^2+10*x
>> int(x,1,t)
ans =
1/2*t^2-1/2
>> int(x^3,a,b)
ans =
1/4*b^4-1/4*a^4
```

Examples of integrals of transcendental functions:

```
>> syms x
>> int(1/x)
ans =
log(x)
>> int(cos(x))
ans =
sin(x)
>> int(1/(1+x^2))
ans =
atan(x)
>> int(exp(-x^2))
ans =
1/2*pi^(1/2)*erf(x)
```

The last integral above introduces the error function `erf(x)` for each element of `x`, where `x` is real. The error function is defined as:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

13.6 Linear Algebra

Operations on symbolic matrices can be performed in much the same way as with numeric matrices.

The following are examples of matrix inverse, product, and determinant.

```
>> A = sym([2,1; 4,3])
A =
[ 2, 1]
[ 4, 3]
>> Ainv = inv(A)
Ainv =
[  3/2, -1/2]
[   -2,    1]
>> C = A*Ainv
C =
[ 1, 0]
[ 0, 1]
>> B = sym([1 3 0; -1 5 2; 1 2 1])
B =
[  1,  3,  0]
[ -1,  5,  2]
[  1,  2,  1]
>> detB = det(B)
detB =
10
```

Systems of linear equations can also be solved symbolically. Consider the following example that was previously solved numerically:

```
>> syms x
>> A = sym([3 2 -1; -1 3 2; 1 -1 -1])
A =
[  3,  2, -1]
[ -1,  3,  2]
[  1, -1, -1]
>> b = sym([10; 5; -1])
b =
[ 10]
[  5]
[ -1]
>> x = A\b
x =
[ -2]
[  5]
[ -6]
```

The results are the same as those obtained numerically. For this problem, there is little advantage to finding the result symbolically. However, solving a system of equations with respect to a parameter, the symbolic approach provides an advantage. Consider the following set of equations

$$\begin{aligned} 2x_1 - 3x_2 &= 3 \\ 5x_1 + cx_2 &= 19 \end{aligned}$$

To solve for x_1 and x_2 as functions of the parameter c:

```
>> syms c
>> A = sym([2 -3; 5 c]);
>> b = sym([3; 19]);
>> x = A\b
x =
[ 3*(19+c)/(2*c+15)]
[       23/(2*c+15)]
```