Engineering Programming in MATLAB : A Primer

February, 2000

Mark A. Austin

Institute for Systems Research,
University of Maryland,
College Park,
Maryland 20742, U.S.A.

Contents

Part I

Introduction to Engineering Computations

Introduction

This text begins with a tutorial describing the concepts on which modern engineering computations are built. In our experience, students are much better prepared to learn a new programming language if they are already familiar with these basic concepts.

After briefly explaining the range of application programs that are found in engineering organizations, Chapter 1 quickly reviews the major contributions of computer technology over the past thirty years. This historical review helps us to see where and how technology has evolved, and provides perspective for where computing and computer technologies are likely to head in the next five to ten years. We then examine the hardware components in a simple computer, the components and purposes of a simple operating system, and the role computer networks are playing in modern-day applications of engineering computing. The latter includes introductions to client/server computing, the Internet, and the World Wide Web (WWW).

Chapter 2 introduces the principles upon which modern engineering software systems are built. Topics include the hardware-software life cycle, the economics of software development, top-down and bottom-up development strategies, software modularity, and information hiding. We conclude this chapter with an introduction to programming language concepts, including high- and low-level programming languages, compiled versus interpreted languages, scripting and markup languages, and so forth.

Chapter 1

Introduction to Engineering Computations

1.1 Applications of Computers in Engineering

During that past three decades, remarkable advances have occurred in the processing speed of computers, the capacity of computers to store, manipulate and present large quantities of data and information, and the ability of computers to communicate with other computers over networks. Evidence of these advances can be found in present-day engineering offices where computers are used in at least four broad capacities:

1. For **storage and manipulation of data and information**. Modern databases can store and manipulate a variety of data and information, including commercial off-the-shelf products, materials, and services; experimental data; the results of a numerical computations; models of designs, design documents and drawings; Geographic Information Systems (GIS) imagery; and so forth.
2. For **communication over computer networks**. Networking tools and technologies allow for the exchange of data and information over networks, and for computers to jointly contribute to the solution of large engineering analyses. Perhaps the greatest use of computer networks is for communication via E-mail.
3. For **desktop publishing**. Word processing packages such as LaTeX and Microsoft Word, and picture editors such as Corel Draw and Photoshop enhance an engineer's ability to write and edit publications.
4. For **numerical and symbolic computations**. Engineering analysis programs (e.g., programs for control systems and finite element analysis; MATLAB and Mathematica) are needed for the solution of engineering problems. The majority of engineers use commercial software for numerical and symbolic calculations, requiring preparation/programming of input files while some engineers will write their own software.

From a business point of view, the most useful application programs will directly improve the performance and reliability, productivity, and economic competitiveness of engineering systems development. The participating application programs should be fast and accurate, flexible, reliable, and of course, easy to use. And they should work together. A good example of the last requirement can be found in modern-day computer-aided design (CAD) systems where engineering analysis programs are integrated with project management tools, databases of project requirements, organizational resources, and commercial off-the-shelf products, materials, and services.

An unfortunate problem caused by these advances is the gap many engineers are finding between their knowledge of these technologies and the opportunities they afford. Solutions to this problem are complicated by the large number of activities in which engineers participate and the inability of many present-day engineering application programs to operate across a variety of hardware platforms and operating systems. Keeping up-to-date with computational technologies is really a lifelong endeavor because some of the application tools and computer programming languages we will use in five to ten years are only just being invented.

1.2 Recent Advances in Computing

A good way of beginning to understand where computers and programming languages might be headed in the near future, is to take a look at where they have come from in the recent past. We therefore begin this section with a little history.

Advances in Computing Since 1970

For more than a decade now, computers have been providing approximately 25% more power per dollar per year. Together with the aforementioned advances in technology and market driven forces, these changes have stimulated the exploration of many new ideas and paradigms. Figure 1.1 summarizes, for example, the major "modes of operation" and "key technologies for computing" versus decade for the past 30 to 35 years (this diagram has been adapted from an article in Scientific American [16]). The highlights are:

1970s : In the early to mid 1970s, mainframe computers were commonplace. They had a computational speed of 1 to 2 MIPS (millions of instructions per second) and were largely viewed as machines for research engineers and scientists. Compared to today's standards, computer memory was very expensive, and human-computer interaction was primitive. In fact, scientists and engineers interacted with a computer by sitting at a terminal and typing commands on a keyboard. The computer would respond by sending text to the terminal screen.

	BATCH	TIME-SHARING	DESKTOP	NETWORKS
Decade	1960s	1970s	1980s	1990s --
Technology	Medium-Scale Integration	Large-Scale Integration	Very-Large Scale Integration	Ultra-Scale Integration
Users	Experts	Specialists	Individuals	Groups.
Objective	Calculate	Access	Present	Communicate
Location	Computer Room	Terminal Room	Desktop	Mobile.
Connectivity	Peripherals	Terminals	Desktops	Laptops Palmtops
User Activity	Punch and Try (computer cards)	Remember and Type (interact)	See and Point (drive)	Ask and Tell (delegate)
Data	Alpha-numeric	Text Vector	Fonts Graphs	Sound Video
Languages	Cobol, FORTRAN	PL/1, Basic	Pascal, C SQL	C++, Java Perl, Tcl/Tk HTML, VRML

Figure 1.1. Paradigms of computing versus decade

Most software developers wrote computer programs dedicated to a specific task (e.g., finite element analysis; control systems package; an accounting or stock control system). Many of these packages were written in FORTRAN and run in "batch mode." The ease with which FORTRAN could be used to evaluate mathematical formulae – hence the name For-mula Tran-slation – was adequate for most engineering applications, and it can reasonably be argued that the choice of language was suitable for its time (unfortunately, some engineers and educators still think it is adequate).

1980s : In the 1980s desktop publishing systems were developed for individuals at work and home. Computers such as the Macintosh presented users with a screen containing window, scroll bar, and button/icon interface components, that could be defined and manipulated with a mouse

device by simply pointing and clicking icons on the graphical interface.

1990s : The use of computers in engineering is now at a point where mainframe computers are being replaced by high speed engineering workstations and modern Personal Computers (PCs) having bit-mapped graphics (a bit is short for binary digit, either a 1 or 0), global network connectivity (e.g., cellular communications; fiber optic; the World Wide Web), and multimedia (i.e., two or more of the following; graphics, voice, digital sound, video) [13]. Laptop computers now provide mobility, without compromising too many of the computational features available on PCs.

It is important to note that computers once viewed as a tool for computation alone, are now seen as an indispensable tool for computations and mobile communications. Access to this information has expanded from "experts" in the 1970s to "groups of individuals" today. A whole host of new programming languages, operating systems, and application programs have been written to support the new modes of functionality and day-to-day operation enabled by these technology advances. Consider, for example, an engineer who has access to a high speed PC with multimedia interfaces and global network connectivity, and who happens to be part of a geographically dispersed development team. The team members can use the Internet/E-mail for day-to-day communications, to share project information among team members, to conduct engineering analyses at remote sites, and to access information from databases on project components, materials, and services. Online assembly of joint ventures, online bidding of projects, and online verification of project performance against design standards may become commonplace in the near future.

1.3 Computer Hardware Concepts

The three main components in a computer are the hardware (including the computer networks connecting individual computers), the operating system, and the application programs that operate on individual computers and across computer networks. In our opinion, programmers should have a basic understanding of the hardware and operating system components in a simple computer because many computer programs are written to interact with a computer's input/ouput (I/O) devices and its operating system.

Hardware Components in a Simple Computer

Figure 1.2 is a schematic of the main hardware components in a typical (simplified) personal computer. Viewed from a high-level of abstraction, a computer is an assembly of processor, memory, and input/output (I/O) modules. A particular computer may have one or more modules of each type, with these modules being connected in some way to produce the main function of the computer.

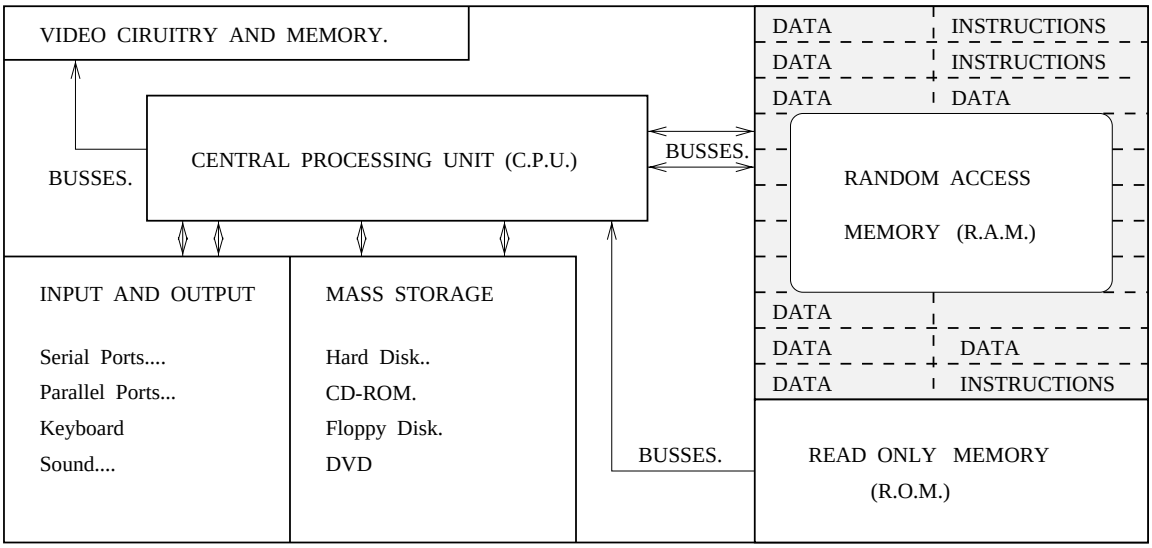


Figure 1.2. Hardware model of a personal computer

In Chapter 2 we will see that a computer program is nothing other than a list of instructions that can be followed mechanically by the computer. Machine instructions are expressed as binary numbers – that is, information represented as a sequence of zeros and ones. A computer can execute a program only if it is expressed in a machine language that can drive the mechanical operations of the computer.

The **central processing unit** (CPU) is the engine that controls the operations of the computer by executing instructions. In a conventional CPU, instructions are fetched from main memory, decoded, and executed one at a time. This process is entirely mechanical, and so if a computer program is to execute without error, the machine code instructions must be complete and unambiguous in their intent. Computers are designed so that they can be easily connected to external devices such as printers and keyboards. A second purpose of the CPU is to handle interrupts. When a device sends an interrupt signal to the CPU, it will halt what it is doing, take care of the device request, and then resume the original computation. The ability of a CPU to work on the solution of multiple tasks is called **multitasking**.

Within the CPU the **control unit** controls the fetch, decode, and execute cycles for instructions stored in memory. The **arithmetic control unit** carries out arithmetic and logical operations on words of data. A word of data is as long as the length of the hardware register in bits. The word length of a computer refers to the size of the unit of data in bits that a CPU can process at a time. Computers with a large word length process data faster than computers with a small word length.

Term	Abbreviation	Number of Bytes/Bits
Byte	B	1 Byte = 8 Bits
Kilobyte	KB	1024 = 2^10 Bytes
Megabyte	MB	1,048,576 = 2^20 Bytes
Gigabyte	GB	1,073,741,824 = 2^30 Bytes
Terabyte	TB	2^40 Bytes

Table 1.1. Terms used to quantify storage

The first processor had 4-bit word lengths. Currently, 32-bit PCs and engineering workstations are commonplace. 64-bit computers will dominate the marketplace in a few years.

The **internal storage unit** (not to be confused with the computer's primary memory) is fast internal memory that temporarily stores and manipulates data. It also contains busses (i.e., wires) for communication of the CPU with I/O devices, and mass storage known as random access memory (RAM) and read only memory (ROM).

The speed of a CPU is closely linked to the size of computer chips from which it is constructed. Broadly speaking, the more transistors a chip has, the more information it can process. State of the art manufacturing processes in 1997 allowed for chips having a miniaturization of 1/290-th of a human hair-width. The result, Pentium processor technology, has a maximum speed of 200 to 230 millions of cycles per second (i.e., 200 to 230 MHz). Pentium Pro chips (manufactured at miniaturization of 1/400-th of a human hair-width) will be able to run at speeds of up to 500 MHz. Predictions are that by the year 2001, next-generation Intel chips will contain 64 bit processing and operate at 1000 MHz.

The primary memory in a computer, called RAM (an acronym for **random access memory**, stores data and low-level program instructions as sequences of binary digits. Present day PCs and engineering workstations have 16 to 128 Megabytes (MB) of RAM (see Table 1.1 for a definition of terms) plus some **read only memory** (ROM). **Mass storage** is where programs, data, images and so forth, are permanently stored. For example, standard CD-ROMs store up to 650 MB, which is enough memory for approximately 70 minutes of audio/music. The new digital video disc (DVD) format, also sometimes called digital versatile disc, will be able to store up to 12 times as much data (8.5 GB using both sides of the disk). This is enough capacity to hold a full-length feature film with Dolby multichannel digital audio.

A **bus** is an electronic pathway in a digital computer that provides a communication path for data to flow between the CPU and its memory, and between the CPU and peripheral devices connected to the computer (e.g., monitor, printer, keyboard and mouse, network interface). Com-

puter systems are designed so that they can be easily expanded by adding new devices. When a new device is added to a computer, a software package known as a **device driver** must be installed so that the CPU can communicate with the new device. In Figure 1.2, busses provide multiline paths for rapid data transfer between different sections of the main computer board.

1.4 Operating System Concepts

An operating system is the set of programs that provides an interface between the computer hardware and the computer users. The operating system manages the sharing of a computer's resources and its memory contents, the low-level details of loading and executing programs, file storage and retrieval, assignment of processes to the screen and keyboard I/O devices, and communication with other computers. The term **operating system kernel** describes the set of programs in an operating system that implements the most primitive of that system's functions, including those for process management, memory management, basic I/O control, and security. Together these operating system features give the computer much of its functionality, including an environment for writing and running programs. The operating system and its components are also the first programs to run when the computer is turned on.

Two of the most popular operating systems are UNIX and WINDOWS 95 (tm), and fortunately, there are many similarities in their basic design and functionality. When you are just starting to learn how a new operating system works, understanding how the file system works is the most important (and potentially confusing) first step. In UNIX and WINDOWS 95, for example, you need to know how hierarchies of files are handled by the operating system and the operations that can be used to assemble, manipulate, and navigate a file system hierarchy. You may also need to learn how to "remote login" to another computer over a computer network. Interested readers should refer to Appendix 1 for a detailed discussion of these concepts for UNIX.

1.5 Computer Networking Concepts

A computer network is simply two or more computers connected together. Computer networks enable humans and computers to communicate by sending messages, and to share data and information resources by exchanging files. Two types of computer network are as follows:

1. **Local Area Network (LAN) :** A LAN is a network where computers are connected together directly. Usually the connection will be some type of cable.
2. **Wide Area Network (WAN) :** A WAN is simply a network of LANs connected together. The connections in present-day WANs are becoming a mixture of cable, fiber optic, and satellite communications.

Communication among LANs is handled by special-purpose computers called **routers**. Routers connect LANs to form a WAN. WANs can then be connected to form even larger WANs.

Client-Server Network Architectures

The sharing of information across computer networks is often implemented as two (or more) programs running on separate computers. One program, called the server, provides a particular resource. A second program, called the client, makes use of that resource. The server and client programs may be running on different machines located in separate rooms or even separate countries. Computer networks, where one server provides information to many clients, are said to have client/server architectures.

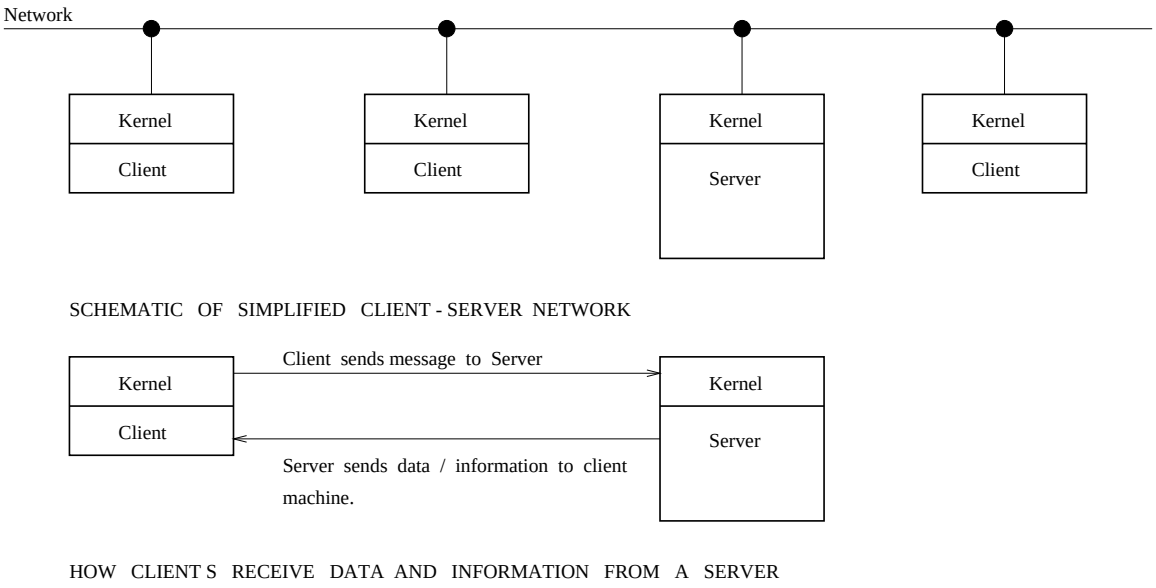


Figure 1.3. Model of communication in client-server system

Figure 1.3 shows, for example, a simplified network where one server machine is connected to three client machines. Any one of the client machines can send a message to the server machine requesting data, information, or even, access to certain operating system or application-package processes. The server machine will respond by sending the requested information/service to the client.

Client/server network architectures are increasing in popularity because of the advantages they afford. By localizing data, information, and operating system/application package processes

on a single server machine, and providing access to client machines on an as-needed basis, maintenance of operating system software and application program software is simplified considerably. Moreover, by moving much of the processing power from stand-alone client machines to powerful server machines, an opportunity exists to design client machines having "minimal" operating system functionality. These so-called "network computers" are expected to retail for considerably less than current PC computers.

The Internet

Millions of computers are now connected together in a massive worldwide network of computers called the Internet. The word **Internet** literally means "network of networks," and on any given day it connects roughly 20 million users in more than 50 countries – see, for example, the countries shaded in black in Figure 1.4. The Internet is rather unique in the sense that nobody owns, is in charge of, or pays for the entire cost of running the Internet.

Development of the Internet dates back to 1969 when the Defense Advanced Research Projects Agency arm of the Department of the Defense commissioned the construction of an experimental computer network. Work on the Advanced Research Projects Agency Network (ARPANET) centered around a perceived problem in the Department of Defense - how to keep U.S. military sites in communication in the event of a nuclear war. If just a few metropolitan areas were wiped out, communications could be severely disrupted. Two years later the network was connecting 15 nodes, all of them research centers. The Internet has since passed through the watchful eyes of many universities and research organizations, and grown in size to include millions of computer hosts.

Date	Number of Host Computers	Date	Number of Host Computers
Aug. 81	213	Jan. 89	80,000
May 82	235	Oct. 90	313,000
Aug. 83	562	Oct. 91	617,000
Oct. 84	1,024	Oct. 92	1,136,000
Oct. 85	1,961	Oct. 93	2,056,000
Nov. 86	5,089	Oct. 94	3,864,000
Dec. 87	28,174	Jul. 95	6,642,000
Jul. 88	33,000	Jul. 96	12,881,000

Table 1.2. Number of hosts on the Internet 8/81 to 7/96

Table 1.2 shows the number of hosts on the Internet versus time for August 1981 through

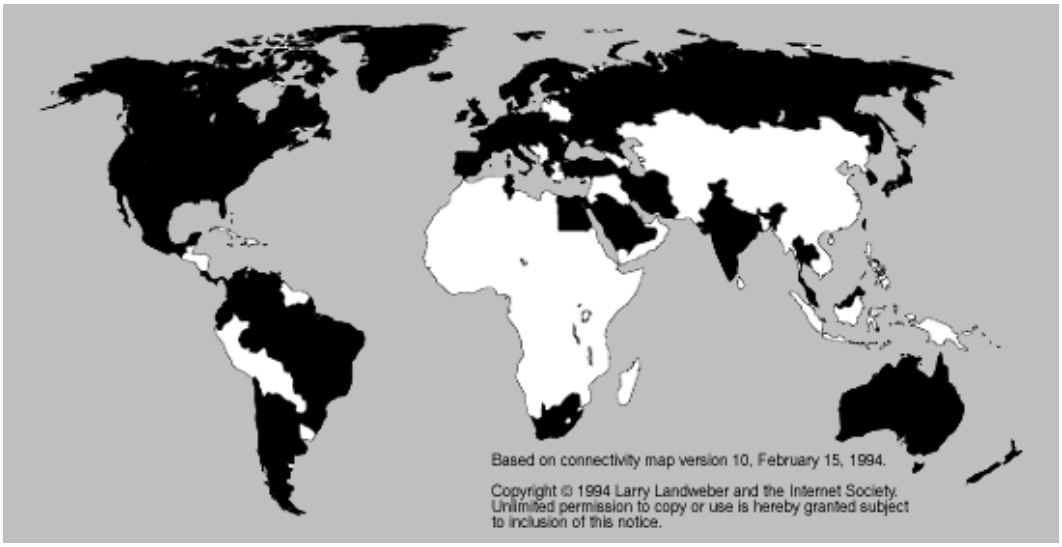


Figure 1.4. Countries having access to the Internet

July 1996. The estimate for the year 2000 is that 40 million people will be connected to the Internet.

Internet Access

The majority of present day home users access the Internet using a modem and phone lines – the upper speed, or bandwidth, at which information can be transmitted is 33.6 kilobits per second. A modest increase in bandwidth to 64 or 128 kilobits per second is possible with Integrated Services Digital Network (ISDN) technology. With 64 kilobits/sec bandwidth connection to the Internet, users can reserve part of the bandwidth channel for voice or fax calls. Most large research organizations and universities are connected to the Internet with one or more T1 lines having bandwidth 1.46 megabits/sec. In the near-future, significant increases in bandwidth (i.e., 10 megabits/sec). may be possible with cable modems.

Protocols for Internet Communication

Before two computers can exchange data and information over a network, they must agree on a specification, or protocol, for communication. Protocols cover issues such as how data will be formatted, conventions for control and coordination of information exchange, and handling of errors.

At the network level, the Internet Protocol (IP) specifies how data are to be physically

transmitted from one computer to another, and the Transmission Control Protocol (TCP) ensures that all the data sent using IP are received without error. Together these protocols are known as TCP/IP, and they form the foundation of many other high-level application-oriented protocols for sending packets of information (e.g., files, E-mail, web pages) across networks.

Internet Domain Names and Addresses

The Internet sends packets of information across a network using a model that is simply an electronic counterpart to the way letters are posted inside an envelope containing a delivery address and a return address. Perhaps the delivery address is the location of your Internet provider and the return address is the location of your home.

Every computer on the Internet has its own unique Internet address. You cannot send an E-mail message to someone, transfer a file via FTP, or access web pages located on a specific computer, unless you know his or her Internet address. In our "letter-and-envelope" analogy, there is the numerical form of an Internet address (it looks like xxx.xxx.xxx.xxx), and its vernacular counterpart. Although the numerical form is somewhat akin to a postal zip code, and an important part of the address delivery, people naturally prefer to work with addresses written in textual terms. Computer domain names follow a three-part format:

```
person's userid    @  domain name of computer(s)
       ^           ^           ^

       |           |           |
    Part 1   the "at" sign   Part 3
```

For example, the internet address `austinisr.umd.edu` has the person-id `austin` and the domain name `isr.umd.edu`. The numerical counterpart of `isr.umd.edu` is `128.8.111.4`. As these examples show, there is never blank space between components of the Internet address.

One important point to note is that user ids need not be unique. For instance, two people with the family name Austin can have the login name `austin` as long as they operate on separate domains.

```
austin@isr.umd.edu
austin@kiwi.berkeley.edu
```

However, the domain name of the computer(s) must be unique.

Computer domain names are composed of subdomain names that follow the nomenclature outlined in Tables 1.3 and 1.4. For example, in the computer address `isr.umd.edu`, the subdomain name `edu` tells us that the computer is located at an educational institution. The subdomain

Domain	Meaning	Domain	Meaning
com	Commercial	mil	Military
edu	Educational	net	Networking
gov	US Government	org	Nonprofit

Table 1.3. Organizational domain names

Domain	Meaning	Domain	Meaning
at	Austria	dk	Denmark
au	Australia	fr	France
br	Brazil	jp	Japan
ca	Canada	nz	New Zealand
de	Germany (Deutschland)	uk	United Kingdom

Table 1.4. Some geographical top-level domains

name `umd` stands for the University of Maryland. Finally, the subdomain name `isr` represents the collection of computers at the Institute for Systems Research at the University of Maryland.

Internet Services

Using the Internet means sitting at a computer and having access to a number of basic services, including:

1. **E-mail :** You can send or receive electronic messages from anyone on the Internet. Anything that can be stored in text file can be mailed. Facilities also exist for converting binary files (e.g., an executable computer program) into a format suitable for transmission via E-mail.
2. **Telnet :** The Telecommunications Network (Telnet) program allows users to remotely login to computers over a network. The computer may be in the next room, or perhaps, on another continent.
3. **Gopher :** Gopher allows a user to request information from an extensive list of gopher servers on the Internet.

4. **File Transfer :** The File Transfer Protocol (FTP) enables the copying of files from one computer to another. Anonymous FTP is a system where an organization makes certain files available for public distribution - you can access such a computer by logging in with the user-id "anonymous." Although no special password is required, it is customary to enter your E-mail address.
5. **Usenet :** An abbreviation for "user's network," Usenet is a collection of more than 5000 discussion groups centered around particular topics. Newsgroups exist for every topic imaginable, including of course, those dedicated to the Internet and its development.

More recently these basic services have been supplemented by the World Wide Web (WWW), allowing easy access to a wide array of media.

The World Wide Web

By providing millions of users in homes, schools, and industry with access to a wide array of media via an easy-to-use graphical user interface, the Web has captured the honorable distinction of being the most exciting part of the Internet.

Development of the Web began in March 1989, when Tim Berners-Lee of the European Particle Physics Laboratory (part of a larger organization known as CERN) proposed the project as a means of transporting research and ideas effectively throughout the CERN organization. The initial project proposal outlined a simple system of using networked hypertext (see the next section) to transmit documents and communicate among members in the high-energy physics community. The first graphical user interface to the Web was written by the Software Design Group at the National Center for Supercomputing Applications (NCSA). They named their graphical interface Mosaic. Mosaic, and the now famous Microsoft Internet Explorer and Netscape browsers, provide mouse-driven graphical user interfaces for displaying hypertext and hypermedia documents containing forms, paragraphs, lists, and tables in a variety of fonts and font styles. Graphics images can be linked to text and vice versa. Sound and video files can be pointed to by documents, and down-loaded over the Internet by simply clicking on the appropriate link.

Today's browsers are simply the beginning of where web development is headed. To help ensure that web browsers will be accepted as common business tools, software developers are busy integrating browser technology with office productivity suites (e.g., E-mail, word processors, graphics, spreadsheets, databases) and in the case of engineering firms, design and analysis software. Work is also currently underway to incorporate encryption technology and client authentication abilities into web servers, thereby allowing users to send and receive secure data. These features will ensure that sensitive data is kept private, a prerequisite for commercial transactions over the Internet.

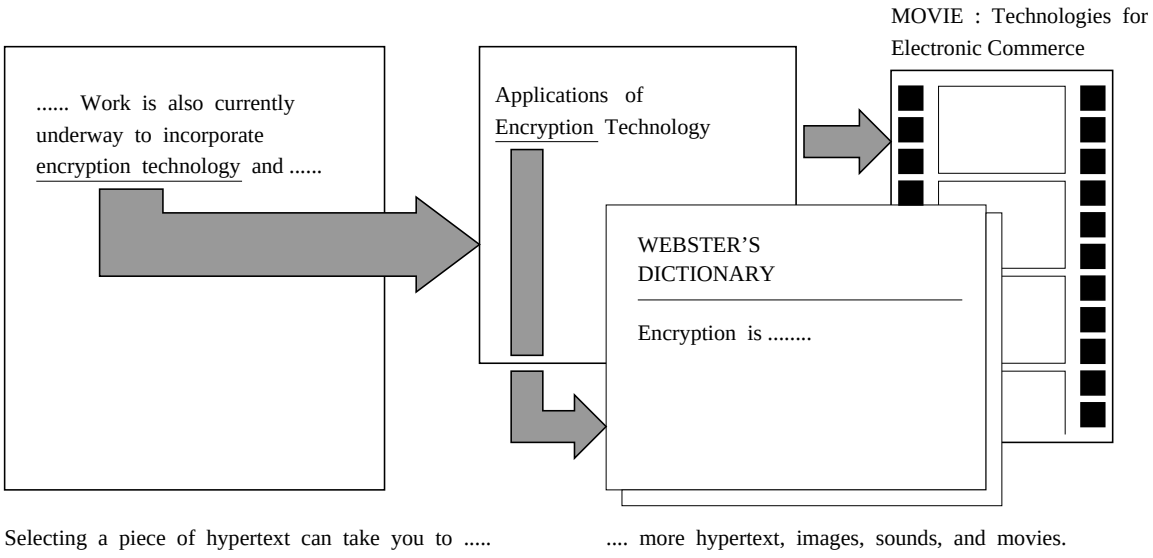


Figure 1.5. How hypertext works

Hypertext and Hypermedia

Hypertext is basically the same as regular text with one important exception – hypertext contains connections within the text to other documents. (See Figure 1.5.) Suppose, for instance, that you were able to somehow select (with a mouse or with your finger) the phrase "encryption technology" in the previous paragraph. In a hypertext system, you would then have one or more documents related to encryption technology appear before you - a textual description of the applications of encryption technology, or perhaps Webster's definition of encryption. These new texts would themselves have links and connections to other documents. Continually selecting text would take you on a free-associative tour of information. In this way, hypertext links, called hyperlinks, can create a complex virtual web of connections.

Hypermedia is hypertext with a difference - hypermedia documents contain links not only to other pieces of text, but also to other forms of media such as sounds, images, and movies. Images themselves can be selected to link to sounds or documents. Hypermedia simply combines hypertext and multimedia.

Suppose that you are a project manager in a large multinational engineering organization. Here are some examples of how web-based hypermedia might be used:

1. By clicking a mouse button on a part in an engineering drawing, you are able to see the pathway of project requirements leading to that part being incorporated in the design.

2. A web-based system might provide up-to-the-minute information on the status of a particular project. Clicking on the textual description of a project requirement could start an audio track describing an engineer's rational for the requirement.
3. By looking at an organization's floor plan, you are able to retrieve information about an office by simply touching a room. An inventory of office equipment, detailed office drawings, and information on the office occupant and their current projects might appear by clicking on the right, center, and left mouse buttons.
4. You are reading a research paper to understand why a new technology is needed by your organization. By selecting text in a research paper, you are able to view a movie of the technology being tested in a laboratory setting. Then by clicking on a mouse button you are able to download and execute an engineering analysis package demonstrating how the technology works for simple case-study problems.

In the implementation of these applications, we expect that the use of hypermedia will elevate a reader's ability to "understand" and "navigate" the concepts presented beyond what is likely to occur with a serial presentation (e.g., a regular book). Realizing this goal requires at the very least, good judgment, attention to detail in document design, and user testing.

How Does the Web Work?

Web software is designed around a distributed client/server architecture (see Figure 1.3). A web client, called a web browser if it is intended for interactive use, is a program that can send requests for documents to any web server. A web server is a program that, upon receipt of a request, sends the document requested (or an error message if appropriate) back to the requesting client. Because the task of document storage is left to the server and the task of document presentation is left to the client, each program can concentrate on those duties and progress independently of each other.

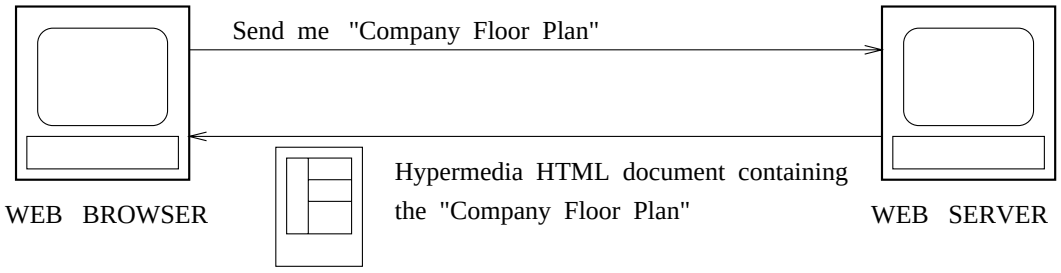


Figure 1.6. Web server and web client transactions

Figure 1.6 shows the sequence of transactions that would take place when the "Company Floor

Plan" is downloaded from a web server and displayed on a web browser. The sequence of events is as follows:

1. A user working at the web client selects a hyperlink in a piece of hypertext connecting to another document such as "Company Floor Plan."
2. The web client uses the address associated with that hyperlink to connect to the web server at a specified network address and asks for the document associated with "Company Floor Plan." (See the upper half of Figure 1.6.)
3. The web server responds by sending the text and any other media within that text (pictures, sounds, or movies) to the client. The web client renders the information for presentation on the user's web browser window. (This step is shown along the lower half of Figure 1.6.)

The language that web clients and servers use to communicate with each other is called the Hypertext Transfer Protocol (HTTP). All web clients and servers must be able to speak HTTP to send and receive hypermedia documents. For this reason, web servers are often called HTTP servers, and "World Wide Web" is often used to refer to the collective network of servers speaking HTTP as well as the global body of information available using the protocol.

Uniform Resource Locators

The Web employs what are called Uniform Resource Locators (URLs) to represent hypermedia links and links to network services within HTML documents. Almost any file or service on the Internet, including FTP, Gopher, and Telnet, can be represented with a URL. Table 1.5 contains some examples.

Uniform Resource Locator	Description
`http://www.ence.umd.edu/`	Connects to an HTTP server and retrieves an HTML file.
`ftp://rtfm.mit.edu/pub/usenet/`	Opens an FTP connection to the usenet frequently-asked questions stored at rtfm.mit.edu
`gopher://gopher.tc.umn.edu/`	Connects to a gopher menu at the University Minnesota home site, inventor of the Gopher.

Table 1.5. Some examples of URLs

A URL is always a single unbroken line of letters and numbers with no spaces. The first part of a URL (before the two slashes) specifies the method of access; `http` is perhaps the most common. Typically, the second part of the URL is the address of the computer where the data or service is located. Further parts may specify the names of files, the port to connect to, or the text to search for in a database.

Sites that run web servers are often named with a `www` at the beginning of the network address. For example, the Department of Civil Engineering web server at the University of Maryland has the URL:

```
http://www.ence.umd.edu/welcome.html
```

`welcome.html` is the name of the HTML file for our department home page. You can ask your web browser to connect to this web server by simply specifying the URL in the location window.

1.6 Hardware-Software Life Cycle

In engineering circles, advances in computer hardware and applications programs are driven by market competition and the need to design, analyze, manufacture, and control complex engineering systems. The complexity of an engineering system can be due to a number of factors including its size (i.e., a large number of interacting parts), nonlinear relationships between the input/output (I/O) parameters, incomplete information, enhanced performance specifications, and so forth. In any case, without the assistance of modern-day computer hardware and engineering applications programs, development of these systems would simply be intractable, if not impossible.

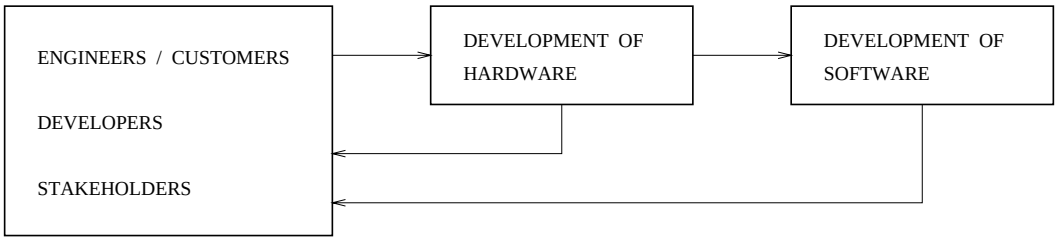


Figure 1.7. Hardware-software lifecycle

Figure 1.7 is a high-level view of components in the hardware-software lifecycle. Typically, it begins with a group of engineers/customers, developers and stakeholders (e.g., owners of a company) reaching consensus on the need for a new product and its pathway of development. Sometimes the results can be really impressive. Since the mid-1980s, for example, manufacturers of computer hardware have doubled the computational speed of their products every 12 to 18 months.

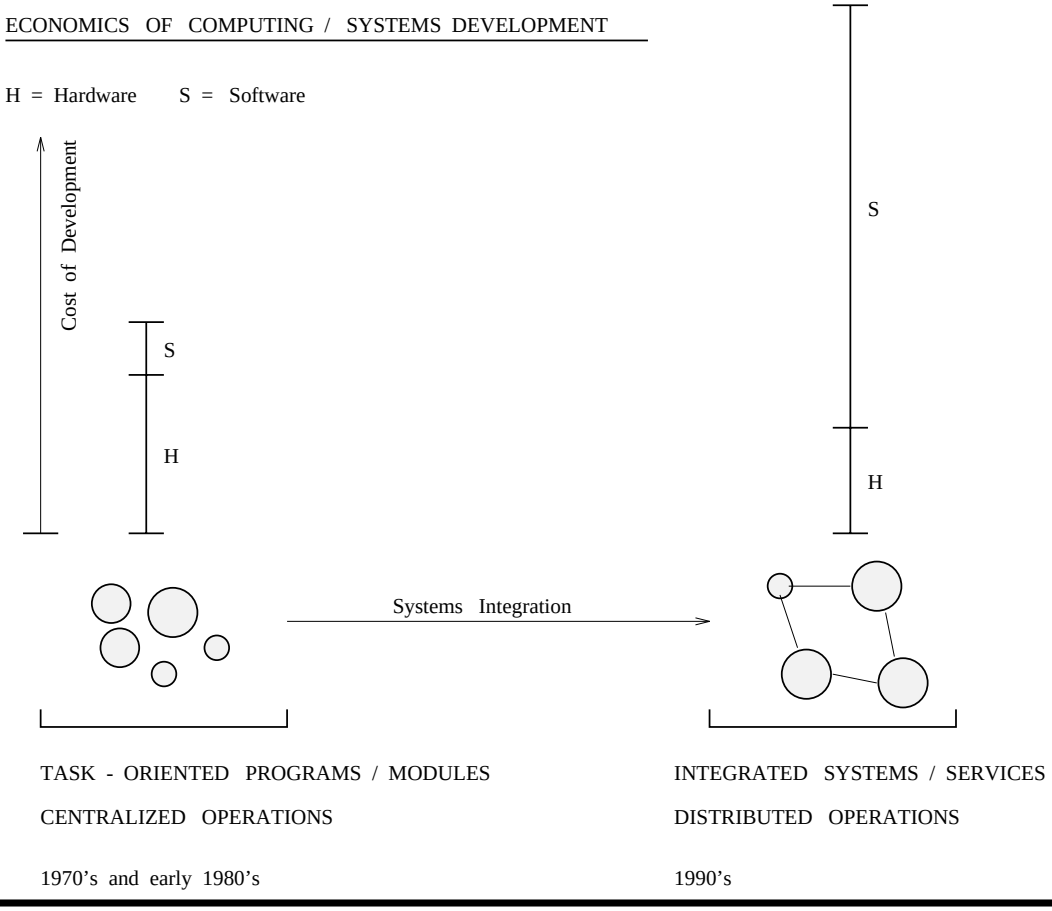


Figure 1.8. Economics of software development and integration.

The economic challenges and difficulty in following up on hardware advances with appropriate software developments are reflected in Figure 1.8. In the early 1970s, software consumed approximately 25% of total costs, and hardware 75% of total costs for development of data intensive systems. For the most part, computer systems were stand-alone, and software developers wrote computer programs dedicated to a specific task (e.g., finite element analysis, control systems package).

Currently, the development and maintenance of software typically consumes more than 80% of total project costs. This change in economics is the combined result of falling hardware costs, increased software development budgets, and a need to solve more difficult problems than in the past. Whereas one or two programmers might have written a complete program twenty years

ago, teams of programmers are now needed to write today's large software programs. Moreover, for organizations that have made large investments in software, there is great reluctance to discard software just because a new technology has come along. What management would like to see instead are the benefits of improved communications without having to reinvest in the basic application-specific software. The objectives of systems integration are to try to bring this situation under control, to ensure that the pathway forward maximizes return for the organization, and protects a company's past investments in software and hardware [7]. Systems integration is not a technical discipline in itself, but rather an approach to the management of organizations that recognizes the different ways its parts interact. Understanding an organization's structure and management practices is critically important because it is a prerequisite to computer automation.

If software developments for engineering/business applications are to have any chance of keeping up with advances in computer hardware and networking, future developments will need to pay close attention to software design and reuse of functions, libraries, modules, program architectures, and programming experience. A key component of the solution lies in the judicious choice of programming language(s). In addition to having the ability to compute and evaluate formulae, programming languages now need to handle and manipulate large quantities of data and information, and lend themselves to rapid development of interactive graphics applications in parallel and networked computing environments. Recently developed languages such as Perl, Tcl/Tk, and Java are expected to play a central role in making this happen [1, 11, 17].

1.7 Principles of Engineering Software Design

The discipline of software engineering is concerned with the design and implementation of computer programs that work, are correct, and are well written. Software engineering principles can be applied to small and large computer programs alike:

Small Computer Programs. Small computer programs are characterized by the data structures and algorithms they employ, design of the program architecture, and the computational efficiency of the program implementation.

Novice computer programmers tend to write programs that are small, composed of perhaps only a few hundred lines of source code. When you are learning to program in a new language, becoming familiar with its syntax, data types, and control structures are the most important things because without them you cannot transform a small-scale task into step-by-step programming instructions.

Large Computer Programs. In real-world engineering environments it is now commonplace for computer programs to be hundreds of thousands (even millions) of lines long. Many of them are so complex that even the best human minds cannot simultaneously comprehend all the details. It is

therefore vitally important that the design and implementation of large computer programs be based on established procedures for software development, including attention to program specification and design, organization, coding, testing, and maintenance of software [3, 10]. Careful planning of these activities is needed because:

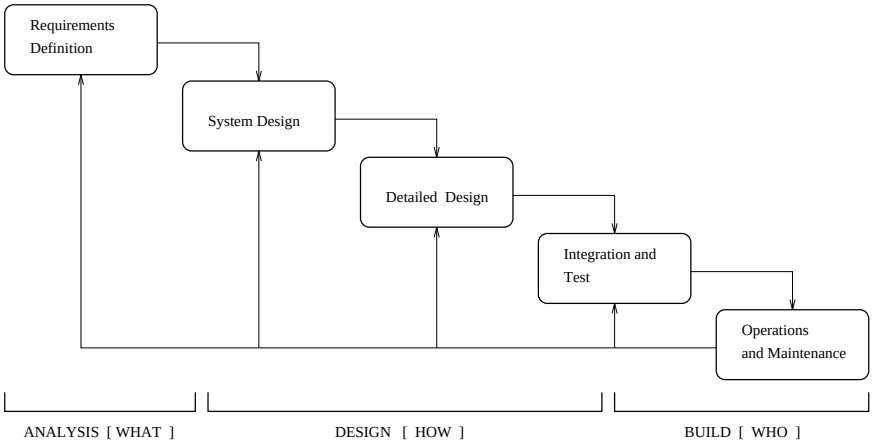
1. Large programs are most often written by programming teams. Team members must be able to understand one another's work. When the planning, design, and coding of a project takes several years, communication of work among employees is of utmost importance.
2. Large programs are often developed within the constraints of short time-to-delivery contract schedules. Programming teams may not have the luxury of starting again, even after a key design flaw is identified.
3. Well-organized programs are easier to debug. A little extra time spent planning the layout of a program may be saved many times over when it comes time to debug and upgrade the software. Indeed, some estimates place maintenance at 60 to 80% of the overall cost of a software project.
4. Large programs often evolve through a series of versions or updates. The programmer making updates is likely to be a person other than the original developer.

Models of Software Systems Development

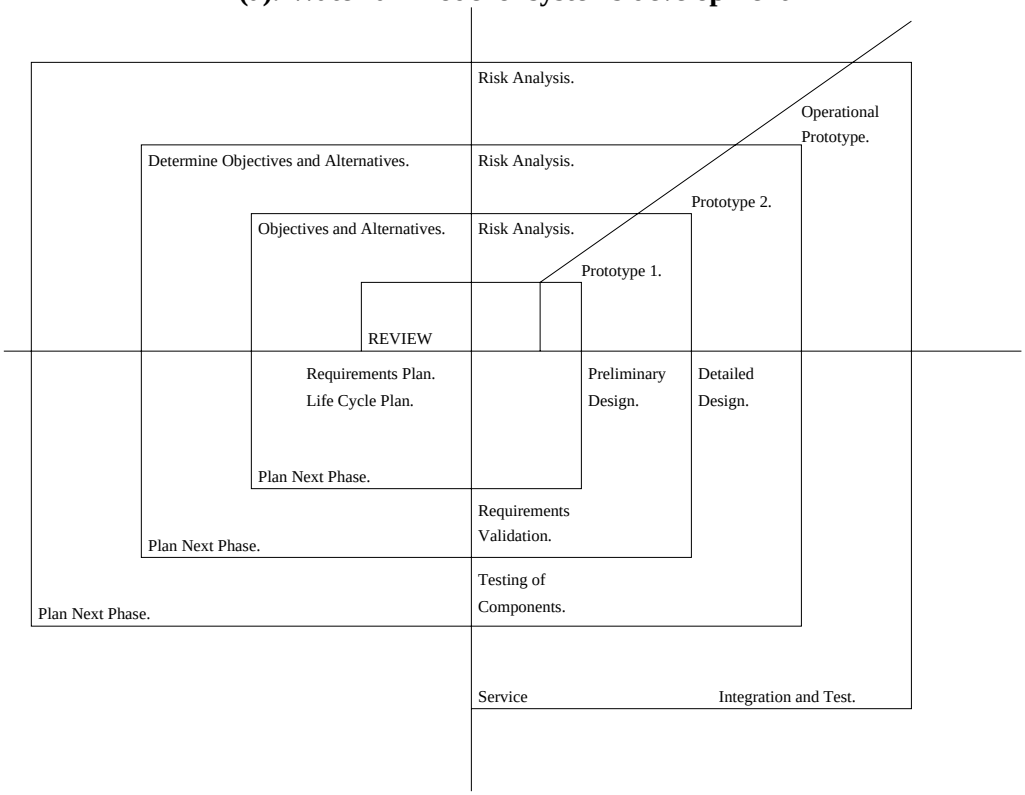
Models of software systems development are an important component of large computer program development because they provide the project participants with a framework for knowing what is expected, and when. In some cases, the model may also form the basis of legal agreements, with contract payments being tied to successful completion of a task identified in the model.

Figure 1.9 is a schematic of two models of software system development that are currently in use. The waterfall model [14] views the software development process as a sequence of stages that includes requirements specifications, design and testing, integration, and maintenance (we get to the details of each step in a moment). Each phase of sequential development is completed, via formal review, before the next phase begins. The waterfall model of development is appropriate when the problem and solution method are well understood.

The spiral model of development, shown in Figure 1.9b, is simply a sequence of waterfall models with risk analysis and control incorporated at regular stages in the project [2]. This risk-oriented approach to iterative enhancement recognizes that implementation options are not always clear at the beginning of a project. For example, a software project that is scheduled for development over a number of years may need technology that has not yet come to market. Maybe the technology will evolve as expected, and maybe it will not. The radial direction of Figure 1.9b corresponds to



(a). Waterfall model of systems development



(b). Spiral model of systems development

Figure 1.9. Models of development for software systems: (a) Waterfall, (b) Spiral

the cumulative cost incurred in the project, and the angular direction corresponds to progress made in completing each cycle of the spiral. Each cycle of development has the following phases:

1. Identify the design and development objectives for the cycle, as well as the alternatives that are possible to achieve the goals.
2. Evaluate different alternatives based on objectives and constraints. Where appropriate, identify uncertainties and risks. Risk means "something that can go wrong" as the consequence of incomplete information, or perhaps, as the result of human errors.
3. Develop strategies such as simulation, prototyping, and benchmarking for resolving uncertainties and risks.
4. Plan the next stage, allowing for any of the possible lifecycle models to be used.

Of course many variations on the waterfall and spiral models are possible. For example, some companies develop software incrementally, using a model that is essentially a chain of waterfall models with each link in the chain corresponding to development of a software release. Subsequent releases add capability to previous releases and fix bugs found in previous versions. The second observation tends to be a sore point with many involved in the software industry and leads them to make comments such as "never buy software that is younger than Version 3.0." Clearly they think that it takes at least two software versions just to find and eliminate insidious startup bugs.

Components of Software Systems Development

The key steps in the development of a large software project are as follows:

Requirements Specification. The objective of the requirements specification is to state the goals of the program as carefully as possible. The goals could include, for example, expected input (keyboard/files/IO board) and output. The requirements should also identify any known limitations, possible errors, and accuracy issues.

The importance of the requirements specification is partly due to the economics of project development at the beginning of the software lifecycle. As indicated in Figure 1.10, decisions made at the beginning of the software lifecycle are inexpensive to make, yet have the greatest commitment of funds. Without a clearly defined requirements specification there is a greatly increased chance of project failure.

Table 1.6 shows the results of a study by Hewlett-Packard to quantify the relative costs, in terms of money and/or time, of correcting design errors. From an economic point of view, the last thing a manufacturer wants is discovery of a fatal error in the engineering system by a customer !

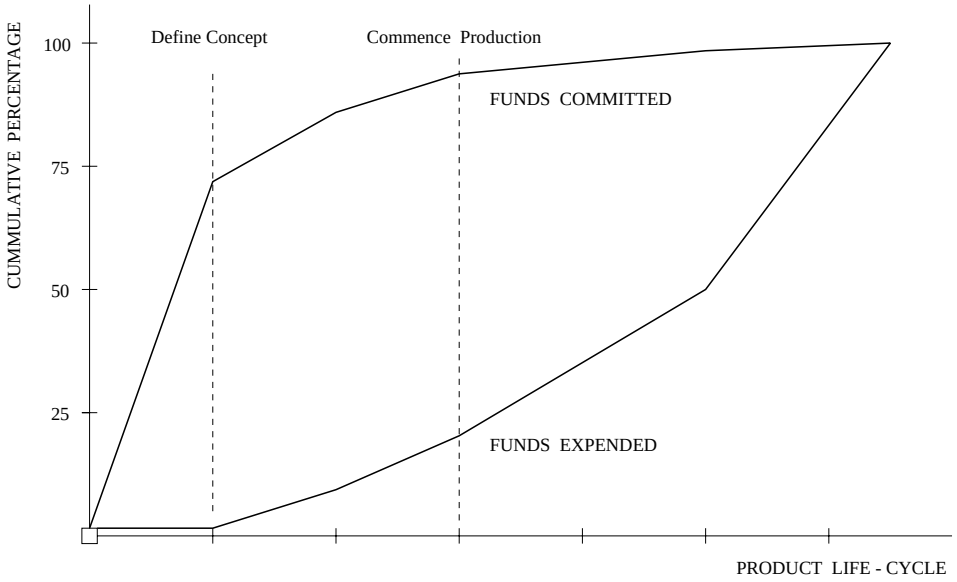


Figure 1.10. Funding commitments in product lifecycle.

Project Phase	Bug Description	Relative Cost
Design	Design team	1
Implementation	Programmers	10-20
Quality assurance	QA personnel	70-100
Shipment to customer	Customer	Very expensive

Table 1.6. Cost of correcting design errors

Analysis. Carefully analyzing a problem and its possible solutions is of utmost importance. Either you have to develop an algorithm yourself, or find one that is already available. Do not hesitate to look for established techniques; for example, an engineering problem might involve solving a set of linear equations using a standard numerical algorithm such as Gaussian Elimination. To develop algorithms from scratch when an already well-thought out and documented technique is available is a waste of time.

Design. The next most important phase of program development is to develop a viable design that you believe will work. Studies indicate that up to 50% of all errors in the software lifecycle are

made during the design phase of the software; the remaining errors tend to be programming and syntax errors (as described previously). Unfortunately, design errors are also the most expensive to repair since many appear only at run-time, and often after the software has been delivered to the customer.

Implementation. The implementation phase focuses on the specification, writing, and documentation of modules of code. When the details of specification are particularly complicated, pseudocode is often inserted as an intermediate step. **Pseudocode** is simply a high-level description of the code (or algorithm) to be implemented, where the semantics are less strict than the programming language itself.

Always document program source code so that it can be understood by you and others at a later date. Documentation includes recording details of your specific solution – how the analysis phase will transfer to the program, what language-specific constructs were used, what method of problem decomposition was used, and any important restrictions in your software design. An executable program is the result of the implementation phase.

Test and Verification. In many engineering industries, such as airplane flight control, airport traffic control, nuclear power plant control, medical instruments, and communications network control, reliability of software is the most sought after attribute. The software must execute as expected and without errors. Obtaining software that performs with a high degree of confidence requires a solid program design, a careful implementation, and a carefully designed suite of test problems that may illuminate software bugs and weaknesses. The identification of bugs and weaknesses is highly desirable, because this is the point at which fixing them is easy.

Maintenance. It is frequently stated that up to three quarters of a programmer's time is spent on maintaining existing software. Maintenance activities include (1) repair of coding (and design) errors, (2) adapting the software to changes in the computing environment (e.g., an update in the operating system), and (3) adapting the software to changes in the customers' requirements. Ask yourself how much code you have written in the past that is still in use (by you or others)? Can you easily understand programs that you wrote years ago? Would it be easy to change or modify your programs for a new purpose?

	Reuse rate(%)	Average lines per year	Average errors/1000 lines
Japan	80	12,447	1.96
USA	30	7,290	4.44

Table 1.7. Reuse of code in the US and Japan

Table 1.7 shows how important it is to write code that can be reused and easily modified

for a new purpose. If you are busy reinventing the wheel while other developers are reusing their software, eventually you will not be able to compete. These issues have to be taken into account before and during development of new programs. Once you have written a program without considering these issues, it is difficult and erroneouss to shoehorn or force the code into a new application. We must learn to take into account the need for future changes in programs and to design programs accordingly.

Modular Program Development

The goal of modular programming is to break a complex task or program into an ensemble of weakly coupled (independent) modules. Each module should be viewed as an independently managed resource, with access highly restrained. Programs consisting of well-designed modules are much simpler to design, write, and debug than equivalent programs that are poorly designed [5, 12]. Guidelines for the design of program modules are:

1. Psychological studies indicate that the average human can only simultaneously comprehend seven (plus or minus two) pieces of information. Hence, one module should have no more than seven subordinate modules.
2. There should be separation between the controller and worker modules.
3. Every module must perform a task appropriate to its place in the hierarchy.
4. Every module should only receive as much information as it needs to perform its function, and ideally, they should exchange as little information as possible.

Modules should enforce the principle of information hiding, namely, that all information about a module should be private to the module unless it is explicitly declared public [12]. Public data should be avoided because it exposes implementation details that make reuse unlikely and maintenance difficult. A key benefit of information hiding is the opportunity it affords for updating the internal details of a module (here we assume that the details of the public interface will not change) without affecting other modules in the system.

Coupling. Coupling measures the extent to which different modules in a system are interconnected with one other. See Figure 1.11. In design we should keep the interfaces as minimal and as simple as possible. Designing for minimal coupling among modules helps to ensure that errors occurring in one module will not propagate across the whole system.

Cohesion. Cohesion is a concept that describes how strongly the attributes and functions of a module are connected together. Ideally, a module should just perform one task. In design we should keep related functions together and unrelated functions apart.

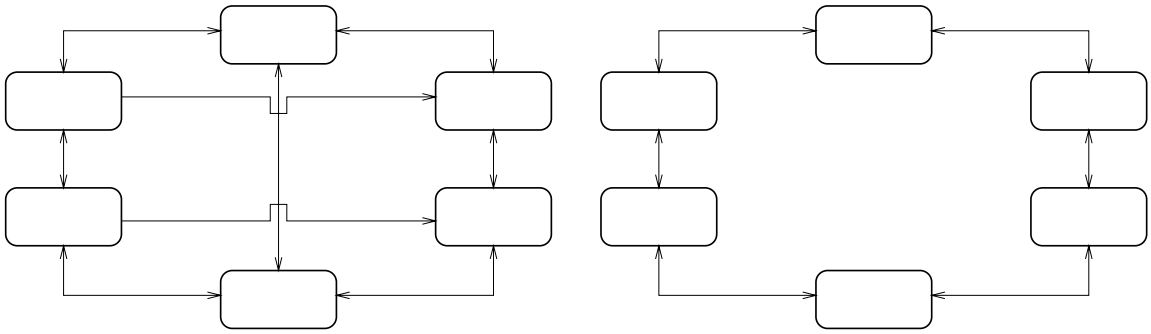


Figure 1.11. (a) highly coupled and (b) loosely coupled systems

The attributes of system coupling and cohesion work together. Generally speaking, modules with components that are well related will have the capability of plugging into loosely coupled systems. Modules should be designed within the framework of a well-defined language or grammar. If this principle is followed, then large programs may be divided into smaller modules (which may be compiled separately), and independent modules may be assembled together (i.e., bottom-up design). Modules designed within the framework of a language are easier to maintain and extend.

Abstraction

Programmers often apply techniques of abstraction to the development process, meaning that they concentrate on the essential features of one part/module of the computer program, and abstract from details of the computer program that are not immediately relevant to the current module. In the development of computer software, two types of abstraction are common:

1. **Procedural.** Software development is based on a stepwise refinement of the system's abstract function. This type of abstraction is most common in programming languages such as C, MATLAB, and FORTRAN.

2. **Data.** Software development is based on the system data and the operations that can be applied to the data. Implementations of data abstraction correspond to objects and the operations that can be applied to the objects. This type of abstraction is common in object-oriented programming languages such as C++ and Java.

Top-down and Bottom-up Software Design

Techniques of abstraction are often used in conjunction with a combination of top-down and bottom-up development strategies, as shown in Figure 1.12.

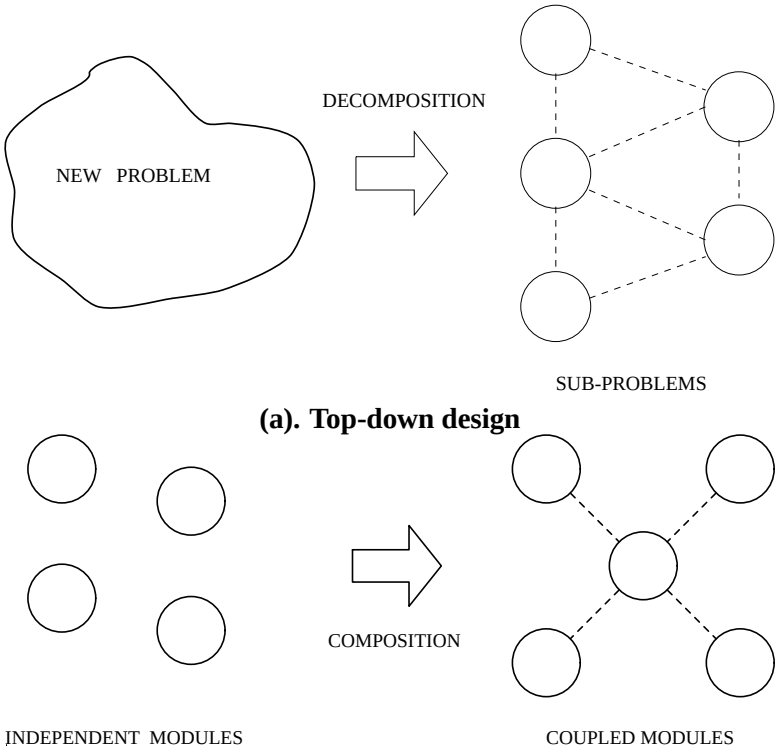


Figure 1.12. Schematics of top-down and bottom-up design

Top-down software development begins at the conceptual level and passes through three stages:

1. The **high-level design** establishes the important subsystems of the design, the purpose and intended behavior of each subsystem, and the relationship among subsystems. As already mentioned, a good design corresponds to subsystems that are as uncoupled as possible.
2. The **intermediate-level design** breaks subsystems into modules. Each module should have one well-defined purpose, hide its data from other modules, and have a minimal number of connections to other modules in the program [9].
3. The **low-level design** involves detailed specification of algorithms and data structures.

You should notice that top-down design delays detailed decisions about the program flow and data structures until they absolutely have to be made.

The strategy of bottom-up design starts with low-level procedures, modules, and subprogram library routines, and tries to combine them into higher-level entities. A key benefit of bottom-up design is its use of already implemented code. For example, numerical linear algebra packages are one area where libraries are routinely linked to C programs for finite element and numerical analysis, solution of differential equations, and solution of engineering control problems.

Top-down and bottom-up designs are extreme strategies for generating a hierarchical program structure and are often at odds with each other. Often the result of a top-down design is software modules that are of a one-time-only form. They are not amenable to reuse because they were designed as components without a preconceived vision of their future use. Similarly, generic modules suitable for bottom-up design may contain many features not needed to solve a specific task.

Software designers never embark on the construction of a new system without first considering available libraries/modules. Conversely, they never build software modules without a preconceived vision of their future use. A balance of the above-mentioned criteria is usually needed and desirable.

1.8 Computer Programming Language Concepts

Computer programs are composed of data and software instructions. The software instructions in a computer program tell the computer hardware how to execute a particular task by manipulating various types of data in a precise manner. Programming languages facilitate the development of these software instructions by providing constructs for organizing computations. The best programming languages enable the writing of good computer programs by being easy to read, understand, and modify.

High- and Low- Level Computer Languages

In the earliest days of computing computers were programmed by entering instructions and data into the computer in binary form. These so-called **machine languages** correspond to the instruction set of a particular computer hardware. They are entirely machine dependent (i.e., very low-level), and can be very tedious and error prone to program.

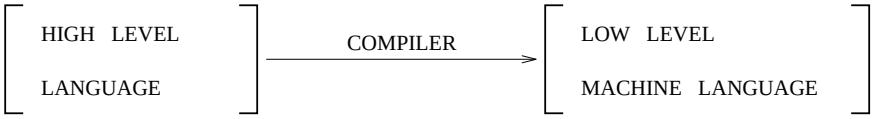
Programming in machine languages was quickly replaced by programming in symbolic **assembly languages**, which make the instructions easy to write and understand by using mnemonics for the machine instructions. For example, the sequence of symbols STO M R1 would store the

contents of register 1 in memory location **M**. Symbolic assembly languages are still low-level, and tend to be machine dependent. An "assembler" converts the assembly language into corresponding binary code and generates a binary program for execution.

Easy-to-read **high-level languages** have been around since the early 1950s. The term "high-level" means that many of the details in a program's development are handled automatically, thereby providing a pathway for programmers to write less code to get the same job done. For example, a number of high-level programming languages have been designed with the keywords `if` and `for` for the construction of simple branching and looping control structures. Once the program source code has been written, a compiler (see details below) will automatically generate the low-level machine instructions to implement the control structures. Whereas one line of assembly language code must be manually written for each low-level program instruction, high-level programming languages generate (on average) about five machine instructions for each line of code written. Because programmers write approximately the same number of lines of source code per day, irrespective of the development language, application programs can be implemented in a high-level language much faster (and more cheaply) than in an assembly language [4]. Examples of high-level programming languages are FORTRAN, C, C++, and Java. These languages are good for building software components from data structures and algorithms from scratch.

Compiled and Interpreted Programming Languages

Because computer hardware can only follow very low-level machine code instructions, which are difficult for humans to understand and manipulate, engineers usually write computer programs in a high-level programming language.



A compiler translates the computer program source code into machine code instructions. The compiler can be thought of as a type of machine; it only understands a limited range of commands. If the compiler does not receive proper input, it can only try to interpret the error, report what is wrong (i.e., error messages), and exit. C, C++, and FORTRAN are among the programming languages that are compiled. Good programmers have some idea into what the code they write compiles (assembly language, bytecode, or machine code), because it is the efficiency of compiled code that matters, not the original number of lines of source code.

In an **interpreted** computer program, high-level statements are read one by one, and translated and executed on the fly (i.e., as the program is running). Scripting languages such as Tcl/Tk and Perl are interpreted, as are application programs written in the MATLAB programming lan-

guage.

The Java programming language is both compiled and interpreted. High-level Java source code is compiled into a low-level bytecode, and then interpreted by a Java Virtual Machine. Do not worry if some of the details seem a little murky at this point – we fill in the details of how this happens in the Java tutorial.

Procedural and Object-Oriented Programming Languages

Programming languages such as FORTRAN, C, and MATLAB are procedural because they enable a complex problem to be decomposed into a hierarchy of functions. Each function has a well-defined set of input parameters and will return a well-defined set of output values.

Programming languages such as C++ and Java are object-oriented because they view complex problems as an assembly of data items together with sets of methods (i.e., functions) that can manipulate the data.

1.9 When to Program in MATLAB?

As indicated in Figure 1.1, the number of programming languages engineers are using is steadily increasing. This trend is due in part to the expanding range of tasks for which engineers are now using computers, and in part by the limited range of tasks current programming languages can handle well. Since computer languages are usually designed with the solution of a certain range of problems in mind, the selection of the right language for the job at hand is of utmost importance.

Use **MATLAB** if you need to solve a problem that can be conveniently represented by matrices, solved using operations from linear matrix algebra, and presented using relatively simple two- and three-dimensional graphics. Computing the solution to a family of linear equations, and representing, manipulating, and displaying engineering data are perhaps the two best examples of problems for which MATLAB is ideally suited.

Not only is the MATLAB programming language exceptionally straightforward to use (every data object is assumed to be an array), but the MATLAB program code will be much shorter and simpler than an equivalent implementation in C or FORTRAN or Java. MATLAB is therefore an ideal language for creating prototypes of software solutions to engineering problems, and using them to validate ideas and refine project specifications. Once these issues have been worked out, the MATLAB implementation can be replaced by a C or Java implementation that enhances performance, and allows for extra functionality – for example, a fully functional graphical user interface that perhaps communicates with other software package over the Internet.

Since the early 1990s the functionality of MATLAB has been expanded with the development of toolboxes containing functions dedicated to a specific area of mathematics or engineering. Toolboxes are now provided for statistics, signal processing, image processing, neural nets, various aspects of nonlinear and model predictive control, optimization, system identification, and partial differential equation computations. MATLAB 5.0 also comes with an Application Program Interface that allows MATLAB programs to communicate with C and FORTRAN programs, and vice versa, and to establish client/server relationships between MATLAB and other software program.

1.10 Review Questions

1. Have you been able to acquire an account and logon to a computer system at your school or company? Otherwise, do you have access to a PC that you can use to compile and run C, MATLAB, and Java programs while reading this book?
2. Do you understand the basic purpose of the computer components, such as the floppy disk, hard disk, memory (RAM and ROM), CPU, video display, keyboard, mouse, and so forth?
3. What are the main purposes of an operating system?
4. What is a computer network?
5. Briefly describe how a client/server system works. Can a computer act as both a client and a server at the same time?
6. Why were the Internet and the World Wide Web originally developed?
7. What technology has enabled web search engines, such as Yahoo and AltaVista, to index tens of millions of web pages in just a few years of time?
8. What is hypermedia?
9. What are the basic steps involved in the development of software? Why is the process broken down into several steps? Why would you use one language rather than another? Are they all the same?
10. Briefly explain the terms **abstraction**, **modularity**, **coupling**, **cohesion**, and **information hiding**?
11. What are the goals of top-down and bottom-up design?
12. What is the difference between procedural abstraction and data abstraction?
13. Why are present-day applications programs written in high-level programming languages?
14. Why does it typically take much longer to write an application program in assembly language than in a high-level programming language?
15. What do the acronyms HTML and VRML stand for? What is the relationship between Java and VRML?

16. For what types of problems is the MATLAB programming language suited? How has the functionality of MATLAB recently been expanded?
17. For what types of problems is the C programming language suited?
18. For what types of problems is the Java programming language suited?
19. Why does it sometimes make sense to develop a software package using more than one programming language?

1.11 Review Exercises

1.1 Acquire an account on a UNIX machine and login to your account. When you have logged in successfully, try to "navigate" the file system. For example, start by typing the `pwd` command. This command prints the working directory, or your current location in the file system. Now try `cd ..` which moves your current directory up to the parent directory. Now try `ls` to list the files in your directory. Now that you are in your parent directory, you can get back to your personal directory by typing `cd` (for change directory). By using `cd ..` and `cd` followed by a directory name, try to navigate and show the directory structure around your "home directory" on the UNIX system. Make a tree-type drawing of the structure.

1.2 Login to a UNIX machine and try the commands `rm`, `ls`, `cd`, `mkdir`, `rmdir` and `cat`. Experiment with the online UNIX manual – that is, type `man` (for manual) command, as in `man mkdir`.

Try specifying options to the commands on your computer system (e.g., `ls -l` in UNIX). Almost all operating system commands have options so that you can specialize the operation of the command. You will eventually learn how to do this with your C programs (it is convenient when you include command line options because it lets the user specify particular features of the program without having to ask every time the program is run). You can find the available options for an operating system command via `man` program on UNIX systems.

1.3 Use the operating system commands explained in Appendix 1 to build a simple directory structure under your UNIX home directory or under the root (C:) on your WINDOWS/NT system. Do not hesitate to create many directories; they are, after all, free!

1.4 Can you get some experience with a text editor such as "vi" or "emacs" on a computer? You will need to have experience with this because you will use an editor to enter your programs before they can be compiled and run. Most PCs come with a basic text editor, and there are many public domain editors you can get free of charge.

Part II

MATLAB Programming Tutorial

Introduction

MATLAB is a great programming language for solving problems that can be conveniently represented by matrices, that lend themselves to a solution with operations from linear matrix algebra and that can be presented using relatively simple two- and three-dimensional graphics. Computing the solution to a family of linear equations and representing, manipulating, and displaying engineering data are perhaps the two best examples of problems for which MATLAB is ideally suited.

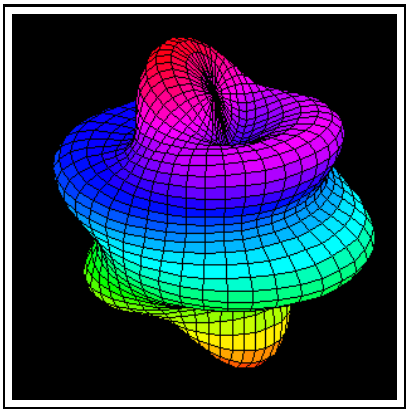
As we soon see, not only is the MATLAB programming language exceptionally straightforward (almost every data object is assumed to be an array), but also MATLAB program code will be much shorter and simpler than an equivalent implementation in C, FORTRAN, or Java. MATLAB is, therefore, an ideal language for creating prototypes of software solutions to engineering problems, and using them to validate ideas and refine project specifications. Once these issues have been worked out, the MATLAB implementation can be replaced by a C or Java implementation that enhances performance and allows for extra functionality (e.g., a fully functional graphical user interface that perhaps communicates with other software package over the Internet).

Chapter 13 is an introduction to programming in MATLAB Version 5.0 and includes all the basic concepts you need to know. Topics include variables and variable arithmetic, matrices and matrix arithmetic, control structures, built-in matrix functions, M-files, and so forth. Four engineering applications are solved at the end of Chapter 13. Where appropriate, pointers are given to equivalent or similar implementations of these problems in C and Java. A brief introduction to MATLAB graphics is contained in Chapter 14. Many other texts weave MATLAB graphics into the introductory sections of MATLAB programming. However, in an effort to keep the tutorial length short, we separate graphics from programming. Chapter 15 covers the solution of linear matrix equations in MATLAB. We demonstrate the power of MATLAB by working step by step through the formulation and solution of a variety of engineering applications involving families of matrix equations and, where applicable, MATLAB graphics.

Chapter 2

Introduction to MATLAB

2.1 Getting Started



The MATrix LABoratory program (MATLAB) was initially written with the objective of providing scientists and engineers with interactive access to the numerical computation libraries LINPACK and EISPACK. These libraries are carefully tested, high-quality programming packages for solving linear equations and eigenvalue problems [6, 15]. MATLAB enables scientists and engineers to use matrix-based techniques to solve problems without having to write programs in traditional languages such as C and FORTRAN. Currently, MATLAB is a commercial matrix laboratory package that operates as an interactive programming environment with graphical output. The MATLAB programming language is exceptionally straightforward to use since almost every data object is assumed to be an array. MATLAB is available for many different computer systems, including Macintosh, PC, and UNIX platforms.

The purposes of this tutorial are two-fold. In addition to helping you get started with MATLAB, we want you to see how MATLAB can be used in the solution of engineering problems. The latter objective is achieved through the presentation of a series of engineering application problems. Throughout this tutorial, we assume that you

1. Will read a few sections and then go to a UNIX workstation or PC/Macintosh to experiment with MATLAB
2. Are familiar with the operating system on your computer
3. Have access to supplementary material on matrices, matrix arithmetic/operations, and linear algebra

2.2 Professional and Student Versions of MATLAB

Professional and student editions of MATLAB Version 5 are available. The functionality of the student edition of MATLAB is limited in the following way; each matrix is limited to 16,384 elements, large enough to study problems having 128-by-128 matrices. From this point on, we assume that you have installed (follow the instructions accompanying the software disks) the student version of MATLAB on your computer.

Entering and Leaving MATLAB

The procedure for entering and leaving MATLAB on UNIX and Mac/PC platforms is as follows.

UNIX Platform. A MATLAB session may be entered by simply typing

```
prompt >> matlab
```

Here `prompt` $>>$ is the operating system prompt. A window should open and start MATLAB. When you run MATLAB under the window system, whether you start from the menu or a system prompt, a small MATLAB logo window will appear while the program is loading and disappear when MATLAB is ready to use. MATLAB will present the (double arrow) prompt

```
>>
```

You are now in MATLAB. From this point on, individual MATLAB commands may be given at the program prompt. They will be processed when you hit the "return" key. A MATLAB session may be terminated by simply typing

```
>> quit
```

or by typing `exit` at the MATLAB prompt.

Macintosh and PC Platforms. Click on the icon for MATLAB or STUDENT-MATLAB. Most of your interaction with MATLAB will take place in a Command window, where you will be presented with the prompt

```
EDU>>
```

The procedure for leaving MATLAB is the same as any other program operating on these platforms.

Both PC and Macintosh platforms come with a variety of programming and application development tools. For example, the PC platform supports an integrated M-file editor, a visual M-file editor, and performance profiler. The Macintosh platform comes with an M-file debugger and Workspace browser.

Online help

Online help is available from the MATLAB prompt, both generally (listing all available commands)

```
>> help
[a long list of help topics follows]
```

and specifically

```
>> help demo
[a help message on MATLAB's demo program follows].
```

The `version` command will tell you which version of MATLAB you are using. The `helpdesk` command

```
>> helpdesk
```

will initiate the execution of a World Wide Web (WWW) browser for MATLAB Online Reference Documentation. Check it out.

2.3 Variables and Variable Arithmetic

MATLAB supports a variety of data types for the desktop solution of engineering computations. Some problems will require the representation of `scalar` numbers or variables whose values are scalar numbers. Other types of computations require the use of complex numbers containing real and imaginary components. Moreover, solutions to a range of engineering problems may be expediently computed through the use of one-dimensional arrays and/or multidimensional matrices of scalars and complex numbers. This section explains how variables work in MATLAB.

The names and (data) types of MATLAB variables do not need to be declared because MATLAB does not distinguish between integer, real, and complex values. In fact, any variable can take integer, real, or complex values. Like most programming languages, variable names in MATLAB should be chosen so that they do not conflict with function or subroutine names, command names, or the names of certain values. However, name conflicts are bound to occur and we deal with this topic below in the subsection entitled "handling name conflicts."

Defining Variables

The equality sign is used to assign values to variables:

```
>> x = 3

  x =

        3

>>
```

As we soon see, variable names can be assigned to scalars, vectors (i.e., one-dimensional matrices), and matrices. Generally speaking, variable names will be a mixture of letters, digits, and the underscore character. The first character in a variable name must be a letter. Although variable names can be of any length, MATLAB Version 5. requires that they be unique within the first 31 characters. It is also important to bear in mind that variable names in MATLAB are case sensitive, meaning that variables `x` and `X` are distinct. Indeed, at this point, `X` is not even defined.

Table 2.1 shows some of the special built-in variable names and numbers used in MATLAB. The following script shows how you can learn their values from the MATLAB interpreter.

```
>> eps
  eps =
```

Variable Name	Meaning	Value
ans	Represents a value computed in an expression but not stored in a variable name.	
eps	Represents the floating point precision for the computer being used. It is the smallest amount with which two values can differ in the computer.	2.2204 e-16
"i" and "j"	Imaginary unit in a complex number.	sqrt(-1)
pi	Pi	3.1415926 ..
NaN	Represents "not a number." NaNs crop up in undefined expressions (e.g., division by zero) and in matrix elements where data is missing.	
inf	Infinity typically results from a division by zero or an arithmetic overflow.	(|><|)
clock	The current time is represented in a six element row vector containing year, month, day, hour, minute, and seconds.	
date	The current date is represented in a character string format.	
flops	Floating point operations count.	

Table 2.1. Special Variable Names and Numbers in MATLAB

```
          2.2204e-16
    >> pi
      ans =

          3.1416

    >> help pi

      PI     3.1415926535897....
             PI = 4*atan(1) = imag(log(-1)) = 3.1415926535897....
    >>
```

Points to note are:

1. The machine's round-off, the smallest distinguishable difference between two numbers as represented in MATLAB, is denoted `eps`.
2. The variable `ans` keeps track of the last output that was not assigned to another variable. You can use this property in a sequence of calculations where the output of one computation is

used in the following calculation.

Program output can be suppressed by simply appending a semicolon (;) to command lines, as in

```
>> x = 3;
>> x

  x =

        3

>>
```

More than one command may be entered on a single line if the commands are separated by commas or semicolons. To see how MATLAB handles these cases, try typing the command sequence

```
>> x = 3, y = 4; z = 5    % define and initialize variables x, y, and z.
```

In MATLAB, the % symbol indicates the beginning of a comment and, as such, the MATLAB interpreter will disregard the rest of the command line.

Arithmetic Expressions

Table 2.2 summarizes the meaning and order of evaluation of MATLAB operators in arithmetic expressions involving scalars and variables. With the exception of the left division operator, arithmetic expressions in MATLAB follow a fairly standard notation. The MATLAB commands

```
>> 2+3;           % Compute the sum "2" plus "3"
>> 3*4;           % Compute the product "3" times "4"
>> 4^2;           % Compute "4" raised to the power of "2"
```

are examples of basic arithmetic operations. The lower half of Table 2.2 shows the precedence and order of operator evaluation for arithmetic expressions. Operators having the highest precedence (i.e., a low precedence number) are evaluated first. So, for example, power operations are performed before division and multiplication, which are done before subtraction and addition. For example,

```
>> 2+3*4^2;
```

generates `ans = 50`. That is,

```
Meaning Of Arithmetic Operators
================================================================================
     Operator     Meaning                                          Example
================================================================================
            ^     Exponentiation of "a" raised to the              2^3 = 2*2*2 = 8
                  power of "b".
            *     Multiply "a" times "b".                          2*3 = 6
            /     Right division (a/b) of "a" and "b".             2/3 = 0.6667
            \     Left division (a\b) of "a" and "b".              2\3 = 3/2 = 1.5
            +     Addition of "a" and "b"                          2 + 3 =  5
            -     Subtraction of "a" and "b"                       2 - 3 = -1
================================================================================

Precedence Of Arithmetic Expressions
================================================================================
    Operators   Precedence                                              Comment
================================================================================
           ()            1        Innermost parentheses are evaluated first.
            ^            2           Exponentiation operations are evaluated
                                                          right to left.
          * /            3      Multiplication and right division operations
                                              are evaluated left to right.
            \            3      Left division operations are evaluated right
                                                             to left.
          + -            4           Addition and subtraction operations are
                                                   evaluated left to right.
================================================================================
```

Table 2.2. Meaning and Precedence of Arithmetic Operators

```
        2 + 3*4^2    <== exponent has the highest precedence.
    ==> 2 + 3*16     <== then multiplication operator.
    ==> 2 + 48       <== then addition operator.
    ==> 50
```

Arithmetic expressions involving operators of equal precedence are evaluated left to right. Of course, parentheses may be used to group terms or to make them more readable. For example,

```
>> (2 + 3*4^2)/2
```

generates `ans = 25`. That is,

```
        (2 + 3*4^2)/2   <== evaluate expression within
                            parentheses. Exponent has
```

```
                          highest precedence.
==> (2 + 3*16)/2     <== then multiplication operator.
==> (2 + 48)/2       <== then addition operator inside
                          parentheses.
==> (50)/2           <== then division operator
==> 25
```

Even though the addition operator has lower precedence than the divide operation, the order of evaluation can be easily altered with the use of parentheses. The second key use for parentheses is for function calls, that is, a function name followed by parentheses containing zero or more arguments. In this case, the function calls associated with a set of parentheses will be evaluated; innermost levels of parentheses first. For example, the step by step order of evaluation for

```
>> 4.0*sin( pi/4 + pi/4 )
```

is

```
    4*sin( pi/4 + pi/4 )      <== begin evaluation of left-hand
                                  side multiplication.
==> 4*sin( pi/4 + pi/4 )      <== evaluate expression within
                                  function parentheses, starting
                                  with leftmost division.
==> 4*sin( 0.7854 + pi/4 )    <== evaluate right-hand side
                                  division.
==> 4*sin( 0.7854 + 0.7854 ) <== evaluate sum.
==> 4*sin( 1.5708 )           <== sin(pi) function call.
==> 4*1.0                     <== finish evaluation of left-hand
                                  side multiplication.
==> 4.0
```

In this example, `sin(x)` is a function call to compute the sine of angle `x` measured in radians.

MATLAB has a `flops` function for counting the number of floating point operations needed to complete a MATLAB command or block of MATLAB commands. We can easily verify, for example, that five floating point operations are needed to evaluate the arithmetic expression

```
>> 4*sin( pi/4 + pi/4 );
>> flops

ans =

      5
>>
```

Numerical Precision of MATLAB Output

All arithmetic is done to double precision, which for 32-bit machines means to about 16 decimal digits of accuracy. MATLAB automatically prints integer values as integers and floating point numbers to four decimal digits of accuracy, with blank lines being inserted between textual lines of output. Exponential format is automatically used when the value of a number falls outside the range of numbers that can be printed using the default format.

MATLAB Command	Meaning	Example
format short	Default -- 4 decimal places	20.1235
format long	Output printed to 14 decimal places	20.12345678901234
format short e	Exponential format with 4 decimal places	2.0123e+01
format long e	Exponential format with 14 decimal places	2.012345678901234e+01

Table 2.3. Formatting Options for MATLAB Output

A summary of formatting options for the printing of the variable

```
>> a = 20.12345678901234
```

is shown in Table 2.3. Two useful commands are `format long`, which instructs MATLAB to display floating point numbers to 16 digits of accuracy, and `format compact`, which instructs MATLAB to abbreviate its output by removing all blank lines. Consider, for example, the script of code

```
>> x = 2.345        % define variable "x".

x =

    2.3450

>> format compact   % "compact" version of output.
>> format long      % "double precision" output.
>> x^30
ans =
     1.271409381050112e+11
>>
>> format short     % switch back to "default" output.
```

To save space in this tutorial, all our calculations from this point on are conducted with the `format compact` in place.

Built-In Mathematical Functions

MATLAB has a platter of built-in functions for mathematical and scientific computations (see Table 2.3). You should remember that the arguments to trigonometric functions are given in radians. Also bear in mind that MATLAB has functions to round floating point numbers to integers: They are `round`, `fix`, `ceil`, and `floor`.

Example. Verify that

```
sin(x)^2 + cos(x)^2 = 1.0
```

for some arbitrary values of `x`. The MATLAB code is

```
>> x = pi/3;
>> sin(x)^2 + cos(x)^2 - 1.0
ans =
     0
>>
```

Alternatively, we can write the arithmetic expression in terms of the `ans` variable storing the angle.

```
>> pi/4;
ans =
     0.7854
>> sin(ans)^2 + cos(ans)^2 - 1.0
ans =
     0
>>
```

Active Variables. When you want to know the active variables, you can use `who`. For example,

```
>> who
  Your variables are:
  ans      x
>>
```

The command `whos` gives a detailed listing of the active variables, together with information on the number of elements, and their size in bytes.

The command `clear` removes an item from the active variable list. For example, try typing `clear x`.

Trigonometric Functions

Function	Meaning	Example
sin (x)	Compute the sine of x (x is in radians).	sin(pi) = 0.0
cos (x)	Compute the cosine of x (x is in radians).	cos(pi) = 1.0
tan (x)	Compute the tangent of x (x is in radians).	tan(pi/4) = 1.0
asin (x)	Compute the arcsine of x. Complex results are obtained if abs(x) > 1 for some elements.	asin(pi/2) = 1.0
acos (x)	Compute the arccosine of x. Complex results are obtained if abs(x) > 1 for some elements.	acos(pi/2) = 0.0
atan (x)	Compute the arctangent of the elements of x.	atan([0.5 1.0]) = 0.4636 0.7854
atan2 (x,y)	Compute the four-quadrant inverse tangent of the real parts of x and y. This function returns an angle between -pi and pi.	atan2 (1,2) = 0.4636
exp (x)	Compute e^x where e is the base for natural logarithms.	exp(1.0) = 2.7183
log (x)	Compute the natural logarithm of x to the base e.	log(2.7183) = 1.0
log10 (x)	Compute the logarithm of x to the base 10.	log10(100.0) = 2.0

Mathematical Functions

Function	Meaning	Example
abs(x)	Return absolute value of x	abs(-3.5) = 3.0
ceil(x)	Round x toward positive infinity	ceil (-3.8) = -3
fix(x)	Round x toward the nearest integer towards zero.	fix (-3.8) = -3
floor(x)	Round x toward minus infinity.	floor (-3.8) = -4
rem(x,y)	Return the remainder of x/y. In C, this function is implemented as the modulo operator.	
round(x)	Round x to the nearest integer.	round(3.8) = 4
sign(x)	Return -1 if x is less than 0, 0 if x equals zero, and 1 if x is greater than zero.	sign(3) = 1
sqrt(x)	Compute the square root of x.	sqrt(3) = 1.7320...

Table 2.4. Common Trigonometric and Mathematical Functions

Program Input and Output

Throughout this tutorial, we employ a variety of techniques for input and output (I/O) of MATLAB variables and matrices. We see, for example, that variables and matrices can be defined explicitly, with built-in functions, and using data that are loaded into MATLAB from an external file. In this section we are concerned only with I/O of variables and introduce other features of I/O as needed.

MATLAB has two functions for the basic input of variables from the keyboard and for formatted output of variables.

Input of Variables from the Keyboard. MATLAB has a built-in function "input" that enables the value of variables to be specified at the keyboard. For example, the command

```
A = input('Please type the value of Coefficient A :');
```

will print the message

```
Please type the value of Coefficient A :
```

enclosed between single quotes on the computer screen. MATLAB will then wait for a numerical value to be typed at the keyboard, followed by the return key. In this particular case, the result of the numerical input will be assigned to variable `A`.

Formatted Output of Variables. MATLAB uses the function `fprintf` for formatted output of messages and numbers. The general syntax is as follows:

```
fprintf( format , matrices or variables )
```

The first argument contains the text and format specifications to be printed, and it is followed by zero or more matrices and variables. MATLAB's format argument operates in an almost identical manner to formatted output in the C programming language. For example, the specification `%f` is used for floating point numbers, and `%e` is used for exponential notation. The command

```
fprintf( 'Volume of sphere = %f\n' , 3.4 )
```

generates the output

```
>>  fprintf( 'Volume of sphere = %f\n' , 3.4 )
Volume of sphere = 3.400000
>>
```

The contents of the function argument between single quotes `'....'` is called a format string. The sequence of characters `%f` is an example of a conversion specification. Conversion specifications begin with a percent sign (%) and are followed by one or more formatting options for output left/right justification, numeric signs, size specification, precision specification, decimal points, and padded zeros. In this particular case, the `%f` simply tells `printf()` to output the contents of `3.4` as a floating point number. By default, the floating point number will be printed with six decimal places of precision.

We use the `'\n'` escape character to print a carriage return and advance the screen output (or cursor) to the beginning of a new-line. Omitting a newline character in `printf` statements will result in long stream of output on the same line – the output will either run off the right-hand side of the computer screen, or possibly jump to the next line automatically (actual behavior will depend on the computer system).

One of the neat features of `printf()` is its ability to accept a variable number of arguments. In practical terms one argument value must be added to the argument list of `printf()` for every field with a % specifier in the string constant. For example, the script of code

```
>> myPi = pi;
>> fprintf('myPi = %5.3f  2*myPi = %5.3f\n', myPi, 2*myPi )
myPi = 3.142  2*myPi = 6.283
>>
```

shows how a single call to `fprintf()` can print two floating point numbers.

Alternative Conversion Specifications. Our approximation of π could have been printed using a number of user-defined formatting options; for example, the left- and right-hand columns of the following table

```
Modified printf() statement             Output
=====================================  ====================
fprintf('myPi = %f \n', myPi );         myPi = 3.141593
fprintf('myPi = %14.7f \n', myPi );     myPi =      3.1415927
fprintf('myPi = %14.7e \n', myPi );     myPi =  3.1415927e+00
fprintf('myPi = %14.7E \n', myPi );     myPi =  3.1415927E+00
fprintf('myPi = %14.7g \n', myPi );     myPi =       3.141593
fprintf('myPi = %14.7G \n', myPi );     myPi =       3.141593
fprintf('myPi = %-14.7f \n', myPi );    myPi = 3.1415927
fprintf('myPi = %-14.7e \n', myPi );    myPi = 3.1415927e+00
=====================================  ====================
```

show modifications to the print statement shown above. The format specification `%14.7f` tells `printf()` to output the contents of `myPi` as a floating point number in a field 14 digits wide

(including the decimal point), with 7 digits of precision to the right-hand side of the decimal point. To ensure that the specification has enough width for the decimal point and the minus sign (if needed), the total number of digits in the output field should be at least 3 larger than the number of digits appearing after the decimal point.

The floating point specifications `e` and `E` tell `printf` to output the floating point number in exponential format. Unless the floating point number to be printed equals zero, the number before the letter `e` will represent a value between 1.0.. and 9.99... The two-digit part after the `e` represents an exponent value expressed as a signed decimal integer. The floating point number will be approximately equal to the first component multiplied by 10 raised to value of the exponent. The total number of digits in the output field should be at least 7 larger than the number of digits appearing after the decimal point. The floating point specifications `g` and `G` tell `printf` to select the better of the `f` or `e` formats. Although the rules for selecting the format are implementation dependent, as a general guideline, if the number to be printed falls within the range-of-conversion specification, use of the `f` format is likely. Otherwise, the floating point number will be printed in exponential format. In the seventh and eighth print statements, the output is left justified by inserting a minus (`-`) at the front of the conversion specification.

Saving and Restoring Variables. To save the value of the variable `x` to a plain text file named `x.value` use

```
>> save x.value x -ascii
```

To save all variables in a file named `mysession.mat` in reloadable format, use

```
>> save mysession
```

To restore the session, use `load mysession`. PC and Macintosh versions also come with a `File` menu for saving and retrieving data.

2.4 Matrices and Matrix Arithmetic

A matrix (or array) of order `m` by `n` is simply a set of numbers arranged in a rectangular block of `m` horizontal rows and `n` vertical columns. We say

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \tag{2.1}$$

is a matrix of size ($\mathtt{m} \times \mathtt{n}$). Sometimes we say "matrix A has dimension ($\mathtt{m} \times \mathtt{n}$)." The numbers that make up the array are called the `elements` of the matrix and, in MATLAB, no distinction is made between elements that are real numbers and complex numbers. In the double subscript notation a_{ij} for matrix element `a(i,j)`, the first subscript *i* denotes the row number, and the second subscript *j* denotes the column number.

By definition, a row vector is simply a (1×n) matrix and a column vector is a (m × 1) matrix. The ith element of a vector

$$V = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 & \cdots & v_n \end{bmatrix} \tag{2.2}$$

is simply denoted v_i. The MATLAB language has been designed to make the definition and manipulation of matrices and vectors as simple as possible.

Definition and Properties of Small Matrices

Matrices can be introduced into MATLAB by explicitly listing the elements through the use of built-in matrix functions and by M-files and external data files. We explain how matrices are defined in M-files and external files in a moment. For now, consider the statements

```
>> A = [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16 ];
```

and

```
>> A = [ 1  2  3  4
         5  6  7  8
         9 10 11 12
        13 14 15 16 ]
```

Both statements create a four-by-four matrix and assign its contents to a variable A. In the explicit declaration of matrices, a matrix is entered in row-major order, meaning all the first row, then all the second row, and so forth. Matrix rows are separated by a semicolon or a new line, and the elements within a row of a matrix may be separated by commas as well as a blank. The elements of a matrix are enclosed by brackets. Matrix elements that are floating point numbers are specified in the usual way (e.g., 3.1415926). Blank spaces must be avoided when listing a number in exponential form (e.g., 2.34e-9).

Row and column vectors are declared as matrices having either one row or column, respectively.

Accessing Matrix Elements. The matrix element located in the ith row and jth column of `A` is referred to in the usual way:

```
>> A(1,2), A(2,3)
ans =
       2
ans =
       7
>>
```

Similarly, the elements in a row or column vector may be accessed by simply typing the vector name followed by the index number inserted between brackets. For example,

```
>> V = [ 5 4 3 2 1 ];   % declare a row vector having 5 elements
>> V (2)                % access the second element in vector V
ans =
       4
>>
```

The elements of vectors and matrices may be easily modified with statements of the form

```
>> A(2,3) = 10;          % modify matrix element A (2,3)
>> V(2)   = 10;          % modify vector element V (2)
```

Size of a Matrix. The MATLAB function `size` returns a one-by-two matrix containing the number of rows and columns in a matrix. The script of code

```
>> size ( [ 1 2 3 4 5; 6 7 8 9 10; 11 12 13 14 15 ] )
ans =
     3     5
>>
```

shows how the `size` function returns the number of rows and columns in a 3-by-5 matrix.

Equal Matrices. Two matrices `A` and `B` are `equal` if they have the same number of rows and columns, and all the corresponding elements are equal (i.e., $a_{ij} = b_{ij}$ for $i = 1 \cdots m$ and $j = 1 \cdots n$). In MATLAB, a copy of the matrix

```
>> A = [ 1 2 3; 4 5 6 ];
```

can be made by simply writing

```
>> B = A
B =

     1     2     3
     4     5     6
>>
```

Square Matrices. If a matrix has the same number of rows as columns (i.e., `m` = `n`), then we say that it is `square` and that the matrix is of `order n`. The group of elements $a_{11}, a_{22} \cdots a_{nn}$ are called the principal diagonal elements.

Empty Matrices. MATLAB allows for the definition of matrices where one or more dimensions of the matrix may be empty, that is, contain no matrix elements. For example, the statement

```
>> A = [];
```

defines an empty matrix `A`. The MATLAB function `isempty()` can be used to test whether a matrix is empty (see Table 2.5 for more details on calling this function).

Defining Matrices with Built-In MATLAB Functions. Table 2.5 shows that MATLAB has a variety of built-in functions for the definition of small matrices. The script of code

```
>> eye (3,4)
ans =
     1     0     0     0
     0     1     0     0
     0     0     1     0
>>
```

shows, for example, how a matrix of 3 rows and 4 columns with 1s along the matrix diagonal is generated by the function call `eye(3,4)`.

Diagonal Matrix. A square matrix `A` whose elements $a_{ij} = 0$ for all $i \neq j$ is called a `diagonal` matrix. We write `A = diag(` $a_{11}, a_{22}, a_{33}, \cdots, a_{nn}$`)`. A diagonal matrix whose elements on the principal diagonal are all 1 is called the `unit matrix` or `identity matrix`. This special matrix has notation I.

Let `X` be a row or column vector containing n elements. `diag(X)` is the (n×n) diagonal matrix with the elements of `X` placed along the diagonal. Consider, for example

```
>> X = [ 4 3 2 1 ];
>> diag(X)
ans =
```

Function	Meaning	Example
eye (n)	Returns a n-by-n identity matrix.	eye (3)
eye(m,n)	Returns a m-by-n matrix of ones along the matrix diagonal and zeros elsewhere.	eye (3,4)
zeros(n)	Returns a n-by-n matrix of zero elements.	zeros (3)
zeros(m,n)	Returns a m-by-n matrix of zero elements.	zeros (3,4)

Table 2.5. Defining Small Matrices with MATLAB Functions

```
     4     0     0     0
     0     3     0     0
     0     0     2     0
     0     0     0     1
>>
```

Conversely, if A is a square matrix then `diag(A)` is a vector containing the diagonal elements of A. Consider, for example

```
>> A = [ 1 2 3; 4 5 6; 7 8 9 ];
>> diag (A)
ans =

     1
     5
     9
>>
```

Lower and Upper Triangular Matrices. A lower triangular matrix L is one where a_{ij} = 0 for all entries above the diagonal. An upper triangular matrix U is one where a_{ij} = 0 for all entries below the diagonal. That is,

$$\mathtt{L} = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \quad \mathtt{U} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{mn} \end{bmatrix} \tag{2.3}$$

The MATLAB functions `triu` and `tril` extract the upper and lower sections of a matrix, respectively. For example, the script

```
>> U = triu(A)
U =
     1     2     3
     0     5     6
     0     0     9
>> L = tril(A)
L =
     1     0     0
     4     5     0
     7     8     9
>>
```

extracts the upper and lower triangular sections of matrix `A` defined in the previous section. The function calls `triu(A,k)` and `tril(A,k)` generate square matrices of values from `A` with zeros below/above the `k-th` diagonal. For example, the script of code

```
>> U1 = triu(A, 1);
>> U2 = triu(A,-1);
```

generates the matrices

$$\texttt{U1} = \begin{bmatrix} 0 & 2 & 3 \\ 0 & 0 & 6 \\ 0 & 0 & 0 \end{bmatrix} \text{ and } \texttt{U2} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 0 & 8 & 9 \end{bmatrix} \tag{2.4}$$

Building Matrices from Blocks. Large matrices can be assembled from smaller matrix blocks. For example, with matrix A in hand, we can enter the following commands:

```
>> C = [A; 10 11 12];     <== generates a (4x3) matrix
>> [A; A; A];             <== generates a (9x3) matrix
>> [A, A, A];             <== generates a (3x9) matrix
```

As with variables, use of a semicolon with matrices suppresses output. This feature can be especially useful when large matrices are being generated. If `A` is a 3-by-3 matrix, then

```
>> B = [ A, zeros(3,2); zeros(2,3), eye(2) ];
```

will build a certain 5-by-5 matrix. Try it.

Reading and Saving Datasets

Suppose that the array of numbers

```
 1    10.0
 2    15.0
 3    14.0

 .... data items removed from file ...

28     9.0
29     4.0
30     0.0
```

is stored in a file `rainfall.dat` and that each row of the file corresponds to the date and daily rainfall measurement (in `mm`) for one calendar month. The command

```
>> load rainfall.dat
```

will read the contents of `rainfall.dat` into an array called `rainfall`, having 30 rows and 2 columns. The command

```
>> save rainfall.dat rainfall -ascii
```

will save the contents of array `rainfall` in the data file `rainfall.dat`. The array elements will be written in ASCII using 8 digits of accuracy. The command

```
>> save
```

will save array `rainfall`, and all the other matrices and variables in the MATLAB workspace, in a nonreadable `mat` file called `matlab.mat`. The contents of `matlab.mat` can be reloaded into MATLAB at a later date by simply reissuing the `load` command. For an extensive list of `save` and `load` command options, see the online documentation.

Application of Mathematical Functions to Matrices

In your high school mathematics classes you probably learned that `sin(x)` is the sine of an angle `x` measured in radians. A typical MATLAB command could be

```
>> x = pi/4;
>> sin ( x )
ans =
      0.7071
>>
```

MATLAB makes an important departure from traditional mathematics in the way it deals with matrices and the mathematical and trigonometric functions listed in Table 2.3. Instead of only allowing for the computation of mathematical formulae on a single variable (as demonstrated in the previous example), MATLAB enables mathematical formulae to be computed on the entire contents of a matrix by writing only one line of code. So how does this work? When we write the statement

```
>> x = pi/4;
```

we naturally think of `x` as being a variable. However, MATLAB, treats `x` as if it were a matrix containing only one element. The same numerical result can be obtained by writing

```
>> sin ( [ pi/4 ] )
ans =
      0.7071
>>
```

If the dimensions of the input matrix are expanded to something like

```
>> x = [ 1 2 3 4 5; 6 7 8 9 10]
x =
     1     2     3     4     5
     6     7     8     9    10
>>
```

then the command

```
>> sin(x)
ans =
    0.8415    0.9093    0.1411   -0.7568   -0.9589
   -0.2794    0.6570    0.9894    0.4121   -0.5440
>>
```

generates a two-by-five matrix containing the sine computations for each of the matrix elements in `x`. In other words, MATLAB systematically walks through all the elements in `x` and computes the `sine` of the corresponding matrix element.

We will soon see that this feature of MATLAB applies to nearly all the functions listed in Table 2.3. The true benefit of this feature lies in the development of MATLAB software for the solution of problems requiring repetitive calculations. Because the need for looping constructs can be eliminated in many cases, the complexity of user-written code needed to implement a numerical algorithm in MATLAB can be significantly simpler than counterpart implementations in C and Java, for example.

Colon Notation

A central part of the MATLAB language syntax is the "colon operator," which produces a list. For example

```
>> -3:3
  ans =
      -3    -2    -1     0     1     2     3
>>
```

The default increment is by 1 but that can be changed. For example

```
>> x = -3 : .3 : 3
  x =
    Columns 1 through 7
     -3.0000   -2.7000   -2.4000   -2.1000   -1.8000   -1.5000   -1.2000
    Columns 8 through 14
     -0.9000   -0.6000   -0.3000         0    0.3000    0.6000    0.9000
    Columns 15 through 21
      1.2000    1.5000    1.8000    2.1000    2.4000    2.7000    3.0000
>>
```

In this particular case you may think of `x` as a list, a vector, or a matrix, which begins at -3 and whose entries increase by .3, until 3 is surpassed. Generally speaking, the colon operator can be used anywhere in MATLAB code where a generated list is appropriate. Consider, for example, the block of statements for generating a table of sines

```
>> x = [0.0:0.1:2.0]' ;
>> y = sin(x);
>> [x y]
```

The first command generates a column vector of elements ranging from zero to two in increments of 0.1. You should note that because `sin` operates entrywise, the second command generates a column vector `y` from the vector `x`. The third command takes the `x` and `y` column vectors and places them in a 21-by-2 matrix table. Go ahead, try it.

Colon notation can also be combined with the earlier method of constructing matrices. For example, the command

```
>>  A = [1:6 ; 2:7 ; 4:9];
```

generates a 3-by-6 matrix `A`.

Submatrices

Any matrix obtained by omitting some rows and columns from a given matrix `A` is called a "submatrix" of `A`. A very common use of the colon notation is to extract rows, or columns, as a sort of "wildcard" operator, which produces a default list. For example,

```
>> A(1:4,3)
```

is the column vector consisting of the first four entries of the third column of `A`. A colon by itself denotes an entire row or column. So, for example,

```
>> A(:,3)
```

is the third column of `A`, and `A(1:4,:)` is the first four rows of `A`. Arbitrary integral vectors can be used as subscripts. For example, the statement

```
>> A(:,[2 4])
```

generates a two column matrix containing columns 2 and 4 of matrix `A`. This subscripting scheme can be used on both sides of an assignment statement. The command

```
>> A(:,[2 4 5]) = B(:,1:3)
```

replaces columns 2,4, and 5 of matrix A with the first three columns of matrix B. Note that the entire altered matrix `A` is printed and assigned. Try it.

Matrix Arithmetic

The following matrix operations are available in MATLAB:

Operator	Description	Operator	Description
+	addition	'	transpose
-	subtraction	\	left division
*	multiplication	/	right division
^	power		

These matrix operations apply, of course, to scalars (1-by-1 matrices) as well. If the sizes of the matrices are incompatible for the matrix operation, an error message will result, except in the case of scalar-matrix operations (for addition, subtraction, and division as well as for multiplication) in which case each entry of the matrix is operated on by the scalar.

Matrix Transpose. The transpose of a $m \times n$ matrix A is the $n \times m$ matrix obtained by interchanging the rows and columns of A. For example, the matrix transpose of

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix} \text{ is } A^T = \begin{bmatrix} 1 & 5 \\ 2 & 6 \\ 3 & 7 \\ 4 & 8 \end{bmatrix} \tag{2.5}$$

The matrix transpose is denoted A^T. In MATLAB, the transpose of a matrix is computed by following the matrix name with the single quote [apostrophe]. For example,

```
>> A = [ 1 2 3 4; 5 6 7 8 ];
>> A'
ans =
     1     5
     2     6
     3     7
     4     8
>>
```

We say that a square matrix A is "symmetric" if $A = A^T$, and it is "skew-symmetric" if A = - A^T.

Matrix Addition and Subtraction. If A is a $(m \times n)$ matrix and B is a $(r \times p)$ matrix, then the matrix sum C = A + B is defined only when m = r and n = p (see Figure 2.1a). The matrix sum is a $(m \times n)$ matrix C whose elements are

$$c_{ij} = a_{ij} + b_{ij} \tag{2.6}$$

for $i = 1, 2, \cdots m$ and $j = 1, 2, \cdots n$. For example,

if

$$\mathtt{A} = \begin{bmatrix} 2 & 1 \\ 4 & 6 \end{bmatrix}$$

and

$$\mathtt{B} = \begin{bmatrix} 4 & 2 \\ 0 & 1 \end{bmatrix}$$

,

then

$$\mathtt{C} = \mathtt{A} + \mathtt{B} = \begin{bmatrix} 2 & 1 \\ 4 & 6 \end{bmatrix} + \begin{bmatrix} 4 & 2 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 6 & 3 \\ 4 & 7 \end{bmatrix}. \tag{2.7}$$

The MATLAB commands for this computation are

```
>> A = [ 2 1; 4 6 ];
>> B = [ 4 2; 0 1 ];
>> C = A + B
C =
     6     3
     4     7
>>
```

Matrix subtraction is identical to matrix addition, except that $c_{ij} = a_{ij} - b_{ij}$ for $i = 1, 2, \cdots m$ and $j = 1, 2, \cdots n$. The matrix addition and matrix subtraction operations require mn floating point operations (flops).

Dot Product. Let

$$X = \begin{bmatrix} x_1 & x_2 & x_3 & \cdots & x_n \end{bmatrix} \tag{2.8}$$

be a row vector containing n elements and

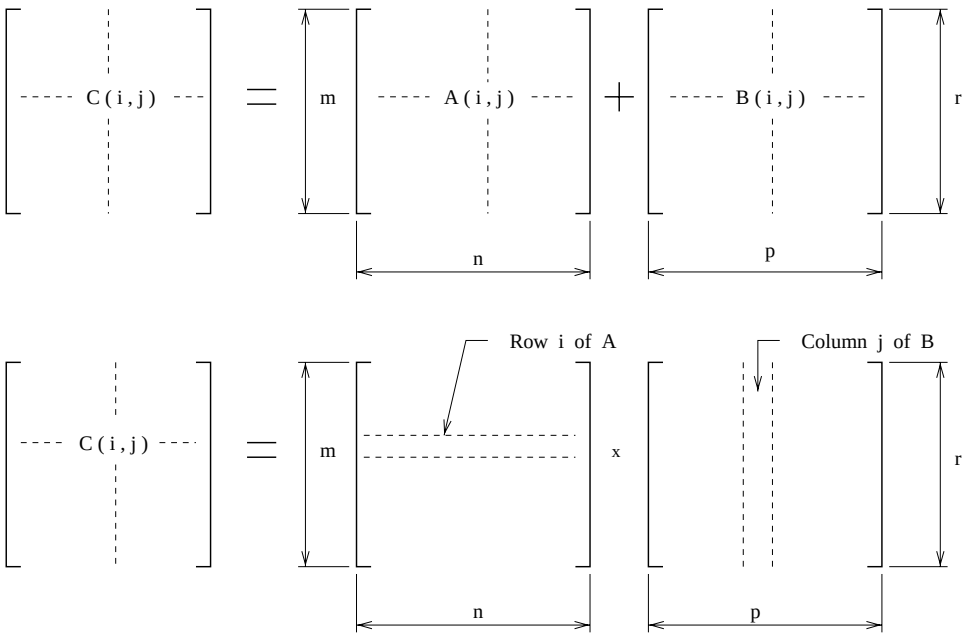


Figure 2.1. Schematics of (a) matrix addition and (b) matrix multiplication

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} \tag{2.9}$$

be a column vector containing the same number of elements. The dot product, also sometimes called the scalar product or inner product, is a special case of matrix multiplication (see the next section) and is defined by

$$X * Y = \sum_{i=1}^{n} x_i \cdot y_i \tag{2.10}$$

For example, the dot product of row vector `X = [ 1, 2, 3, 4, 5 ]`, with its matrix transpose, can be computed in MATLAB by simply typing

```
>> X = [ 1, 2, 3, 4, 5 ];
>> Y = X';
>> X*Y
   ans =
       55
>>
```

The results of this computation can be easily verified by hand

```
X*Y = [ 1 2 3 4 5 ]*[ 1 2 3 4 5 ]'
    = 1*1 + 2*2 + 3*3 + 4*4 + 5*5
    = 1 + 4 + 9 + 16 + 25
    = 55
```

The dot product operation on two vectors containing n elements requires n floating point operations (flops).

Matrix Multiplication. Let A and B be $(m \times n)$ and $(r \times p)$ matrices, respectively. The matrix product $A \cdot B$ is defined only when interior matrix dimensions are the same (i.e., $n = r$). The matrix product C = A · B is a $(m \times p)$ matrix whose elements are

$$c_{ij} = \sum_{k=1}^{n} a_{ik}\, b_{kj} \tag{2.11}$$

for $i = 1, 2, \cdots m$ and $j = 1, 2, \cdots n$. From a geometric point of view, c_{ij} is the dot product of the ith row of A with the jth column of B (see Figure 2.1b.). Assuming that matrices A and B are as defined in the previous section:

$$\mathtt{C} = \mathtt{A} \cdot \mathtt{B} = \begin{bmatrix} 2 & 1 \\ 4 & 6 \end{bmatrix} \cdot \begin{bmatrix} 4 & 2 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 \cdot 4 + 1 \cdot 0 & 2 \cdot 2 + 1 \cdot 1 \\ 4 \cdot 4 + 6 \cdot 0 & 4 \cdot 2 + 6 \cdot 1 \end{bmatrix} = \begin{bmatrix} 8 & 5 \\ 16 & 14 \end{bmatrix}. \tag{2.12}$$

The MATLAB commands for this operation are

```
>> A = [ 2 1; 4 6 ];
>> B = [ 4 2; 0 1 ];
>> C = A*B
C =
     8     5
    16    14
>>
```

Computation of the matrix product requires $2mnp$ flops. When the dimensions of the matrices are approximately equal, then computational work is $2n^3$ flops.

Example. Because the number of rows equals the number of columns in square matrices, square matrices can always be multiplied by themselves. The triple matrix product of

```
>> A = [ 2 1; 4 6 ];
```

can be evaluated by writing

```
>> A*A*A
ans =
    48    56
   224   272
>>
```

or by simply writing

```
>> A^3
ans =
    48    56
   224   272
>>
```

Example. A magic square is a square matrix that has equal sums along all its rows and columns. For example,

```
>> magic(4)
ans =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
>> magic(4)
```

In this particular case, the elements of each row and column sum to 34. Now we can use matrix multiplication to check the "magic" property of magic squares:

```
>> A = magic(4);
>> b = ones(4,1);
>> A*b;              <== (4x1) matrix containing row sums.
>> v = ones(1,4);
>> v*A;              <== (1x4) matrix containing column sums.
```

Scalar Multiplication of Matrices. Scalars multiply matrices on an element-by-element basis. For example, with matrix A in place

```
>> 2*A               % multiply elements of matrix "A" by "2".
ans =
     4     2
     8    12
>> A/4               % right division of "A" by "4".
ans =
    0.5000    0.2500
    1.0000    1.5000
>> 4\A               % left division of "A" by "4".
ans =
    0.5000    0.2500
    1.0000    1.5000
>>
```

Matrix Division. Left and right division of matrices is a generalization of left and right division in variable arithmetic. Let A and B be two matrices. Right division of two matrices, written

```
>> A/B
```

corresponds to the solution of the linear matrix equations `A.X = B`. Similarly, left division of two matrices is written

```
>> A\B
```

and corresponds to the solution of the linear matrix equations `X.A = B`. We discuss the nature of these solutions in more detail in Chapter 15.

Matrix Inverse. The inverse of a square matrix A, denoted A^{-1}, is given by the solution to the matrix equations

$$A \cdot A^{-1} = A^{-1} \cdot A = I.$$

In MATLAB, A^{-1} can be computed with `inv(A)`.

Arithmetic on Submatrices. MATLAB provides for the computation of arithmetic operations on submatrices. Suppose that matrix A is

```
>> A = [  1   2   3   4   5   6;
          7   8   9  10  11  12;
         13  14  15  16  17  18;
         19  20  21  22  23  24  ];
>>
```

Columns 2 and 4 of A can be multiplied on the right by the 2-by-2 matrix `[1 2;3 4];`

```
>> A(:,[2,4]) = A(:,[2,4])*[1 2;3 4]
A =
     1    14     3    20     5     6
     7    38     9    56    11    12
    13    62    15    92    17    18
    19    86    21   128    23    24
>>
```

Matrix A is altered and printed.

Matrix Element-Level Operations

MATLAB has a special convention whereby a dot positioned in front of the arithmetic operations listed in the upper half of Table 2.2 will force entry-by-entry computation of the matrix operation. By definition, the matrix addition and matrix subtraction operations are already computed on an element-by-element basis. Matrix operations such as multiplication and left and right matrix division are not. The best way of seeing how this works is to simply walk through a couple of examples.

Example. We start with

```
>> [1,2,3,4].*[1,2,3,4]
  ans =
       1     4     9    16
>>
```

Here the expression `[1,2,3,4].*[1,2,3,4]` evaluates to

```
[1,2,3,4].*[1,2,3,4] ==> [ 1*1 ,2*2 ,3*3 ,4*4 ]
                     ==> [   1 ,  4 ,  9 , 16 ].
```

The same result can be obtained with `[1,2,3,4].^2`. Similarly, the expression

```
>> [1,2,3,4]./[1,2,3,4]
ans =
     1     1     1     1
>>
```

evaluates to a vector of 1s.

Example. Suppose that we wish to verify $\sin(x)^2 + \cos(x)^2 = 1$ for a variety of argument values x. The block of code

```
>> x = [ -pi : pi/4 : pi ];
```

creates a one-by-nine array x containing values that range from $-\pi$ to π in increments of $\pi/4$. The command

```
>>  sin(x).^2 + cos(x).^2
```

generates the output

```
ans =
     1     1     1     1     1     1     1     1     1
>>
```

The order of evaluation is as follows. First, one-by-nine matrices are computed for $\sin(x)$ and $\cos(x)$. Element-level operations square each element in $\sin(x)$ and then each element in $\cos(x)$. The output is simply the sum of the latter two matrices.

2.5 Control Structures

Control of flow in the MATLAB programming language is achieved with logical expressions, branching constructs for the selection of a solution procedure pathway, and a variety of looping constructs for the efficient computation of repetitious operations.

Logical Expressions

In MATLAB, a logical expression involves the use of relational and logical operands for the comparison of variables and matrices of the same size. Table 2.5 summarizes the six relational operators and three logical operators in the MATLAB programming language.

```
Relational Operators
=======================================
Operator      Description
=======================================
       <      Less than
       >      Greater than
      <=      Less than or equal
      >=      Greater than or equal
      ==      Equal
      ~=      Not equal
=======================================

Logical Operators
=======================================
Operator      Description      Precedence
=======================================
       ~      Not                      1
       &      And                      2
       |      Or                       3
=======================================
```

Table 2.6. Summary of Relational and Logical Operators

Relational operators allow for the comparison of variables, and matrices of the same size. In the case of variable comparisons, the outcome will be 1 if the comparison is true, and 0 otherwise. Indeed in MATLAB, as in other languages such as C and Java, true is represented by a nonzero integer (usually one) and false is represented by zero. To see how this works in practice, we look at the script of code:

```
>> 3 < 5
```

Function	Description
any(x)	For vector arguments x, any(x) returns 1 if any of the elements of x are nonzero. Otherwise, any(x) returns 0. When x is a matrix argument, any(x) operates on the columns of x, returning a row vectors on 1s and 0s.
all(x)	For vector arguments x, all(x) returns 1 if all the elements of the vector are nonzero. Otherwise it returns 0. For matrix arguments x, all(x) operates on the columns of x, returning a row vector of 1s and 0s.
find (x)	This function finds the indices of the nonzero elements of x. The function argument can be combined with logical and relational expressions.
isnan (x)	isnan(x) returns 1s where the elements of x are NaNs and 0s where they are not. Recall that NaN means "Not a Number".
finite (x)	finite(x) returns 1s where the elements of x are finite and 0s where they are not.
isempty(x)	In MATLAB, an empty matrix has a zero size in at least one dimension. isempty(x) returns 1 if x is an empty matrix and 0 otherwise.

Table 2.7. Logical Functions in MATLAB

```
   ans =
        1
 >> a = 3 == 5
    a =
        0
 >>
```

In the first example, 3 is less than 5, so the result of the relational expression is true (i.e., 1). The second example contains both an assignment "=" operator, and an equality "==" operator. Because the precedence of evaluation for the "==" operator is higher than the assignment operator, the relational comparison is made first, and then the result is assigned to variable a. In this particular case, 3 is not equal to 5, and so variable a assumes a value of 0 (i.e., false).

When relational operands are applied to matrices of the same size, as in

```
>> A = [ 1 2; 3 4 ];
>> B = [ 6 7; 8 9 ];
>> A == B
  ans =
       0     0
       0     0
>> A < B
  ans =
       1     1
       1     1
>>
```

the result will be a matrix of 0s and 1s giving the value of the relationship between corresponding entries.

Logical expressions can be combined by using the logical operands, `and`, `or`, and `not` (see the lower half of Table 2.5). The following table shows how the logical operators can be combined.

```
=======================================================
       A        B         A & B        A | B         ~A
=======================================================
    True     True          True         True      False
    True    False         False         True      False
   False     True         False         True       True
   False    False         False        False       True
=======================================================
```

Logical operands also apply to matrices of the same size. In the script

```
>> A = [ 1 2; 3 4 ];
>> B = [ 6 7; 8 9 ];
>> A & B
ans =
     1     1
     1     1
>>
```

all the matrix elements in `ans` evaluate to 1 because matrices `A` and `B` contain only nonzero matrix elements.

MATLAB also has an ensemble of built-in functions for the evaluation of logical expressions involving matrices. A summary of their capabilities is given in Table 2.5.

Selection Constructs

Generally speaking, a selection construct enables the details of a program's problem-solving procedure to be tied to the evaluation of one or more logical expressions. The following diagram shows the syntax for three commonly used selection constructs in the MATLAB programming language:

```
If-end construct          If-else-end construct       If-elseif-end construct
================          =====================       =======================

if < condition1 >,        if < condition1 >,          if < condition1 >,
   < program1 >              < program1 >               < program1 >
end;                      else                        elseif < condition2 >,
                             < program2 >                < program2 >
                          end;                        elseif < condition3 >
                                                         < program3 >

                                                        .....

                                                      elseif < conditionN >
                                                         < programN >
                                                      end;
```

In each of these constructs, the block of statements <program1> will be executed when the logical expression <condition1> evaluates to true. Otherwise, the program control moves to the next program construction. For the leftmost construct, this means the end of the selection construct. For the if-else-end construct, the block of statements <program2> will be executed when <condition1> evaluates to false. For example, in the block of statements

```
>> a = 2; b = 1;
>> if a < b,
      c = 3;
   else
      c = 4;
   end;
>> c
   c =
        4
>>
```

the logical expression a<b evaluates to false, thereby causing the second block of program statements to be executed.

Another variation is the "if-elseif-end" construct. MATLAB will systematically evaluate the sequence of logical expressions <condition1>, <condition2> $\cdots$ <conditionN>

until one evaluates to true. After the corresponding block of <program> statements has been executed, the program control will jump to the end of the "if-elseif-end" construct.

Looping Constructs

MATLAB provides a number of looping constructs for the efficient computation of similar calculations. The syntax for the `while` and `for` looping constructs is

```
While-loop construct
====================

while < condition1 >,
    < program1 >
end
```

```
For-loop construct
====================

for i = <array of values>
      < program1 >
end
```

In the while looping construct, the block of statements <program1> will be executed while the logical expression <condition1> evaluates to true. For example, the following script

```
A "while looping" construct
===========================

>> i = 2;
>> while ( i < 4 ),
      y = 4*i,
      i = i + 1,
   end
>>
```

```
Iteration No   i     i < 4      y
=================================

          1    2      true      8
          2    3      true     12
          3    4     false
```

shows a simple while loop and an analysis of the values involved in the looping construct.

In the for looping construct, the block of statements <program1> will be executed for each of the vectors `i` defined by the column elements in <array>. For example, the looping construct

```
A "for looping" construct
=========================

>> for i = [2,4,5,6,10],
       c = 2*i
   end
>>
```

```
Iteration No     i      c
=========================

          1      2      4
          2      4      8
          3      5     10
          4      6     12
          5     10     20
```

executes five times. The values for the variable `i` during execution are successively `2`, `4`, `5`, `6`, and `10`. When <`array`> is a two-dimensional matrix, the values of `i` will be vectors corresponding to the matrix columns of <`array`>. For example, an extension of our previous example is

```
A "for looping" construct       Iteration No       i           c
=========================       ====================================

>> for i = [ 2 4 5 6 10;
             1 2 3 4  5 ],                1  [  2 1]'   [ 4   2 ]'
       c = 2*i                            2  [  4 2]'   [ 8   4 ]'
   end                                    3  [  5 3]'   [ 10  6 ]'
>>                                        4  [  6 4]'   [ 12  8 ]'
                                          5  [ 10 5]'   [ 20 10 ]'
```

Looping constructs may be nested of course. Here is an example of the contents of a matrix being initialized inside a nested for loop:

```
MATLAB source code                   i      j                   A(i,j)
======================          ======================================

>> for i=1:2,                        1      1      A(1,1) = 1/1 = 1.0
      for j=1:2,                     1      2      A(1,2) = 1/2 = 0.5
          A(i,j) = i/j;              2      1      A(2,1) = 2/1 = 2.0
      end                            2      2      A(2,2) = 2/2 = 1.0
   end
>>
```

There are actually two loops here, with one nested inside the other; they define A(1,1), A(1,2), A(2,1), and A(2,2) in that order.

Programming Tip. Looping constructs in the MATLAB programming language need to be used with care. The first potential problem is with `while` loops and the implicit danger of the program control becoming trapped inside a loop because <`condition1`> never evaluates to false.

A second important issue is performance. Generally speaking, MATLAB code written with the `for` and `while` looping constructs will execute much slower than if MATLAB's implied looping constructs are employed. For example, the block of MATLAB code

```
>> x = [ 0 : 0.1 : 2.0 ];
>> y = zeros(1,21);
>> for i = 1:21,
       y(i) = sin(x(i));
   end
>>
```

will have faster execution if it is simply written

```
>> x = [ 0 : 0.1 : 2.0 ];
>> y = sin(x);
```

This is because the former looping constructs are interpreted, and MATLAB's implied looping constructs are executed as a low-level compiled code.

2.6 General-Purpose Matrix Functions

MATLAB has an ensemble of built-in matrix functions that are useful for general-purpose engineering and scientific computations.

Sorting the Contents of a Matrix

When `x` is a matrix, `sort(x)` sorts each column of `x` in ascending order. For example

```
>> x = [  1   3   4;
          5  -4  16;
         -4  -8 -10 ];
>> sort(x)
ans =
    -4    -8   -10
     1    -4     4
     5     3    16
>>
```

`sort(x)` also returns an index matrix `i`, containing a mapping between elements in the `x` matrix and the `y` (sorted) matrix. For example

```
>> [ y, i ] = sort(x)
y =
    -4    -8   -10
     1    -4     4
     5     3    16
i =
     3     3     3
     1     2     1
     2     1     2
>>
```

In colon notation `y(:,j)` = `x(i(:,j),j)`. Hence, by walking down the first column of the index matrix `i`, we see that matrix elements `x(3,1)`, `x(1,1)`, and `x(2,1)` have increasing values.

Summation of Matrix Contents

When `x` is a vector, `sum(x)` returns the sum of the elements of `x`. When `x` is a matrix, `sum(x)` returns a row vector with the sum over each column. For example,

```
>> sum ( [  1   3   4;
            5  -4  16;
           -4  -8 -10 ] )
ans =
     2    -9    10
>>
```

Watch out for this function being used in statistical analysis of experimental data and least squares analysis.

Maximum/Minimum Matrix Contents

Maximum/Minimum Matrix Contents

The MATLAB function `max(A)` returns the maximum element in vector `A`, and `min(A)` returns the minimum element in vector `A`. When `A` is a matrix, `max(A)` will return a row vector containing the maximum value in each column of `A`. Similarly, `min(A)` will return a row vector containing the minimum value in each column of `A`. For example, the script of code

```
>> A = [ 1 2 3 4; 5 6 7 8 ];
>> max (A)
ans =
     5     6     7     8
>> min (A)
ans =
     1     2     3     4
>>
```

shows how the maximum and minimum values of each column of `A` are computed.

Some MATLAB functions can return more than one value. In the case of `max`, the interpreter returns the maximum value and also the column index where the maximum value occurs. For example, the script of code

```
>> [m, i] = max(A)
m =
     5     6     7     8
i =
     2     2     2     2
>>
```

shows how the maximum value in each column of A is located at index 2.

Random Numbers

The MATLAB function `rand` returns numbers and matrices of numbers containing elements that are uniformly distributed between zero and one. Theoretical considerations indicate that the average value of these elements should be close to 0.5.

For example, the function call `rand(3)` will return a three-by-three matrix with random entries. The function call

```
>> A = rand(10,30);
```

generates a 10-by-30 matrix of elements uniformly distributed between zero and one, and assigns the result to matrix A. The average value of the matrix elements can be computed by simply writing

```
>> average = sum(sum(A))/300
average =
    0.5059
>>
```

In the nested function call `sum(sum(A))`, the sum of the elements in each matrix column is computed, and then the sum of the matrix column sums is evaluated. The average value of the matrix elements corresponds to the sum of matrix element values in A divided by the total number of matrix elements, 300 in this case.

The function `randn` generates matrices with elements chosen from a normal distribution with mean 0.0 and variance 1.0.

The mean value and scatter of uniform and normally distributed random numbers may be adjusted by multiplying the contents of `rand` and `randn` by suitable linear transformations. For example, to obtain a 3-by-5 matrix of elements uniformly distributed between -1 and 1, we can simply write:

```
>> B = 2*rand(3,5) - ones(3,5)
B =
   -0.1650    0.8609   -0.8161    0.4024   -0.4751
    0.3735    0.6923    0.3078    0.8206   -0.9051
    0.1780    0.0539   -0.1680    0.5244    0.4722
>>
```

Sequences of random numbers will be generated when the functions `rand` and `randn` are called repeatedly. Sequences of random numbers that are repeatable – in other words, the same sequence of

random numbers will be the same each time the MATLAB program is executed – can be generated by initializing a seed number. The syntax is

```
>> rand  ('seed', n );
>> randn ('seed', n );
```

where `n` is the seed number (greater than unity). Further information on initializing sequences of random numbers in MATLAB can be obtained by typing `help rand`.

2.7 Program Development with M-Files

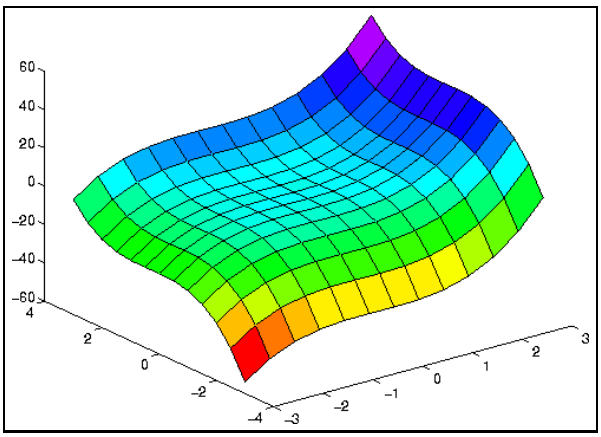


Figure 2.2: Matlab Graphic

We have so far assumed in this tutorial that all MATLAB commands will be typed in at the keyboard. Practical considerations dictate that this mode of operation is suitable only for the specification of the smallest problems, perhaps a handful of MATLAB commands or less. A much better problem-solving approach is to use a text editor to write the commands in an M-file, and then ask MATLAB to read and execute the commands listed in the M-file. This section describes two types of M-files. Script M-files correspond to a main program in programming languages such as C. Function M-files correspond to a subprogram or user-written function in programming languages such as C. An M-file can reference other M-files, including referencing itself recursively.

User-Defined Code and Software Libraries

Computer programs written in MATLAB are a combination of **user-defined code** – that is, the computer program code we write ourselves – and collections of external functions located in **software libraries** or MATLAB toolboxes. Software library functions are written by computer vendors, and are automatically bundled with the compiler. Software libraries play a central role in the development of C programs because the small number of keywords and operators in C is not enough to solve real engineering and scientific problems in a practical way. What really makes C useful is its ability to communicate with collections of functions that are external to the user-written source code. This is where much of the real work in C programs takes place.

The ANSI C standard requires that certain libraries are provided with all implementations of ANSI C. For example, the standard library contains functions for I/O, manipulation of character strings, handling of run-time errors, dynamic allocation and deallocation of memory, and functions for C program interaction with the computer's operating system. Mathematical formulae are evaluated by linking a C program to the math library. Engineering application programs may also communicate with graphical user interface, numerical analysis, and/or network communications libraries.

Generally speaking, if there is a library function that meets your needs, by all means use

it. The judicious use of library functions will simplify the writing of your C programs, shorten the required development time, and enhance C program portability. Software is said to be portable if it can, with reasonable effort, be made to execute on computers other than on the one on which it was originally written.

Program Development Cycle

When you are a novice programmer, the two most important issues you are likely to be concerned about are learning the syntax of the C language and becoming familiar with the step-by-step details of planning, writing, compiling, running, testing, and documenting small C programs. The pathway of C program development is summarized in Figure 2.3. The key steps are as follows:

1. **Develop Pseudocode.** Pseudocode is a nickname for an English description of ideas that will eventually be translated into a programming language. Like most engineering design processes, development of pseudocode descriptions is inherently iterative. Pseudocode descriptions begin with general high-level statements of the required problem solving procedures and are successively refined until they look "more or less" like a program. For us, pseudocode is a description of the program steps that can eventually be written in C.

   Beginning the program development cycle with a pseudocode problem description is important because it forces us to deal with design issues at an early stage, when corrections and changes are easy to make. For example, with a pseudocode description of a problem solving procedure in hand, the correctness of an algorithm can be tested by stepping through it in your head or on scratch paper. Most experienced programmers will attest to the fact that errors in logic are the most difficult to find, especially in the latter stages of program development. Getting programs to compile (syntax, structure, etc.) is the easy part.

2. **Edit Program.** The pseudocode is translated into a C program by using a text editor for the creation of source code files containing a description of the problem to be solved. Most computer systems support a variety of text editors. Because some of them will be much easier to learn and use than others, it is a good idea to ask your instructor for advice on the best place to start and for information on basic editor commands to insert and modify text.

   All file names in our C programs will consist of a base name with an optional period and suffix. The first character of the file name must be a letter, possibly followed by more letters and numbers. C source code files are where the C language description of our engineering problem descriptions will go and, by convention, they will be stored in files having names with a `.c` suffix. Header files contain function declarations and symbolic constants, and act as an interface between user-defined C code and one or more external software libraries. By convention, header file names will have a `.h` suffix. We soon see that object files are generated

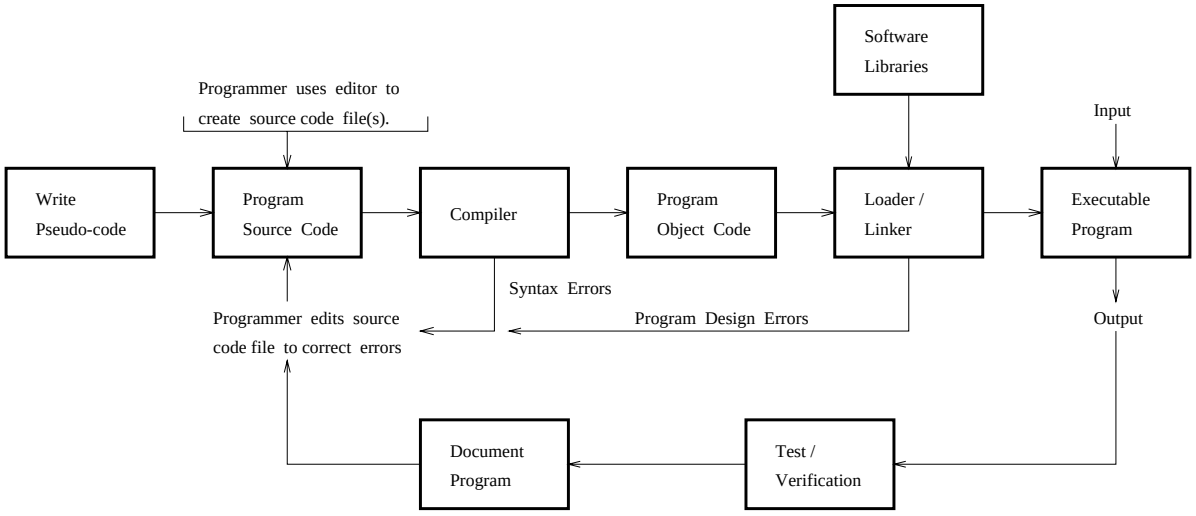


Figure 2.3. Program Development Cycle for MATLAB (needs fixing!!!!)

by the compiler and have a `.o` suffix. The suffix convention is used by compilers to identify the type of information stored in a file.

3. **Compile Program.** A C compiler is a computer program that translates high-level problem descriptions, written in the C language, into equivalent low-level language descriptions that can be understood by a particular type of computer (i.e., the computer hardware). You should think of the C compiler as a type of machine that only understands the grammatical rules of the C programming language. If the compiler does not receive proper input it can only try to interpret the error, report what is wrong by printing error messages, and exit. If the rules of the language are satisfied, however, the compiler will generate one or more object files containing low-level machine language instructions. The linker is a software program that resolves all cross references among object files, connects the user-written code to relevant software library modules and, finally, generates a single executable program file. The chain of "compilation activities" is summarized by the center row of blocks in Figure 2.3.

4. **Performance Analysis.** Performance analysis consists of an evaluation of the program performance with respect to the design specifications. As indicated on the right-hand side of Figure 2.3 most computer programs will receive some form of input from the keyboard, or perhaps from a file, and generate output to the computer screen, or perhaps to a file. The critical question to ask at this point is "does the computer program do what it was designed to do?"

5. **Debug and Test Program.** The first few drafts of a C program nearly always contain errors (or bugs). You will need to correct (or debug) your program if there are syntax, executional, or logic errors, as determined in the performance analysis or compilation. Syntax errors are those encountered when the program is compiled into machine code; if you have typing errors, then the compiler will not be able to understand what you are doing. Execution errors are found when the program is run, for example, if the bounds of an array are exceeded or, perhaps, if division by zero occurs. Logic errors are usually discovered after the program has finished executing or during execution. If execution does not proceed as expected, then perhaps there is a flaw in the logic of the program. In each case, symbolic debuggers such as `dbx` [8] may be used to interactively run the program and detect errors (see following comment). It may be faster and easier to go back and reconsider the design of the program. Have you effectively transferred the design to C, or is there a flaw in the design? Answering these questions first may lead to a solution faster than learning a whole new bag of tricks with a debugger.

6. **Document Code.** You should document your source code so that other programmers know what the source code does and how it may be used. The process of documenting your program should be integrated into the program development cycle. Some (experienced) programmers will tell you that they first develop the code and then document it. It is our experience, however, that with the pressures of life being what they are (everyone is pressed for time), this strategy of development results in code that works but is poorly or never documented.

In this solution of practical engineering problems, several iterations of Steps 1 through 6 may be required before a fully developed, correct, working program is obtained. Indeed, the purpose of the looping constructs in Figure 2.3 is to show that when an error occurs in the language syntax, program design, and/or program performance, the programmer must return to Steps 1 and 2, make the necessary adjustments to the program design/source code, and recompile and test the program.

M-File Preparation

UNIX Platforms

In a UNIX environment, a standard text editor can be used to prepare the contents of an M-file. The M-file should be located in the same directory as the MATLAB program. After the M-file has been created and saved, the list of commands inside the M-file can be executed by moving to the MATLAB window and typing the M-file name without the `.m` extension. To be more precise, suppose that we create a program file

```
myfile.m
```

in the MATLAB language. The commands in this file can be executed by simply typing the command `myfile` at the MATLAB prompt. The MATLAB statements will run like any other MATLAB function. You do not need to compile the program since MATLAB is an interpretative (not compiled) language.

UNIX commands can be invoked from inside the MATLAB environment by preceding the command by a `!` (e.g., `!ls` will list the files located in the current working directory).

Macintosh and PC Environments

An M-file can be prepared in a Mac/PC environment by clicking on `new` in the file menu. A new window will appear. After the list of commands has been typed into the file, it can be saved by clicking on `save as` in the edit menu. The file can be executed from the command window by typing the M-file name without the `.M` extension. A second pathway in the Macintosh environment is to click on `save and go` in the file menu.

Script M-Files

Suppose that we use our favorite text editor to create a file called `sketch.m`, containing

```
[x y] = meshgrid(-3:.5:3, -3:.5:3);
z = x.^3 + y.^3 + x.^2 - y.^2
surf(x,y,z);
```

Without getting into too many details on MATLAB graphics at this point, the first two lines generate a rectangular grid of `x,y,z` points in a format suitable for a three-dimensional MATLAB plot. The function call `surf(x,y,z)` asks MATLAB to generate a surface plot. The simple command

```
>> sketch
```

will force MATLAB to open the file `sketch.m` and systematically execute the sequence of commands within. The result, shown at the top of Section 13.6, has been captured from MATLAB graphics on a UNIX workstation, and will be the same as if you had entered the three lines of the file at the prompt.

Function M-Files

Function files provide extensibility to MATLAB by allowing you to create new problem-specific functions having the same status as other built-in MATLAB functions. Unlike some other traditional programming languages, each function file contains one function. Functions are like

scripts, but for the purposes of enhancing computational speed, are compiled into a low-level byte-code when called for the first time.

Square Root Calculation

In this section we develop a MATLAB function file called `sqroot.m` to compute the square root of a positive number `N` via the recursive relationship

$$x_{n+1} = \frac{1}{2} \cdot \left[x_n + \frac{N}{x_n} \right]. \tag{2.13}$$

A good initial estimate of the square root is $x_o = N/2$. The program uses the criterion

$$\left| \frac{x_{n+1} - x_n}{x_n} \right| \leq \varepsilon \tag{2.14}$$

for a test on convergence, where ε is a very small number. In our implementation, ε is taken as `eps`, the floating point precision for the computer being used (see Table 2.1).

Computer Program 2.1 : Square Root Calculation

```
function sqroot(x)
% SQROOT Compute square root by Newton's method

% Check that value of function argument is positive.

  if x <= 0,
     error('In sqroot() : argument x must be positive');
  end;

% Initial guess.

  xstart = x/2;

% Iteration loop to compute square root.

  for i = 1:100
      xnew = ( xstart + x/xstart)/2;     % new estimate of square root.
      disp(xnew);                        % print xnew.
      if abs(xnew - xstart)/xnew < eps,  % check convergence of iterations.
         break,                          % iterations.
```

```
    end;
    xstart = xnew;                    % update estimate of square root.
end
```

When MATLAB executes a function M-file for the first time, it will open the appropriate text M-file and compile the function into a low-level representation that will be stored within MATLAB. For those cases where an M-file function references other M-file functions, they will also be compiled and placed in MATLAB's memory.

The command sequence:

```
>> format long
>> format compact
>> sqroot(20)
```

loads the contents of `sqroot.m` into MATLAB memory and then computes the square root of 20 by iteration

```
>> sqroot(20)
     6
   4.66666666666667
   4.47619047619048
   4.47213779128673
   4.47213595499996
   4.47213595499958
   4.47213595499958
>>
```

A function M-file will terminate its execution when either a `return` statement is encountered or, as is the case in this example, an end-of-file (EOF) is reached.

The first line of Program 2.1 contains declarations for the function name, `sqroot`, and its input argument(s). Without this line the file would simply be a script file. MATLAB requires that M-file functions be stored in files having the same name (i.e., the function `sqroot` must be stored in an M-file called `sqroot.m`).

Your MATLAB programs should contain lots of comments telling the reader in plain English what is occurring. Some day that reader will be you, and you will wonder what you did. Comment lines up to the first noncomment line in a function M-file are the help text returned when you request online help. For Program 2.1, this means

```
>>  help sqroot
```

```
    SQROOT Compute square root by Newton's method
>>
```

We also urge you to use the indented style that you have seen in the previous programs. It makes the programs easier to read, the program syntax easier to check, and forces you to think in terms of building your programs in blocks.

Matrices and variables in a function file have scope that is local by default. The `sqroot` function is simple enough that all the relevant details can be handled with `x`, `xstart`, and `xnew`. The statement `disp( xnew )` displays the matrix `xnew` as text, without printing the matrix name.

Error messages play an important role in nearly all computer programs since they help you debug future programs. Because the algorithm in Program 2.1 can only compute the square root of positive floating point numbers, we need to check that `sqroot()` is called with positive argument. The block of MATLAB code

```
if x <= 0,
   error('In sqroot() : argument x must be positive');
end;
```

prints an error message when argument `x` is negative. For example, the script of code

```
>> sqroot(-4)
Error using ==> sqroot
In sqroot() : argument x must be positive
>>
```

tries to computer the square root of -4. The function `error()` displays the message enclosed within single quotes and causes an exit from the M-file to the keyboard.

Variations on Standard Use. It is important to bear in mind that our implementation of `sqroot()` is very minimal. As already demonstrated, function calls of the type `sqroot(20)` are possible. However, we want the ability to integrate function calls into expressions involving matrices. This can be done by changing the function declaration to

```
function [y] = sqroot(x)
```

Now expressions of the type

```
>> x = 3.5 * sqroot ( 10000 )
x =
      350
>>
```

are possible.

Statistics of Experimental Data

In this example, we show how the mean and standard deviation of an array of experimental data can be computed inside a single MATLAB function. Suppose that

$$x = [x_1, x_2, x_3, \cdots, x_N]^T \tag{2.15}$$

is a column vector of data points collected in an experiment. The mean value of the data points is given by

$$\mu_x = \frac{1}{N}\left[\sum_{i=1}^{N} x_i\right] \tag{2.16}$$

and the standard deviation by

$$\sigma_x = \left[\sum_{i=1}^{N} \frac{[x_i - \mu_x]^2}{N}\right]^{1/2} = \left[\frac{1}{N}\sum_{i=1}^{N}\left[x_i^2\right] - {\mu_x}^2\right]^{1/2}. \tag{2.17}$$

For the case where an experiment generates a two-dimensional array of experimental data, Equations (2.16) and (2.17) can be applied to each column in the two-dimensional matrix.

Computer Program 2.2 : Statistics of Experimental Data

```
function  [mean, stdev] = stat(x)

% STAT  Mean and standard deviation
%       For a vector x, stat(x) returns the
%       mean and standard deviation of  x.
%       For a matrix x, stat(x) returns two row vectors containing,
```

```
%      respectively, the mean and standard deviation of each column.

[m  n] = size(x);

if m == 1
   m = n;     % handle case of a row vector
end

mean  = sum(x)/m;
stdev = sqrt(sum(x.^ 2)/m - mean.^2);
```

The first line of Program 2.2 tells MATLAB that function `stat` will accept an array argument, and return a one-by-two matrix containing the mean and standard deviation of the experimental data points. Notice that `mean` and `stdev` are both computed inside the function body.

As indicated in the documentation for Program 2.2, when `x` is a row or column vector, `stat(x)` returns the mean and standard deviation of `x`. For a general matrix `x`, `stat(x)` returns two row vectors containing, respectively, the mean and standard deviation of each column. With Program 2.2 placed in the M-file `stat.m`, the MATLAB commands

```
>> y = [1:10];
>> [ym, yd] = stat(y)

  ym =
       5.5000
  yd =
       2.8723
>>
```

compute the mean and standard deviation of the integers 1 through 10. The mean and standard deviation of the entries in the vector `y` are assigned to `ym` and `yd`, respectively.

Variations on Standard Use. Two variations on the standard use of `stat.m` are possible. First, single assignments can also be made with a function having multiple output arguments. For example,

```
>> xm = stat(x)
```

(no brackets needed around `xm`) will assign the mean of `x` to `xm`. In this case, standard deviation of `x` will be lost. Second, when `x` is a two-dimensional matrix, `stat(x)` will return the matrices containing mean and standard deviation of each column in `x`. For example, when

```
>> x = [ 1 2; 3 4; 5 6 ];
```

`stat(x)` gives

```
>> [ xm, xstd ] = stat(x);
>> xm
xm =
     3     4
>> xstd
xstd =
    1.6330    1.6330
>>
```

Programming Tip. Generally speaking, function M-files are more difficult to debug than script M-files because you cannot use MATLAB to print the value of variables inside the function. We therefore suggest that you develop function files first as script files and then once the script file works properly, encapsulate the script inside a function declaration with appropriate arguments and comment statements.

Handling Name Conflicts

Suppose that we do not know `sum` is a built-in function and type the MATLAB statement

```
>> x = 1;
>> y = 2;
>> z = 3;
>> sum = x + y + z;
```

with the intent of using `sum` to represent the sum of values stored by the variables `x`, `y`, and `z`. The name `sum` now represents a variable and MATLAB's built-in `sum` function is hidden (you can check this with the command `who`).

When a name is typed at the prompt or used in an arithmetic expression, the MATLAB interpreter evaluates the name by systematically walking through four steps

1. It looks to see if the name is a variable.
2. It looks to see if the name is a built-in function.
3. It looks in the current directory to see if the name matches a script file (e.g., `sum.m`).
4. It looks in the MATLAB search path for a script file matching the name (e.g., `sum.m`).

Clearing the variable `sum` (i.e., by typing `clear sum`) reactivates the built-in function `sum`.

2.8 Engineering Applications

Now that we are familiar with MATLAB's matrix and M-file capabilities, we work step by step through the design and implementation of four programs. They are

1. A MATLAB program that computes and plots the relationship between Fahrenheit and Celsius temperature.
2. A MATLAB program that computes and plots the time-history response (i.e., "displacement versus time" and "velocity versus time") of an undamped single degree of freedom (SDOF) oscillator.
3. A MATLAB program that will prompt a user for the coefficients in a quadratic equation, and then compute and print the roots.
4. A MATLAB program that reads experimental data from a data file, and then computes and plots a histogram of the data. The mean and standard deviation of the experimental data are computed with `stat.m` (see Section 2.7).

A key issue in the design of almost every MATLAB program is the problem of finding a good balance among the use of user-defined code and built-in MATLAB library functions. The advantages of user-defined functions are that they enable the development of customized problem-solving procedures. However, writing and testing user-defined code can be a very time-consuming process. Library functions have the benefit of enabling reuse of code that has already been written and thoroughly tested. A judicious use of library functions can result in significant reductions in the time and effort needed to write and test MATLAB programs.

Obtaining a good balance in the use of user-defined code and library functions requires experience and, to some extent, is an art. This means that novice MATLAB programmers must place a priority on becoming familiar with the availability and purposes of library functions.

Temperature Conversion Program

Problem Statement. The relationship between temperature measured in Fahrenheit (T_f) and temperature Celsius (T_c) is given by the equation

$$T_f = \frac{9}{5}T_c + 32 \tag{2.18}$$

Write an M-file that computes and plots the temperature conversion relationship for the range -50 through 100 degrees Celsius.

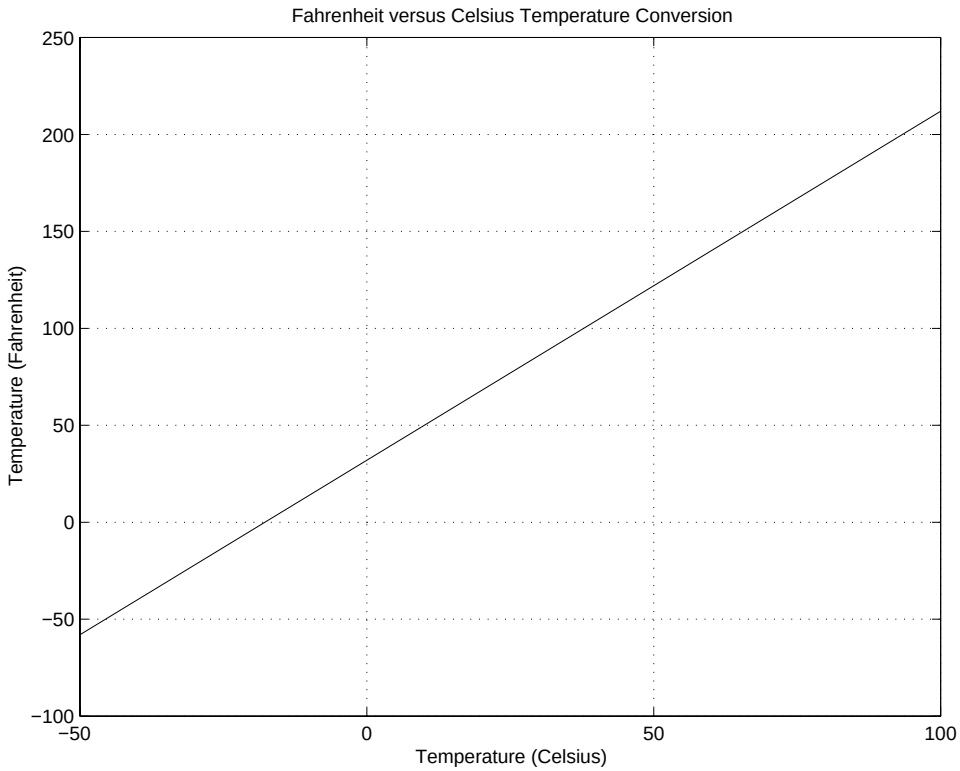


Figure 2.4. Fahrenheit Versus Celsius

In addition to having a suitable title, the vertical and horizontal axes of your graph should be properly labeled. See, for example, Figure 2.4.

Pseudocode. We begin our implementation with the observation that Equation (2.18) is linear. It follows that the temperature conversion relationship can be plotted by simply drawing a straight line between the (T_c, T_f) coordinates evaluated at T_c = -50 and T_c = 100. The pseudocode for this problem is

```
allocate two-by-one arrays to hold the Celsius and
Fahrenheit temperatures.

set the Celsius array values to -50 and 100 degrees.
compute the Fahrenheit temperatures.

plot the array values.
label the horizontal/vertical axes.
```

```
add a plot title.
```

Program Source Code. Here is the program source code.

Computer Program 2.3 : Temperature Conversion Program

```
% ======================================================================
% temperature.m -- Compute and plot a graph of Celsius versus Fahrenheit
%                  for the range -50 through 100 degrees Celsius.
%
% Written By : Mark Austin                                    March 1997
% ======================================================================

% Allocate arrays for Celsius and Fahrenheit temperatures.

  tempC = zeros(2,1);
  tempF = zeros(2,1);

% Compute temperatures at graph end-points.

  tempC(1)  = -50; tempF(1)  = 9*tempC(1)/5 + 32;
  tempC(2)  = 100; tempF(2)  = 9*tempC(2)/5 + 32;

% Plot and label the graph

  plot( tempC, tempF );
  grid;
  xlabel('Temperature (Celsius)');
  ylabel('Temperature (Fahrenheit)');
  title('Fahrenheit versus Celsius Temperature Conversion');
```

Running the Program. As indicated in the block of comment statements at the top of the source code, Program 2.3 is stored in the M-file `temperature.m`. The command

```
>> temperature
```

executes the script file and generates Figure 2.4.

Program Architecture. Table 2.8 shows how key tasks in the problem statement have been mapped to user-defined MATLAB code and function calls in MATLAB.

Tasks	*	User-defined code	MATLAB library
Define arrays tempC and tempF.	*	In temperature.m	zeros()
	*		
Initialize tempC.	*	In temperature.m	
Compute tempF.	*	In temperature.m	
	*		
Plot tempF versus tempC.	*	In temperature.m	plot()
Add a grid to the plot.	*		grid
Label the horizontal axis.	*		xlabel()
Label the vertical axis.	*		ylabel()
Add a plot title.	*		title()

Table 2.8. Mapping of Tasks to MATLAB Code

As already mentioned, all the user-defined MATLAB code is located in the script M-file `temperature.m`. A variety of built-in MATLAB functions are employed for the matrix allocation and to generate the graph.

Analysis of the MATLAB Code. The Celsius and Fahrenheit temperature values are stored in the arrays `tempC` and `tempF`, respectively. Once array `tempC` has been initialized, the contents of `tempF` are evaluated via Equation (2.18). The command

```
plot( tempC, tempF );
```

creates a linear plot of array `tempF` versus `tempC`. The contents of `tempC` are the data points along the horizontal axis. The data points along the vertical axis are stored in `tempF`. The command `grid` adds a grid to the plot. Labels along the horizontal and vertical axes, and a plot title, are added to the figure with the three commands

```
xlabel('Temperature (Celsius)');
ylabel('Temperature (Fahrenheit)');
title('Fahrenheit versus Celsius Temperature Conversion');
```

Free Vibration Response of Undamped Single Degree of Freedom (SDOF) System

Problem Statement. The free vibration response of an undamped single degree of freedom (SDOF) oscillator is given by

$$y(t) = y(0)\cos(wt) + \frac{v(0)}{w}\sin(wt) \tag{2.19}$$

where t is time (seconds), and m and k and the mass and stiffness of the system, respectively. $y(t)$ and $v(t)$ are the displacement and velocity of the system at time t. By definition, $w = \sqrt{(k/m)}$ is the circular natural frequency of the system.

Write an M-file that will compute and plot the "displacement versus time" (i.e., y(t) versus t) and "velocity versus time" (i.e., v(t) versus t) for the time interval 0 through 10 seconds when mass, m = 1 and stiffness, k = 10. The initial displacement and velocity are y(0) = 1 and v(0) = 10, respectively. To ensure that your plot will be reasonably smooth, choose an increment in your displacement and velocity calculations that is no larger than 1/20th of the system period $T = 2\pi\sqrt{(m/k)}$.

Pseudocode. We begin our analysis by noting that $v(t)$ is simply the derivative of the displacement with respect to time. Hence, in mathematical terms

$$v(t) = \frac{dy(t)}{dt} = -y(0)w\sin(wt) + v(0)\cos(wt) \tag{2.20}$$

Since neither $y(t)$ nor $v(t)$ are linear functions, we implement our solution in three stages. First, a two-dimensional array will be allocated for the system response storage. Components of the system response (i.e., $y(t)$ and $v(t)$) will then be computed and stored in the array columns. Finally, the array contents will be plotted. The pseudocode for this problem is

```
define number of points, "npoints", for plotting.
setup array response (npoints,3) for storing t, y(t), and v(t).
    column 1 will store the time values
    column 2 will store the system displacement
    column 3 will store the system velocity

define problem parameters;
    sdof mass "m"
    sdof stiffness "stiff"
    time increment 'dt' for plotting purposes.
    initial displacement "x0"
    initial velocity     "v0"

for i = 1 to npoints
    column 1 of array = time t = (i-1)*dt;
    column 2 of array = y(t)
    column 3 of array = v(t)
end loop

plot y(t) versus t for t = 0 to 10 seconds.
hold the plot.
plot v(t) versus t for t = 0 to 10 seconds.
```

```
    add a grid to the plot.
    label the horizontal/vertical axes.
    add a plot title.
```

Notice how certain statements have been indented to show the structure of the eventual program. Once preliminary ideas are written this way the steps can be refined until the pseudocode is very close to the target language.

The fidelity of the system response evaluation is controlled by the variable `npoints`. In addition to using several arrays for storage and plotting purposes, the pseudocode employs a looping construct for the systematic evaluation of the system response at regular intervals between $t = 0$ and $t = 10$ seconds.

Program Source Code. Here is the program source code.

Computer Program 2.4 : Free Vibration Response of Undamped SDOF System

```
% ================================================================
% sdof.m -- Compute dynamic response of sdof system.
%
% Written By : Mark Austin                                March 1997
% ================================================================

% Setup array for storing and plotting system response

  nopoints = 501;
  response = zeros(nopoints,3);

% Problem parameters and initial conditions

  mass  = 1;
  stiff = 10;
  w     = sqrt(stiff/mass);
  dt    = 0.02;

  displ0    =  1;
  velocity0 = 10;

% Compute displacement and velocity time history response

  for i = 1 : nopoints
      time = (i-1)*dt;
      response(i,1) = time;
```

```
      response(i,2) =  displ0*cos(w*time)   + velocity0/w*sin(w*time);
      response(i,3) = -displ0*w*sin(w*time) + velocity0*cos(w*time);
  end

% Plot displacement versus time

  plot(response(:,1), response(:,2));
  hold;

% Plot velocity versus time

  plot(response(:,1), response(:,3));

  grid;
  xlabel('Time (seconds)');
  ylabel('Displacement (m) and Velocity (m/sec)');
  title('Time-History Response for SDOF Oscillator');
```

Running the Program. Assume Program 2.4 is stored in the script M-file `sdof.m`. The command

```
>> sdof
```

generates the curves shown in Figure 2.5.

How do we know that these graphs are correct? From a mathematical viewpoint, we expect that the natural period of this system will be

```
T = 2*pi*sqrt(m/k) = 6.282/sqrt(10) = 2.0 seconds,
```

A quick visual inspection of Figure 2.5 reveals that both $y(t)$ and $v(t)$ oscillate with a natural period of 2 seconds (the time-step increment, dt = 0.02 sec easily satisfies the stated criteria for a smooth graph). The second point to notice is that at t = 0 seconds, the displacement and velocity graphs both match the stated initial conditions. Moreover, you should observe that because the initial velocity is greater than zero, we expect the $y(t)$ curve to initially increase. It does. A final point to note is the relationship between the displacement and velocity. When the oscillator displacement is at either its maximum or minimum value, the mass will be at rest for a short time. In mathematical terms, peak values in the displacement curve correspond to zero values in the velocity curve.

Program Architecture. The left- and right-hand sides of Table 2.9 show how key tasks in the SDOF problem statement have been mapped to user-defined MATLAB code and calls to MATLAB functions. Once again, because this problem is relatively straightforward, all the user-defined source code is located within one M-file, `sdof.m`.

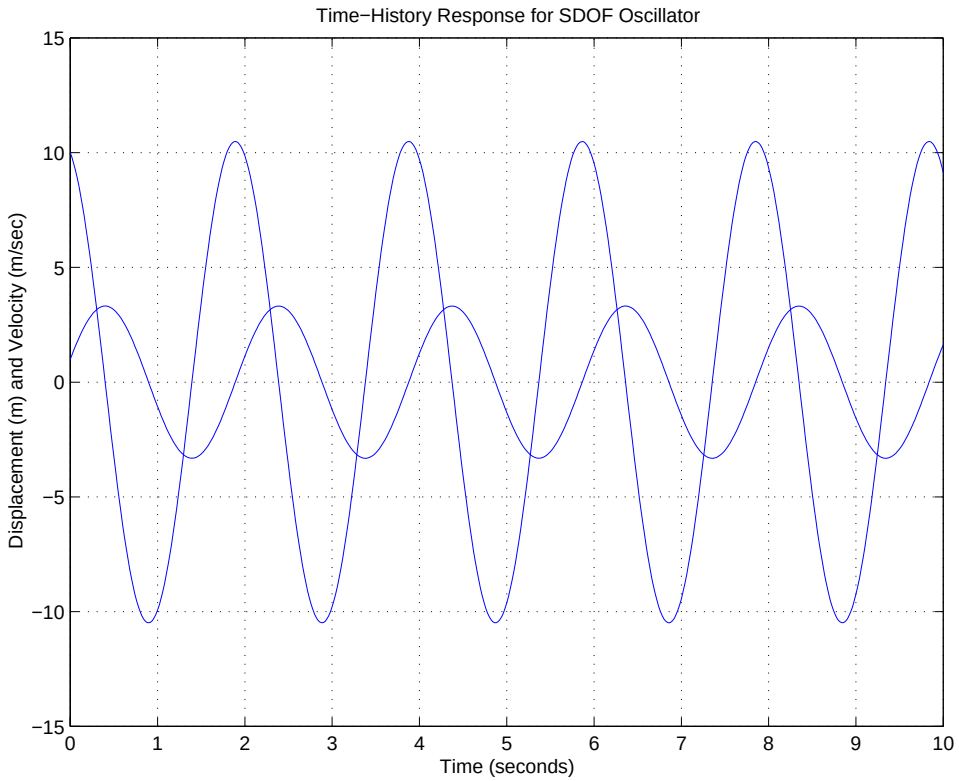


Figure 2.5. Time-history response of SDOF oscillator

Built-in MATLAB functions are used for the response array allocation, the `sin()` and `cos()` trigonometric calculations, and the square root calculation. Of course, we could have substituted `sqrt()` in the MATLAB library with the `sqroot()` function from Program 2.1. Both implementations give the same numerical result. However, use of the `sqroot()` function requires that MATLAB automatically locate and compile `sqroot.m` after it has been referenced from `sdof.m`.

Analysis of the MATLAB Code. The variables `displ0` and `velocity0` store the SDOF displacement and velocity at time, t = 0. The looping construct

```
for i = 1 : nopoints
    time = (i-1)*dt;
    response(i,1) = time;
    response(i,2) =  displ0*cos(w*time)   + velocity0/w*sin(w*time);
    response(i,3) = -displ0*w*sin(w*time) + velocity0*cos(w*time);
end
```

Tasks	*	User-defined code	MATLAB library
Setup array response (npoints,3).	*	In sdof.m	zeros()
Compute circular natural freq w.	*		sqrt()
	*		
Compute y(t) and v(t).	*	In sdof.m	sin()
	*		cos()
	*		
Plot y(t) and v(t) versus t.	*	In sdof.m	plot()
Hold the graphics.	*		hold
Label the horizontal axis.	*		xlabel()
Label the vertical axis.	*		ylabel()
Add a grid to the plot.	*		grid
Add a plot title.	*		title()

Table 2.9. Mapping of Tasks to MATLAB Code

systematically walks along the rows of the `response` array and evaluates the time t, displacement $y(t)$, and velocity $v(t)$ for columns one through three of `response`. The execution speed will be rather slow because the looping construct is interpreted. The command

```
plot( response(:,1), response(:,2) );
```

draws the contents of column two in array `response` versus column one. In other words, a plot of $y(t)$ versus t is drawn. The graph of $v(t)$ versus t is generated with

```
plot( response(:,1), response(:,3) );
```

A far more efficient way of computing and storing the system response is with the commands

```
time     =  0.0:0.02:10;
displ    =  displ0*cos(w*time)   + velocity0/w*sin(w*time);
velocity = -displ0*w*sin(w*time) + velocity0*cos(w*time);
```

The first statement generates a (1x501) matrix called `time` having the element values 0, 0.02, 0.04 .... 10.0. The dimensions of matrices `displ` and `velocity` are inferred from the dimensions of `time` with the values of the matrix elements given by the evaluation of formulae on the right-hand side of the assignment statements. The required plots can be generated with

```
plot(time, displ);
hold;
plot(time, velocity);
```

The first statement creates a plot of vector `displ` versus vector `time`. The `hold` command places a hold on the current plot and all axis properties so that subsequent graphing commands may be added to the existing graph. The third statement creates the plot of $v(t)$ versus t.

Of course, the benefits of "fast evaluation of the SDOF formulae" and "convenient storage of the results in array response" can be combined. The block of code

```
time = [ 0:0.02:10 ]';
response(:,[1]) =  time;
response(:,[2]) =  displ0*cos(w*time)   + velocity0/w*sin(w*time);
response(:,[3]) = -displ0*w*sin(w*time) + velocity0*cos(w*time);
```

defines a (501x1) matrix called `time`, and then uses the second technique to compute (501x1) matrices of $x(t)$ and $v(t)$. The results of these calculations are assigned to the columns of response with submatrix notation

```
response(:,[k])
```

for the kth column of response.

Quadratic Equation Solver

Problem Statement. It is well known that the roots of a quadratic equation are given by solutions to

$$p(x) \;=\; ax^2 \;+\; bx \;+\; c \;=\; 0 \tag{2.21}$$

A number of solution cases exist. For example, when both a = 0 and b = 0, we consider the equation to be `extremely degenerate` and leave it at that. When $a = 0$ and $b \neq 0$, we consider the case `degenerate`; in this situation, the equations reduce to $p(x) = bx + c = 0$, which has one root. Otherwise, we have

$$\text{Roots} \;=\; \left[\frac{-b \pm \sqrt{b^2 \;-\; 4ac}}{2a}\right] \tag{2.22}$$

The term $b^2 - 4ac$ is known as the discriminant of the quadratic equation.

Write a program that will interactively prompt a user for the equation coefficients a, b, and c at the keyboard, and then compute and print its roots along with an appropriate message for the type

of solution. Appropriate messages are `extremely degenerate`, `degenerate`, `two real roots`, and `two complex roots`. For example, if $a = 1$, $b = 2$, and $c = 3$, the computer program should print

$$\text{Two complex roots} : \text{Root } 1 = -1.0000 + 1.414214\, i \tag{2.23}$$

$$\text{Root } 2 = -1.0000 - 1.414214\, i \tag{2.24}$$

Pseudocode. Because this problem contains numerous details and types of solutions, we refine our pseudocode over several iterations of development.

Iteration 1. A first draft pseudocode version for this task might look similar to the following:

```
announce the quadratic equation solving program.
request a, b, and c from the user.

calculate the discriminant.
if the discriminant is >= 0 then the roots are real:
   compute and print real roots x1 and x2
else the roots are complex
   the real part is -b/2a
   the imaginary part is the square root of ( the
   negative of the discriminant ) / 2a

print the results.
```

Iteration 2. Using successive refinement we can develop this pseudocode further into

```
print 'quadratic equation solving program'
print 'Enter the coefficients a, b, and c'
read the coefficients

discriminant = b^2 - 4ac
if the discriminant is >= 0 then equation has real roots x1, x2.
   x1 = -b/2a - sqrt( discriminant) / 2a
   x2 = -b/2a + sqrt( discriminant) / 2a
else, the discriminant is < 0, and roots are complex numbers
   (re, im) with
   re = -b/2a
   im = sqrt( - discriminant) / 2a
   print 'Complex roots'
```

Iteration 3. Iteration 2 still does not look like MATLAB code so at least one more stage of development is desirable

```
print 'quadratic equation solving program'
print 'Enter the coefficients a, b, and c'

input coefficients a, b, c
if( a is equal to zero )
    compute roots to determinate equations
    exit

discriminant = b^2 - 4ac
if (discriminant >= 0.0)    % The roots are real
   root1 = -b/2a - sqrt( discriminant )/2a
   root2 = -b/2a + sqrt( discriminant )/2a
   print 'The first  root is ' root1
   print 'The second root is ' root2
else                        % The roots are complex
   re = -b/2a
   im = sqrt( -discriminant )/2a
   print 'The first  complex root is', re, im
   print 'The second complex root is', re, -im
```

This version of pseudocode is very close to MATLAB, and we are ready to begin writing the MATLAB code for our quadratic equation solver.

Program Source Code. The source code for Program 2.5 is contained in two files, a script M-file called `quadratic.m` and a function M-file called `discriminant.m`. Here are the details.

Computer Program 2.5 : Compute Roots of Quadratic Equation

```
% ========================================================================
% quadratic.m -- Coefficients are read in from keyboard
%                Roots of Quadratic are printed to screen.
%
% Note : Naive implementation of quadratic equation solver. This algorithm
%        does not take into account possible loss of accuracy when two
%        floating point numbers of almost equal size are subtracted.
%
% Written By : Mark Austin                                    July 1997
% ========================================================================

% Print Welcome Message

  disp('Welcome to the Quadratic Equation Solver (Version 1)');
  disp('=================================================');
```

```
% Prompt User for Coefficients of Quadratic Equation

  disp('Please enter coefficients for equation a.x^2 + b.x + c');
  A = input ('Enter coefficient a : ');
  B = input ('Enter coefficient b : ');
  C = input ('Enter coefficient c : ');

% Print Quadratic Equation to Screen

  fprintf('Equation you have entered is : %g.x^2 + %g.x + %g\n', ...
          A, B, C);

% Compute Roots of simplified equations : A equals zero

  RootsFound = 0;
  if A == 0 & B == 0,
     fprintf('Cannot solve extremely degenerate equation' );
     fprintf('%14.8g = 0.0\n', C );
     RootsFound = 1;
  end;

  if A == 0 & B ~= 0 & RootsFound == 0,
     Root1 = - C/B;
     fprintf('Degenerate root : Root = %14.8g\n', Root1 );
     RootsFound = 1;
  end;

% Compute Roots of Quadratic Equation : A not equal to zero

  if RootsFound == 0,
     Discriminant = discriminant(A,B,C);   % Compute discriminant of
                                           % quadratic equation.
     if Discriminant >= 0,                 % Case for two real roots

        Root1 = -B/2.0/A - sqrt( Discriminant )/2.0/A;
        Root2 = -B/2.0/A + sqrt( Discriminant )/2.0/A;

        fprintf('Two real roots : Root1 = %14.8g\n', Root1 );
        fprintf('               : Root2 = %14.8g\n', Root2 );
     else                                  % Case for complex roots

        fprintf('Two complex roots : Root1 = %14.8g + %14.8g i\n', ...
                -B/2.0/A,  sqrt( -Discriminant )/2.0/A);
        fprintf('                  : Root2 = %14.8g + %14.8g i\n', ....
                -B/2.0/A, -sqrt( -Discriminant )/2.0/A);
     end;
  end;
```

```
function  [ discrim ] = discriminant( A, B, C )

% DISCRIMINANT : Compute discriminant in quadratic equation.
%

discrim = B*B - 4*A*C;
```

Running the Program. The following script of code

```
>> quadratic
Welcome to the Quadratic Equation Solver (Version 1)
====================================================
Please enter coefficients for equation a.x^2 + b.x + c

Enter coefficient a : 1.2

Enter coefficient b : 3.4

Enter coefficient c : 5.6
Equation you have entered is : 1.2.x^2 + 3.4.x + 5.6
Two complex roots : Root1 =     -1.4166667 +      1.6308655 i
                  : Root2 =     -1.4166667 +     -1.6308655 i
>>
```

shows a typical session of I/O for Program 2.5. You should verify that this solution is correct by substituting a = 1.2, b = 3.4, and c = 5.6 into Equation (2.22).

Although this example does not demonstrate it, a key limitation of this program is the absence of checking for loss of numerical accuracy that occurs when two floating point numbers of almost equal size are subtracted. Situations of this type arise, for example, when a = 1.0, b = 1000000.0, and c = 1.0. An algorithm for overcoming this problem is explained in Problem **??** of the C tutorial.

Program Architecture. Table 2.10 shows how key tasks in the quadratic equation solution procedure have been mapped to user-defined MATLAB code and MATLAB function calls.

Other than than the discriminant computation in `discriminant.m`, the heart of our quadratic equation solver is contained in `quadratic.m`. When MATLAB executes `quadratic.m` for the first time and encounters the reference to a function `discriminant`, the function M-file `discriminant.m` will be located and compiled into MATLAB's memory.

Tasks	*	User-defined Code	MATLAB Library
Announce quadratic program.	*	In quadratic.m	disp()
Read equation coefficients.	*	In quadratic.m	input()
	*		
Compute discriminant.	*	In discriminant.m	
Square root computation.	*	In quadratic.m	sqrt()
Solve quadratic equation.	*	In quadratic.m	
Print roots of quadratic equation.	*		fprintf()
	*		

Table 2.10. Mapping of Tasks to MATLAB Code

The MATLAB function `disp()` displays messages enclosed within single quotes. `fprintf()` displays messages containing formatted output. The function `input()` prompts a user for keyboard input (there is no point in reinventing the wheel). Finally, we employ the MATLAB function `sqrt()` for the square root calculation in Equation (2.22).

Analysis of the MATLAB Code. The script M-file, `quadratic.m`, contains the commands needed to compute and print the roots of the quadratic equation. The function M-file, `discriminant.m`, contains the function

```
function  [ discrim ] = discriminant( A, B, C )
```

accepting three matrix arguments, and returning a matrix

```
discrim = B*B - 4*A*C;
```

containing the equation discriminant. We implicitly assume in this function declaration that `A`, `B`, and `C` will be one-by-one matrices. The execution of `discriminant` is terminated by an EOF.

Statistical Analysis of Experimental Data

Problem Statement. Suppose that the concentration of spores of pollen per square centimeter are measured over a 15-day period and stored in a data file `expt.dat`.

```
1  12
2  35
3  80
4  120
```

```
 5  280
 6  290
 7  360
 8  290
 9  315
10  280
11  270
12  190
13  90
14  85
15  66
```

The first and second columns of `expt.dat` store the "day of the experiment" and the "measured pollen count," respectively.

Write a MATLAB program that will read the contents of the data file into an array and create and label a two-dimensional bar plot showing the "pollen count" versus "day." The program should then compute the mean and standard deviation of the pollen count, and plot and label dashed lines for the mean pollen count and the mean pollen count $\pm$ one standard deviation.

Pseudocode. We begin our analysis by noting that the mean and standard deviation of the experimental data can be computed by the function `stat()` in Program 2.2. Hence, the pseudocode for this problem is

```
read the contents of file expt.dat into the array expt.

draw and label the bar chart.

call the function stat() to compute the mean and standard
deviation of the experimental data.

construct a working array of coordinate points for plotting the
dashed lines -- horizontal lines are required for:
    mean value - 1 standard deviation.
    mean value alone.
    mean value + 1 standard deviation.
```

Program Source Code. Here is the program source code.

Computer Program 2.6 : Statistical Analysis of Experimental Data

```
%  ====================================================
%  expt.m -- Statistical analysis of experimental data.
%
%  Written By : Mark Austin                   July 1997
%  ====================================================

% Store experimental results in array

  load expt.dat

% Generate bar plot of experimental results

  bar(expt(:,1), expt(:,2),'b')
  xlabel('Day of Expt');
  ylabel('Pollen Count');

% Compute terms from experimental results.

  [xm, xd] = stat(expt(:,2))

% Create and display mean value of pollen count

  mean_minus = xm(1,1) - xd(1,1);
  mean_plus  = xm(1,1) + xd(1,1);

  data = [  1, xm(1,1), mean_minus, mean_plus;
           15, xm(1,1), mean_minus, mean_plus ];

  hold;
  plot (data(:,1), data(:,2), 'b');
  plot (data(:,1), data(:,3), 'b:');
  plot (data(:,1), data(:,4), 'b:');

  text(1,   xm(1,1) + 10,'Mean Pollen Count');
  text(1,mean_minus + 10,'Mean - Std');
  text(1, mean_plus + 10,'Mean + Std');
```

Running the Program. The script of code

```
>> format compact
>> expt
  xm =
     184.2000
  xd =
     114.2309
>>
```

shows the command needed to run Program 2.6 and the textual output that is generated.

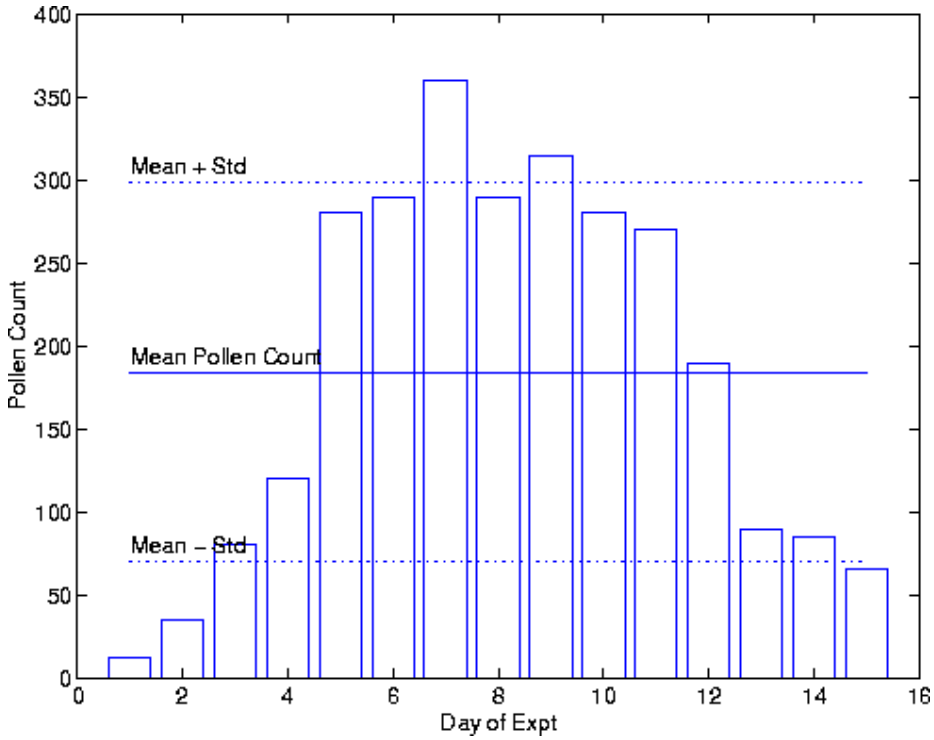


Figure 2.6. Pollen count versus day of experiment.

Figure 2.6 shows the bar chart of pollen count measurements and mean/standard deviation statistics. Solid- and dashed-line summaries of the mean data value and the mean value $\pm$ one standard deviation are then superimposed on the bar chart.

Program Architecture. Table 2.11 shows how tasks in the problem statement have been mapped to user-defined MATLAB code and calls to built-in MATLAB functions.

Tasks	*	User-defined code	MATLAB library
	*		
File holding the experimental data.	*	expt.dat	
	*		
	*		
Read the file expt.dat into an array "expt".	*	In expt.m	load
	*		
	*		
Draw the bar chart.	*	In expt.m	bar()
Label the horizontal axis.	*		xlabel()
Label the vertical axis.	*		ylabel()
	*		
Compute the mean and std of the experimental data.	*	In stat.m	
	*		
	*		
Hold the graphics.	*	In expt.m	hold
Draw the dashed horizontal lines.	*		plot()
Label the dashed lines.	*		text()

Table 2.11. Mapping of Tasks to MATLAB Code

After the script M-file `expt.m` has loaded the contents of `expt.dat` into an array `expt` containing two columns and created a bar chart of the data, the function M-file `stat.m` is called to compute the mean and standard deviation of the experimental data.

Analysis of the MATLAB Code. The command

```
load expt.dat;
```

loads the content of data file `expt.dat` into the array `expt`. Then the command

```
[xm, xd] = stat( expt(:,2) )
```

calls the user-defined function `stat` defined in `stat.m` to compute the mean and standard deviation of data values stored in the second column of `expt`. The mean and standard deviation

are represented by `xm` and `xd`, both one-by-one matrices. A two-by-four array, `data`, holds the coordinate values of the solid- and dashed-line segments for plotting purposes.

Otherwise, Program 2.6 relies on a variety of function calls to the MATLAB graphics library to draw and label the bar chart. A detail description of these functions and their capabilities are found in Chapter 14.

Further Information

1. The MATLAB program comes with a lot of M-file examples. To find their location on your computer, type the MATLAB command `path`. This will lead you to some really nifty demos.

2.9 Review Questions

1. What does the `helpdesk` command do?
2. What is the maximum array size supported by the student edition of MATLAB Version 5.
3. MATLAB output can be rather lengthy. What is an easy way of shortening the length of the output?
4. What is the purpose of the `eps` constant?
5. What does the `input` function do?
6. Briefly explain how "precedence of arithmetic operators" works in MATLAB.
7. Explain step by step how the arithmetic expression

```
>> 1/2*( 2 + 3*4^2 )
```

   is evaluated in MATLAB.

8. Consider the script of code

```
>> ix = 1;
>> ij = 2*ix;
>> ik = 2*(ix==1) + (3*ij ~= 6);
```

   What is the value of `ik`?

9. By default, MATLAB prints floating point numbers to four decimal places of accuracy. How would you adjust this option?
10. What are the three ways of defining a matrix in MATLAB?

11. What is the output generated by the MATLAB commands?

```
>> A = [ 1 2; 3 4 ];
>> B = [ A 2*A ]
```

12. What is the output generated by the sequence of commands?

```
>> x = [ 1, 2, 3; 4, 5, 6 ];
>> sin(2*x);
```

13. What does the command

```
>> x = -10.0: 0.2: 10.0
```

do?

14. What is the output generated by the sequence of commands?

```
>> x = [ -pi : pi/2 : pi ];
>> sin(x).^2 + cos(x).^2
```

15. If A is a $(m \times n)$ matrix and B is a $(r \times p)$ matrix, what restrictions must exist on m, r, n, and p for the matrix sum C = A + B to be defined?

16. If A is a $(m \times n)$ matrix and B is a $(r \times p)$ matrix, what restrictions must exist on m, r, n, and p for the matrix product C = A · B to be defined?

17. Suppose that

```
>> y = [ 1 2 3 ; 4 5 6; 7 8 9 ];
```

How would you use the `for` looping construct to compute the sum of the elements in matrix y?

How would you use the `sum()` function to sum the matrix elements in y (a one line answer will suffice)?

18. Why does the fragment of code

```
>> y = zeros(1:1000);
>> for i = 1:1000
       y(i) = 2*sin(i);
   end
```

execute slower than

```
>> y = 2*sin([1:1000]);
```

19. Suppose that a matrix `data` contains

```
data = [ 1.5  1.0  3.0;
         6.5 -1.2 12.4;
         2.5 -1.0  3.8;
         2.4  8.1  5.8 ];
```

How would you use MATLAB to compute the maximum matrix element value in each row and column of matrix `data`?

2.10 Programming Exercises

2.1 **Beginner.** Figure 2.10 shows a mass m resting on a frictionless surface. The mass is connected to two walls by springs having stiffnesses k_1 and k_2.

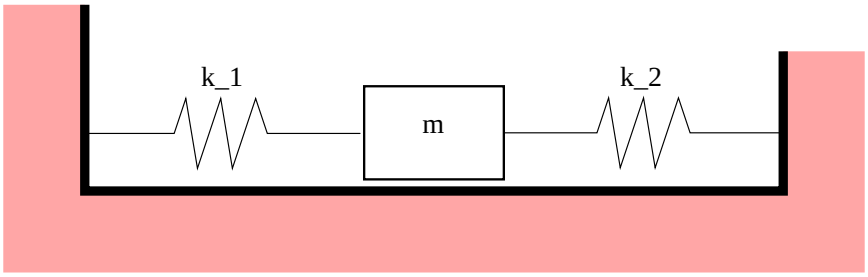


Figure 2.7. Mass-spring system

The natural period of the mass-spring system is:

$$T = 2\pi\sqrt{\frac{m}{k_1 + k_2}} \tag{2.25}$$

Write a MATLAB program that will prompt a user for m, k_1, and k_2, check that the supplied values are all greater than zero, and then compute and print the natural period of the mass-spring system.

2.2 **Beginner.** Suppose that during squally conditions, regular one second wind gusts produce a forward thrust on a yacht sail corresponding to

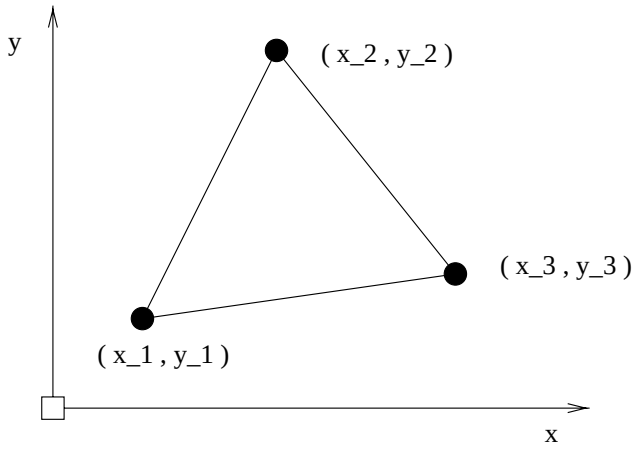
$$F(t) = \begin{cases} 4 + 15 \cdot t - 135 \cdot t^3 & 0.0 \leq t \leq 0.3, \\ (731 - 171t)/140 & 0.3 < t \leq 1.0. \end{cases} \tag{2.26}$$

F(t) has units kN. Write a MATLAB program that computes and prints $F(t)$ for $0 \leq t \leq 3$ seconds. The program output should look something like the following:

```
     Time          Thrust
(seconds)            (kN)
========================
```

```
0.00          4.00
0.25          5.64
0.50          4.61
0.75          4.31
1.00          4.00
1.25          5.64
1.50          4.61
1.75          4.31
2.00          4.00
2.25          5.64
2.50          4.61
2.75          4.31
3.00          4.00
```

2.3 **Beginner.** The adjacent figure shows a triangle defined by the vertex coordinates (x_1, y_1), (x_2, y_2), and (x_3, y_3).



Write a MATLAB program that will

1. Interactively prompt a user for the x and y coordinates at each of the triangle vertices.
2. Print the triangle vertices in a tidy table.
3. Compute and print the area of the triangle (make sure that it is positive) and its perimeter.

2.4 **Beginner.** Figure 2.8 is a schematic of an irregular polygon having seven sides. Suppose that the x and y vertex coordinates are stored as two columns of information in the file `polygon.dat`.

```
1.0  1.0
1.0  5.0
```

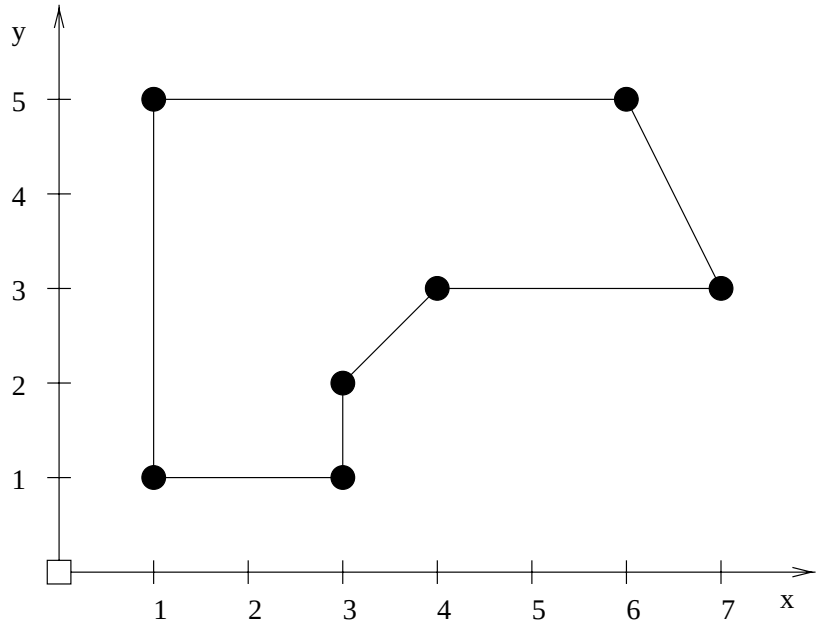


Figure 2.8. Seven-sided irregular polygon

```
6.0  5.0
7.0  3.0
4.0  3.0
3.0  2.0
3.0  1.0
```

Write a MATLAB script file that will read the contents of `polygon.dat` into an array `polygon`, and then compute and print

1. The minimum and maximum polygon coordinates in both the x and y directions
2. The minimum and maximum distance of the polygon vertices from the coordinate system origin
3. The perimeter and area of the polygon

Note. You should use the `max()` and `min()` functions provided by MATLAB for Part 2. In Part 3, you should use the fact that the vertices have been specified in a clockwise manner.

2.5 **Beginner.** The single-payment compound-interest factor,

$$F = P \cdot [1 + i]^n \tag{2.27}$$

defines the future value of a present day investment `P` that earns an interest rate `i` for `n` years. During the first year `P` dollars grows to `P.(1+i)` dollars. In the second year, `P.(1+i)` dollars grows to `P.(1+i).(1+i)` dollars, and so forth. The term $(1 + i)^n$ is called the compound-interest factor.

Write a MATLAB program that will prompt a user for `P`, `i` and `n`. Use your program to demonstrate that when $1,000 is invested at 12% for 4 years, the future sum is $1,573.50.

2.6 **Beginner.** Suppose that a savings bank offers a tiered rate of interest that increases with the account balance as follows:

$$\text{Interest Rate(balance)} = \begin{cases} 5\% & \$0.0 \leq balance \leq \$5,000, \\ 8\% & \$5,000 < balance \leq \$10,000, \\ 10\% & \$10,000 < balance, \end{cases} \tag{2.28}$$

Write a MATLAB program to:

1. Draw a plot of "rate of interest" versus "balance" for account balances up to $25,000.

2. Suppose that a customer deposits $6,000 for 20 years. Compute and print the compound balance for years 1 through 20.

Note. When the interest rate is not constant over `n` years, the term $(1 + i)^n$ in Equation 2.27 is replaced by $(1 + i_1) \cdot (1 + i_2) \cdots (1 + i_n)$.

2.7 **Beginner.** Leibnez's series is given by:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} + \cdots \tag{2.29}$$

Write a MATLAB script file to compute Leibniz's series summation for 1000 terms. First, use a for-loop construct to compute the series summation. Repeat the experiment using an

array for the series coefficients, appropriate matrix element-level operations for the alternating signs. and `sum()` for the summation of series terms.

Use the MATLAB facilities `tic` and `toc` to monitor the time needed to compute each implementation. How does the relative speed of the two methods vary as a function of the number of terms in the series?

2.8 **Intermediate.** Using no more than three lines of MATLAB code, and no looping constructs, demonstrate that the series summation:

$$S = \frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \frac{1}{3 \cdot 4} + \frac{1}{4 \cdot 5} + \cdots \tag{2.30}$$

approaches 1.

2.9 **Beginner.** An efficient way of computing the cube root of a number `N` is to compute the root of

$$f(x) = x^3 - N = 0 \tag{2.31}$$

with the method of Newton Raphson, namely:

$$x_{n+1} = x_n - \left[\frac{f(x_n)}{f'(x_n)}\right] \tag{2.32}$$

Substituting Equation (2.31) into Equation (2.32) and rearranging terms gives the recursive relationship

$$x_{n+1} = \frac{1}{3} \cdot \left[\frac{2x_n^3 + N}{x_n^2}\right] \tag{2.33}$$

Write a MATLAB program that will

1. Prompt the user for a number `N`, and compute the cube root of `N` via Equation (2.33). The details of the cube root calculation should be contained within a function M-file called `cuberoot.m`. A suitable function declaration is

```
[ answer ] = cuberoot ( N )
```

Your M-file function should use Equation (2.14) for a test on convergence.

2. Print the number `N`, its cube root, and the number of iterations needed to compute the result.

`Hint:` Your solution to this problem should be similar to Program 2.1.

2.10 **Intermediate.** Write and test a function M-file `matadd.m` for the element-by-element addition of matrices `A` and `B` using Equation (2.6). An appropriate function declaration is

```
[ matrixsum ] = matadd ( A, B )
```

After your M-file function has checked that matrices `A` and `B` have compatible sizes, Equation (2.6) should be evaluated inside a set of 2 nested for loops.

Write a test program to allocate and initialize matrices `A` and `B`, and compute their sum using the `matadd` function and MATLAB's built-in library for matrix addition. What is the relative speed of these two approaches?

2.11 **Intermediate.** Repeat the experiment described in Problem **??** but for the multiplication of two matrices `A` and `B`. An appropriate function declaration is

```
[ matrixproduct ] = matmult ( A, B )
```

Again, the matrix multiplication should be computed with a set of 2 nested for loops.

2.12 **Intermediate.** Monte Carlo methods solve problems by experiments with random numbers on a computer. They have been around since about the mid-1940s. A relatively straightforward way of estimating π is to conduct an experiment where darts are randomly thrown at a square board of side length `D`, as shown in Figure 2.10.

Given that the area of the circle $A_{circle} = [\pi D^2/4]$, and the area of the square $A_{square} = D^2$, then

$$\pi = 4 \cdot \left[\frac{A_{circle}}{A_{square}} \right] \tag{2.34}$$

In the Monte Carlo experiment, N darts are thrown at the board. Let $P_i = (x_i, y_i)$ be the coordinate point of the i^{th} dart (i = 1,2,3 ... N). Point P_i is inside the circle if

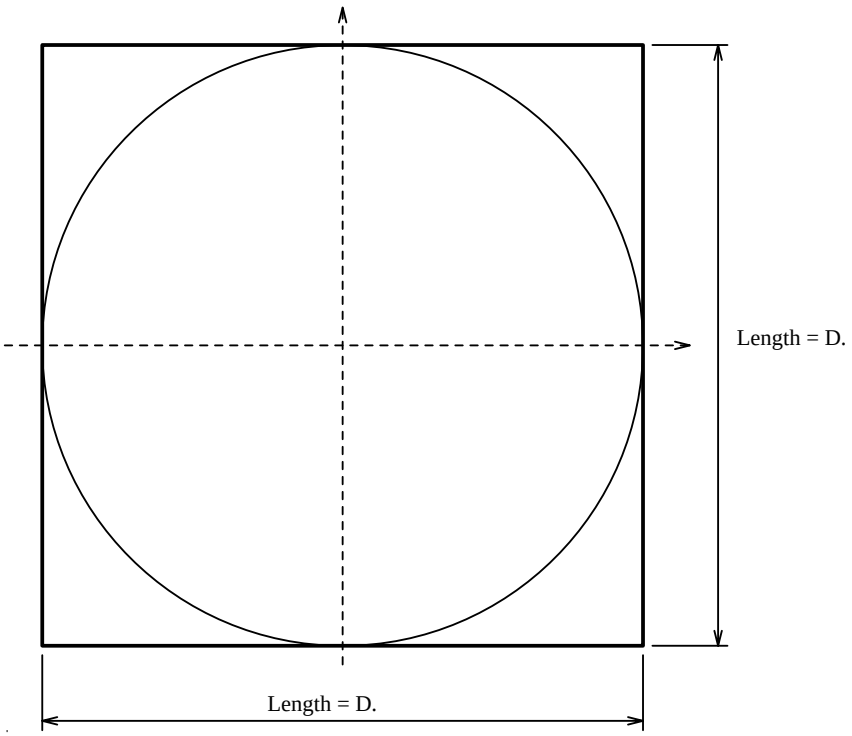


Figure 2.9. Estimating π via integration

$$x_i^2 + y_i^2 \leq \left[\frac{D^2}{4}\right] \tag{2.35}$$

If at the conclusion of the experiment X darts have landed inside the circle (X $\leq$ N), then a Monte Carlo estimate of π is

$$\pi \sim 4 \cdot \left[\frac{X}{N}\right] \tag{2.36}$$

Write a MATLAB program that will

1. Simulate this experiment by generating pairs of uniformly distributed random numbers, one for the x-coordinate and a second for the y-coordinate. The random numbers should

be scaled so that they cover the interval `[ -D/2, D/2]`. Calculate whether each data point P_i lies inside the circle. At the conclusion of the experiment, sum the number of darts that lie inside the circle and evaluate Equation (2.36).

2. Print the total number of trials in the experiment (N), the total number of trial points inside the circle (X), and the Monte Carlo estimate of π.

3. Draw Figure 2.10 together with the (x, y) coordinates of the dart throwing experiment.

2.13 **Intermediate.** Extend the functionality of Program 2.1 so that it will compute the square root of matrix elements in matrices containing only positive matrix elements. For example, the function call

```
>> sqroot( [ 1 2 3; 4 5 6; 7 8 9 ] )
```

should generate the output:

```
ans =

    1.0000    1.4142    1.7321
    2.0000    2.2361    2.4495
    2.6458    2.8284    3.0000
```

When one or more of the matrix elements is not positive, `sqroot.m` should print an appropriate error message and terminate its execution.

2.14 **Intermediate.** A key limitation of the quadratic equation solver in Program 2.5 is the absence of checking for loss of numerical accuracy due to subtraction cancelation. We note from Equation (2.22) that this will occur whenever the quadratic equation has two real roots, and either coefficient `a` or coefficient `c` (or both) is very small compared to coefficient `b`.

Rather than naively applying Equation (2.22) directly, numerical accuracy can be improved by computing the quantity

$$\mathrm{Q} \;=\; -0.5 \cdot \left[b + \mathrm{sign(b)} \cdot \sqrt{b^2 \;-\; 4ac} \right] \tag{2.37}$$

where `sign(b)` is a function that gives `1` for positive `b` and `-1` for negative `b`. Equation (2.37) follows from multiplying Equation (2.22) by the unity fraction

$$\left[\frac{-b \mp \sqrt{b^2 \;-\; 4ac}}{-b \mp \sqrt{b^2 \;-\; 4ac}} \right] \tag{2.38}$$

The roots of the quadratic are `c/Q` and `Q/a`, respectively. Extend Program 2.5 so that it avoids numerical errors due to subtractive cancellation. Develop some testcase problems to show that your improved implementation works even when Program 2.5 provides inferior solutions.

Chapter 3

MATLAB Graphics

Graphics are an indispensable part of engineering education and professional practice because they provide insight into the complicated relationships that exist between multi-dimensional scientific and engineering phenomena. One of the neat features of MATLAB is its graphics capabilities. Using only a few simple MATLAB commands, two- and three-dimensional plots and subplots can be created, with labels and titles, axes, and grids.

3.1 Simple Two-Dimensional Plotting

MATLAB has an ensemble of functions for the simple two-dimensional plotting of functions and data that take the form $(x_i, y_i),\ i = 1, 2, 3, \cdots, n$. They are

- **plot(x,y)** Draws a linear plot of vector y versus vector x.
- **semilogx(x,y)** Draws a plot of vector y versus vector x, using a logarithmic scale for x and a linear scale for y.
- **semilogy(x,y)** Draws a plot of vector y versus vector x, using a logarithmic scale for y and a linear scale for x.
- **loglog(x,y)** Draws a plot of vector y versus vector x, using logarithmic scales for both x and y axes.

In each of these function calls, the contents of x are the data points along the horizontal axis of the plot. The data along the vertical axis of the plot is stored in a second vector y. Vectors x and y should have the same length. If x or y is a matrix, then the vector is plotted versus the rows or columns of the matrix, whichever line up.

By default, MATLAB will plot each data point pair on a set of axes scaled to cover the range of values in x and y, and connect the marked data points with straight line segments. The latter helps to highlight trends implied by the sequence of data point pairs.

For function calls that involve logarithmic transformations, it is important to remember that the logarithm of `x` is undefined for negative `x`. MATLAB will handle these accidental oversights by printing an error message indicating that data points have been omitted from a plot.

Coordinate labels, plot titles, grid, and textual messages can be added to simple two-dimensional plots with

- **grid on** Add grid lines to a plot. The command `grid off` takes them off. The command `grid` by itself toggles the grid state.
- **xlabel('x axis label')** Add text beside the x axis on the current axis.
- **ylabel('y axis label')** Add text beside the y axis on the current axis.
- **title('title of plot')** Add text at the top of the current axis.
- **text(x, y, 'text')** Add 'text' string to a plot where (x, y) is the coordinate of the center leftedge of the character string taken from the plot axes.
- **gtext('text')** Activate the use of the mouse to position a cross-hair on the graph at which point the 'text' will be placed when any key is pressed.

Example. Draw the graph of

$$y(x) = \left[\frac{\sin(2x)}{2x}\right] \text{ for } -10 \leq x \leq 10 \tag{3.1}$$

The sequence of MATLAB commands

```
>> x = -10 : 0.2 : 10;
>> y = 1/2*sin(2.*x)./x;
Warning: Divide by zero
>> plot(x,y)
>> grid
>> xlabel('x')
>> ylabel('sin(2x)/2x')
```

generates Figure 3.1.

Vector `x` is a (1×101) matrix holding the coordinates of domain `[-10,10]` partitioned into intervals of 0.2, and `y` is a vector of the same length giving the values of $y = \sin(2x)/2x$ within

the partition. Equation (3.1) is evaluated successfully for each of the elements in matrix `x` except `x(51)` = 0, where a "divide by zero" occurs. Does this make sense? By writing a Taylor series expansion for $\sin(2x)$ and dividing through by $2x$, it is relatively easy to show that y(0) = sin(0)/0 = 1. MATLAB does not pick up on this point. Instead, the element of vector `y` corresponding to x = 0 is assigned `NaN`, and the `plot()` function treats the array element $y(51)$ as missing data (see the upper sections of Figure 3.1).

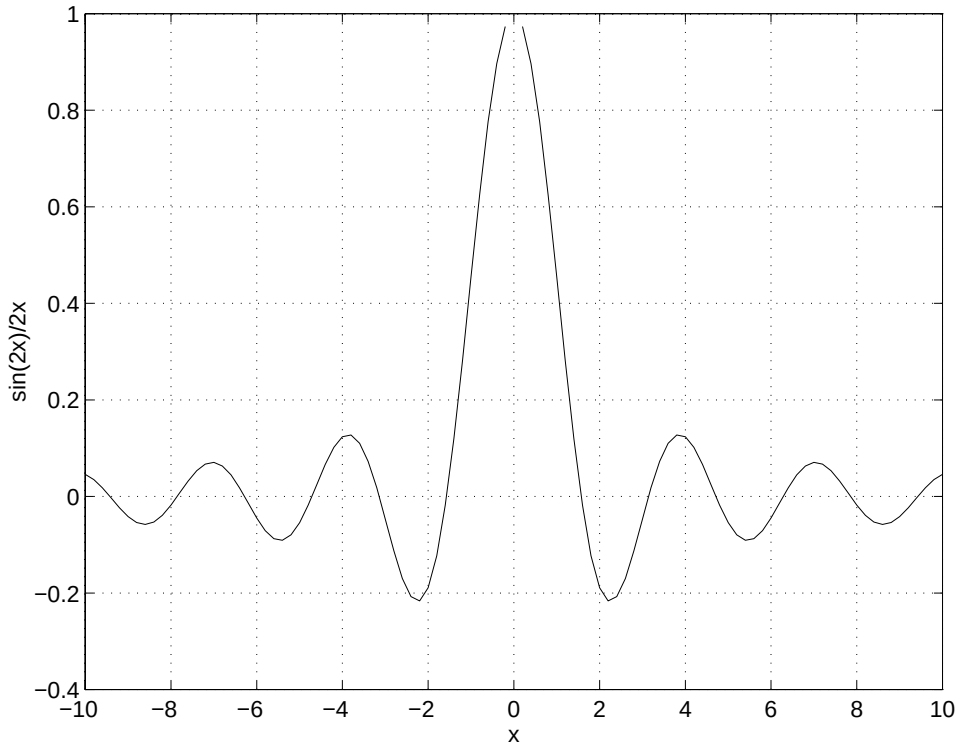


Figure 3.1. y = sin(2x)/2x for -10 $\leq$ x $\leq$ 10

Note. One way of "mitigating" this problem is to replace the zero element(s) in matrix `x` with `eps`, that is

```
>> x = -10 : 0.2 : 10;
>> x = x + (x==0)*eps
```

In the second MATLAB command, a component of matrix `x` will be incremented by `eps` when the logical expression `(x==0)` evaluates to true. Otherwise, the components of `x` remain unchanged.

Now the `y` matrix is generated without error, and `sin(0)/0` evaluates to `1`.

Axes of Plots. In Figure 3.1, MATLAB has automatically fixed the range of `x` and `y` values so that the plotted function fills the space that is available (i.e., xmin = min(x), xmax = max(x)). In doing so, the "missing data item" at y(0) has been unintentionally hidden by the border of Figure 3.1. Perhaps it would be more evident if the range of `y` values in Figure 3.1 were rescaled to cover the interval `[ -0.4 , 1.2]`?

MATLAB provides the function `axis` to control the scaling and appearance of axes in a plot. In this tutorial, we make frequent use of the function call

- **axis([xmin, xmax, ymin, ymax])** Set the range of `x` and `y` values in the current plot to `[ xmin, xmax]` and `[ ymin, ymax]`, respectively.

In addition, the command `axis('equal')` changes the current axis box size so that equal tick mark increments on the x- and y-axis are equal in size. Similarly, the command `axis('off')` turns off all axis labeling and tick marks. Axis labeling and tick marks is turned back on again with `axis('on')`.

Thus, the sequence of MATLAB commands

```
>> x = -10 : 0.2 : 10;
>> x = x + (x==0)*eps;
>> y = 1/2*sin(2.*x)./x;
>> plot(x,y)
>> axis([ -10 10 -0.4 1.2 ])
>> grid;
>> xlabel('x')
>> ylabel('sin(2x)/2x')
```

generates a graph nearly identical to Figure 3.1, in this case with the range of `x` and `y` values covering `[-10,10]` and `[-0.4,1.2]`, respectively, and with the correct evaluation of $\sin(2x)/2x$ at $x = 0$.

Parametrically Defined Curves. Plots of parametrically defined curves can also be made. For example, a plot of the parametric curve

$$[\text{x(t)}, \text{y(t)}] = [\cos(3t), \sin(2t)] \text{ for } 0 \leq t \leq 2\pi \tag{3.2}$$

can be created with the MATLAB commands

Line type	solid (-),	dashed (-),	dotted (:),	dashdot (-.)
Mark type	point (.),	plus (+),	star (*),	circle (o), x-mark (x).
Color	yellow (y),	magenta (m),	cyan (c),	red (r), green (g), blue (b), white (w), black (k).

Table 3.1. Summary of Line Types and Line Colors

```
>> t = 0: 0.02: 2*pi;
>> plot( cos(3*t), sin(2*t) )
>> grid
>> axis('equal')
```

The result is shown in Figure 3.2. Notice how we have used the `axis('equal')` command to equalize the scales in the x- and y-axis directions. This makes the parametric plot take its true shape instead of an oval.

Setting Line and Mark Types, and Colors. As already demonstrated in Figures 3.1 and 3.2, MATLAB will automatically connect the data points in a plot by solid lines. When the data points are closely spaced, this gives the appearance of a smoothly drawn curve.

Table 3.1 contains the line and mark types, and color options, that can be used in MATLAB plots. For example, the script

```
>> t = 0: 0.04: 2*pi;
>> plot( t.*cos(3*t), 2.*t.*sin(2*t), 'o' )
>> grid
```

generates the pathway of `'o'` points shown in Figure 3.3. In this case, we have deliberately omitted the `axis('equal')` command so that the horizontal and vertical axes can be independently scaled to fill the rectangular space available on a page.

Combinations of line and mark types, and color settings, can be specified by simply bundling the combined settings into a text string enclosed by single quotes. For example, the command

```
>> plot( t.*cos(3*t), 2.*t.*sin(2*t), 'og' )
```

tells MATLAB to plot Figure 3.3 with green circles.

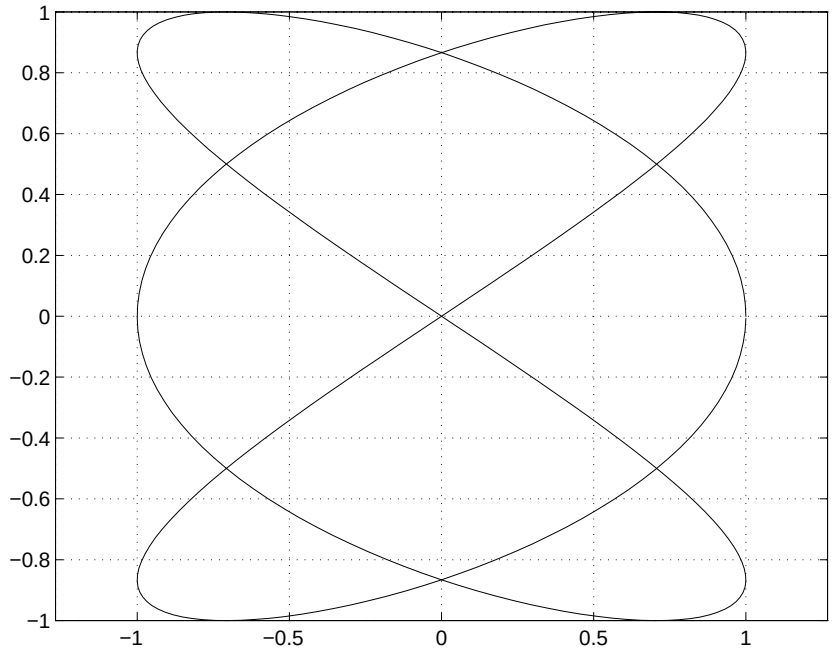


Figure 3.2. Parametric contour (x,y) = [cos(3t), sin(2t)]

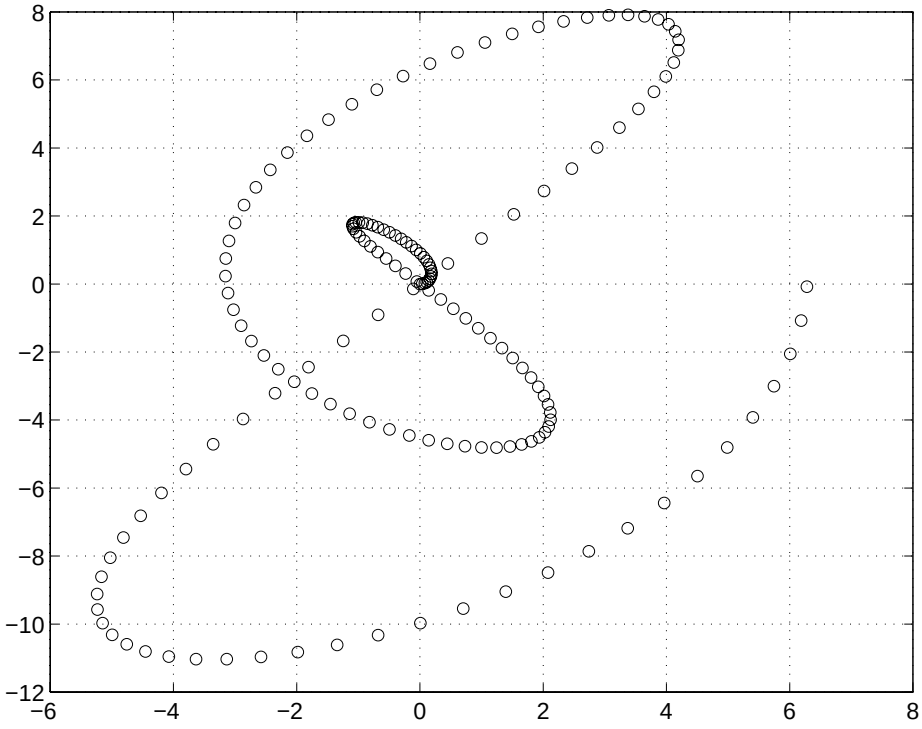


Figure 3.3. Parametric contour (x,y) = [t*cos(3t), 2t*sin(2t)]

Clearing Plots. The command `clf` deletes all objects from the current figure. The command `cla` deletes all objects (e.g., lines, text) from the current axes.

Histograms, Bar Charts, and Stem Diagrams

The function specifications that follow are a subset of MATLAB functions available for plotting histograms, bar charts, stair-step plots, and stem diagrams:

- **bar(x,y)** Draws a bar graph of the elements of vector `y` at locations specified in vector `x`. The x values must be ascending order.
- **hist(x,y)** Plot a histogram of the values in vector `y` using the bins specified in vector `x`.
- **stairs(x,y)** A stair-step graph is a bar graph without internal lines. This function call draws a stair-step graph of the elements of vector `y` at locations specified in vector `x`.
- **stem(x,y)** Plots the data sequence `y` as stems from locations specified in `x`. Each stem is terminated by a circle positioned at the data value.

For a complete list of options for each function, type

```
>> help function_name
```

(e.g., help bar) within MATLAB.

Example. The block of MATLAB code

```
>> x = 1:6;
>> y = zeros(1,6);
>> for i = 1:600
       ii      = ceil( 6*rand(1) );
       y(1,ii) = y(1,ii) + 1;
   end,
>> bar(x,y)
>> axis([0, 8, 0, 125 ]);
>> grid
>> xlabel('Number on Dice (1 through 6)');
>> ylabel('Number of scores in Experiment');
```

simulates an experiment where a regular die is thrown 600 times and the total number of scores, `1` through `6`, is counted and plotted as a bar chart (see Figure 3.4). You should observe that the sum of column heights 1 through 6 is 600 and that the average column height is 100.

Figure 3.5 is a graphical representation of the same die-throwing experiment, but in this case, the grid has been removed and the frequency of die scores is displayed as a stem chart.

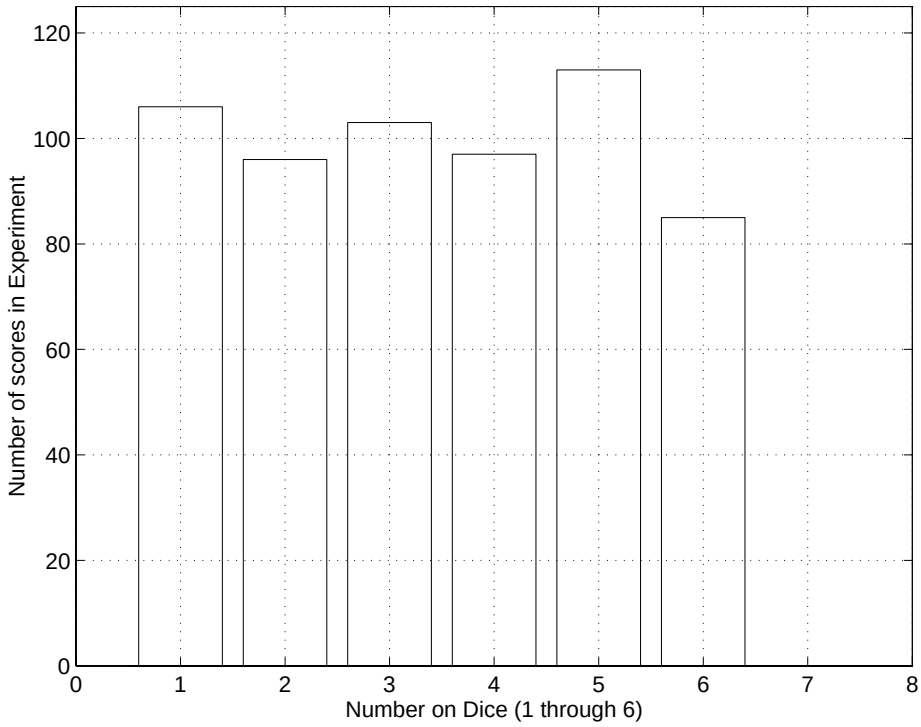


Figure 3.4. Bar chart of scores in die-throwing experiment

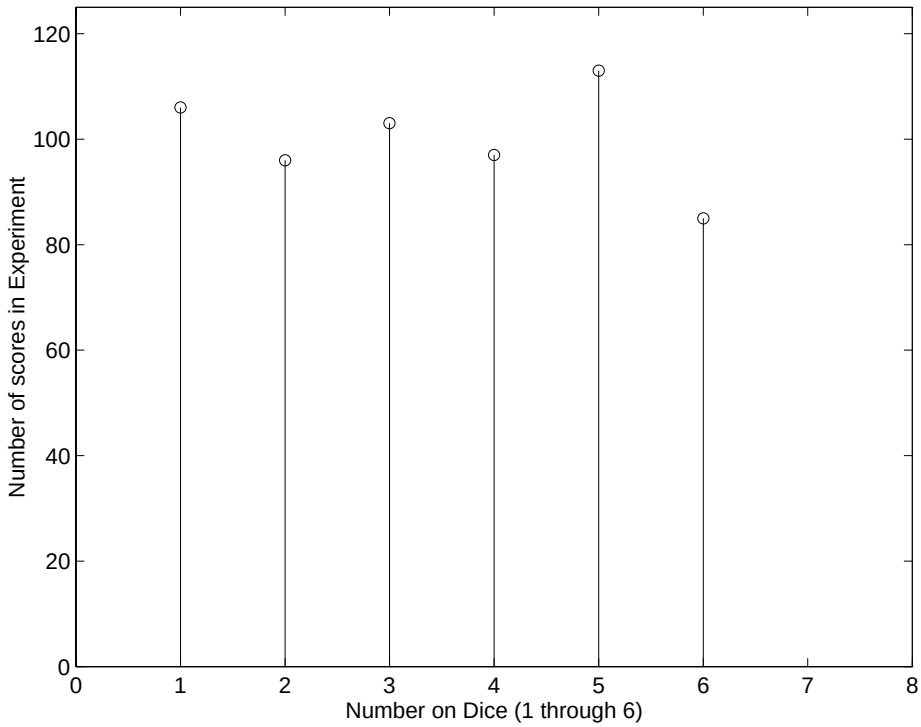


Figure 3.5. Stem chart of scores in die-throwing experiment.

Multiple Plots

A straightforward way of plotting multiple curves on the same graph is with the command

- **plot(x,y,w,z, ...)** Opens a graphics window and draws a linear plot of vector `y` versus vector `x`, and vector `w` versus vector `z` on the same graph. Again, if `x`, `y`, `w` or `z` is a matrix, then the vectors are plotted versus the rows or columns of the matrix, whichever line up.

MATLAB will plot the data set `y` versus `x` first, then `w` versus `z` and so forth. This strategy of implementation means that the length of vectors `x` and `y` need not be the same as `w` and `z`.

The following examples illustrate three ways of making multiple plots on a single graph, as shown in Figure 3.6.

Example. Consider the script of code

```
>> x  = 0 : .01 : 2*pi;
>> y1 = sin(x);
>> y2 = sin(2*x);
>> y3 = sin(4*x);
>> plot( x, y1 ,'--', x, y2, ':', x, y3,'+' )
>> grid
>> title ('Dashed, line, and dotted line graph')
```

Three plots are drawn with one call to `plot()`. Here we use a dashed line and dotted line for the first two graphs, while a `+` symbol is placed at each node for the third.

Example. When one of the arguments to `plot` is a vector and the other is a matrix, `plot` will graph each column of the matrix versus the vector. For example

```
>> x = 0 : .01 : 2*pi;
>> y = [ sin(x)', sin(2*x)', sin(4*x)' ];
>> plot(x,y)
```

`x` is an array having one row and 629 columns. `y` is a matrix having 629 rows and 3 columns.

Holding Figures

So far in this tutorial we have generated all our figures, including those with multiple plots, with one call to `plot`. Another way of generating figures with multiple plots is with the command `hold`, which freezes the current graphics screen so that subsequent plots can be superimposed on it.

Example. Figure 3.6 can also be generated with the sequence of commands

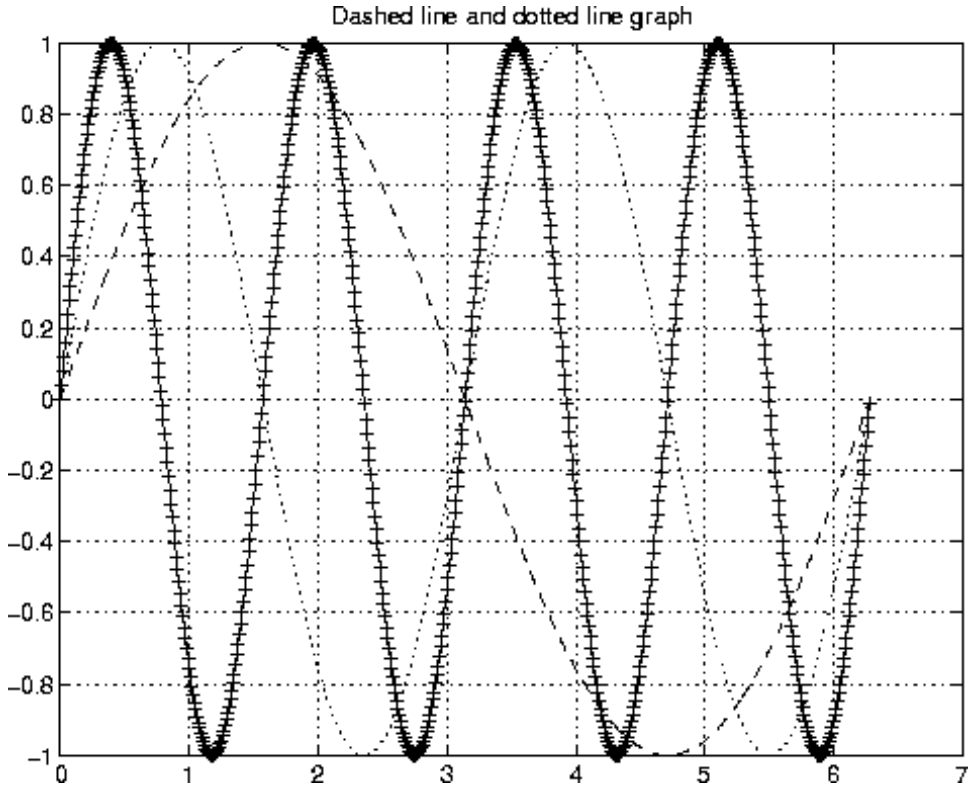


Figure 3.6. Overriding default line- and point- types

```
>> x  = 0 : .01 : 2*pi;
>> plot( x,   sin(x), '--' )
>> hold
>> plot( x, sin(2*x), ':' )
>> plot( x, sin(4*x), '+' )
>> grid
>> title ('Dashed, line, and dotted line graph')
```

The "hold" can be released by entering `hold` again. The command `hold on` holds the current plot and all axis properties so that subsequent graphing commands add to the existing graph. Conversely, the command `hold off` returns to the default mode whereby `plot` commands erase the previous plots and reset all axis properties before drawing new plots.

3.2 Three-Dimensional Plots

MATLAB uses the function `plot3`, the three-dimensional counterpart of `plot`, to display arrays of data points $(x_i, y_i, z_i),\ i = 1, 2, 3, \cdots, n$ in three-dimensional space. The syntax is

- **plot3(x,y,z)** When `x`, `y`, and `z` are vectors of the same length, this function plots a line in three-dimensional space through the points whose coordinates are the elements of `x`, `y`, and `z`.

There are many variations on this function's use, and we simply recommend you type `help plot3` for a list of options.

Example. Suppose that 8 experimental measurements (x,y,z) are stored in the rows of a matrix called `data`:

```
>> data = [ 2.5  1.3  0.0;
            0.0  2.0  0.0;
            1.0  3.0  0.0;
            2.5  3.5  4.0;
            3.0  1.0 -2.0;
            2.0 -1.0 -2.0;
            3.5  4.0 -2.5;
            0.0  1.0  0.0 ];
>>
```

The block of MATLAB commands

```
>> plot3 ( data(:,1), data(:,2), data(:,3), 'o' )
>> axis([ -1 4 0 5 -5 5 ])
>> grid
>> xlabel('x'), ylabel('y'), zlabel('z')
```

generates the data points, grid, and axes in Figure 3.7. The first thing that you should notice about this script is how we have used colon notation (e.g., `data(:,1)`) to extract the vectors `x`, `y`, and `z` from `data`. This script also demonstrates that many of the optional features we have used with `plot`, such as line and mark types, grid, and axis settings, can also be used with `plot3`.

Now we add the labels `Start` and `End` to the first and last points in array `data`. The block of MATLAB code

```
>> a = size ( data )
>> text( data(      1,1) + 0.2 , data(      1, 2), 'Start' );
>> text( data( a(1,1),1) + 0.2 , data( a(1,1), 2), 'End' );
```

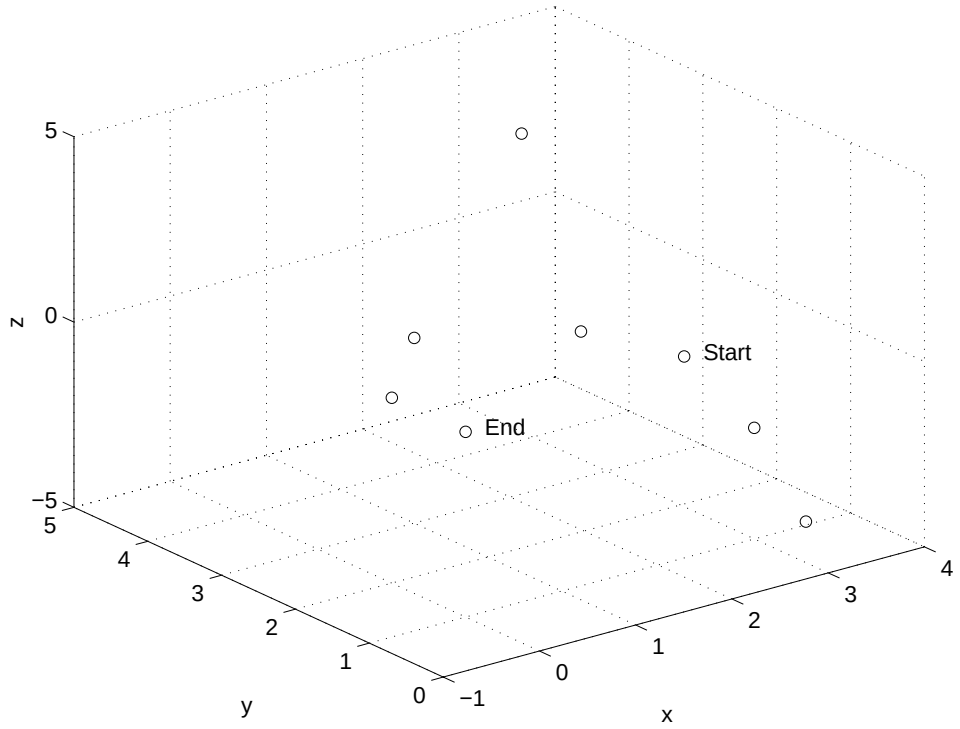


Figure 3.7. Three-dimensional plot of experimental data points.

uses the function `size` to obtain the number of rows and columns in array data – `a(1,1)` contains the number of rows in `data`, and `a(1,2)` the number of columns. The `text` function then adds the desired labels at the coordinates of the first and last data points.

3.3 Mesh and Surface Plotting

MATLAB provides a suite of functions for creating three-dimensional mesh and surface plots of functional relationships

$$z = f(x, y) \tag{3.3}$$

above a rectangular region defined in the x-y plane. MATLAB represents $z = f(x, y)$ as an array of data points above a regular grid of points lying in the (x-y) plane (see Figure 3.8). The surface shape

(or mesh shape) is highlighted by connecting the neighboring data points by straight-line segments, with the result in many cases looking like a fishing net.

Three steps are needed to create a mesh or surface plot:

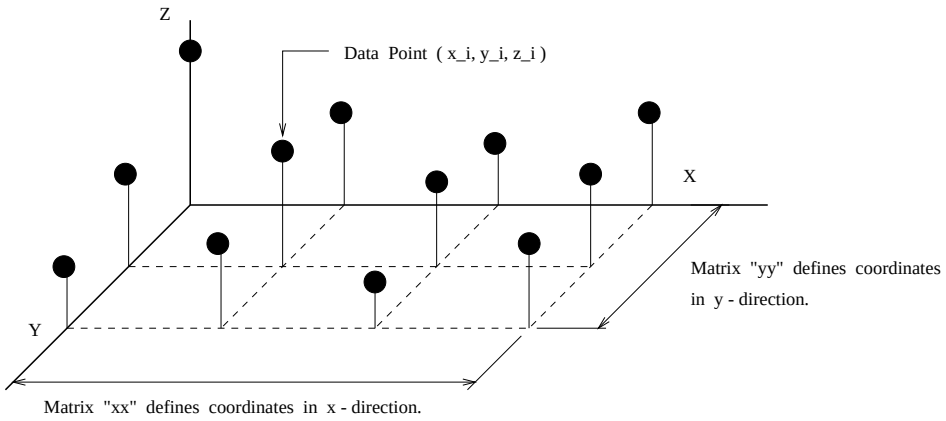


Figure 3.8. Three-dimensional mesh above regular grid of points in (x-y) plane

1. **Generate Two-Dimensional Grid in (x-y) Plane.** A rectangular grid of points in the (x-y) plane is generated by defining vectors `xx` and `yy` for the coordinate positions of the nodes along the x- and y-axes. Then, the function call

   ```
   [ x, y ] = meshgrid ( xx, yy )
   ```

   transforms the domain into rectangular arrays `x` and `y` for the efficient evaluation of $z = f(x, y)$. The rows of the output array `x` are copies of the vector `xx`, and the columns of the output array `y` are copies of the vector `yy`.

2. **Compute z Values at (x,y) Coordinates of Grid.** Now that matrices `x` and `y` of coordinates in the (x-y) domain are place, we can systematically evaluate Equation (3.3) for each (x,y) coordinate pair. The results are stored in a rectangular array z.

   We soon see that by writing $f(x, y)$ in terms of matrix element-level operations, in many cases, the entire evaluation of (x, y) coordinate pairs can be achieved with only one MATLAB statement.

3. **Draw Mesh and Surface Plots.** Three-dimensional mesh and surface plots may be produced with the functions:

- **mesh(z)** Creates a three-dimensional mesh plot of the elements of matrix `z`. If `z` is an m-by-n matrix, then the x- and y-axes will cover the ranges 1 through m and 1 through n, respectively.
- **surf(z)** Draws a three-dimensional surface plot of the elements of matrix `z`. Otherwise, it is the same as `mesh(z)`.
- **mesh(x,y,z)** Creates a three-dimensional mesh plot of the elements of matrix `z`, the x- and y-axes labeled to cover the range of values in matrices `xx` and `yy`.
- **surf(x,y,z)** Creates a three-dimensional surface plot of the elements of matrix `z`, the the x- and y-axes labeled to cover the range of values in matrices `xx` and `yy`.

The `mesh` and `surf` functions can also be used with various color options, and we refer you to the online MATLAB documentation for the relevant details.

Example. Suppose that we want to create a three-dimensional plot of the function

$$z = f(x, y) = \left[x^2 + y^2\right] \cdot \frac{\sin(y)}{y} \tag{3.4}$$

over the domain $-10 \leq x \leq 10$ and $-10 \leq y \leq 10$.

The block of MATLAB code

```
>> xx = -10: 0.4: 10;
>> yy = xx;
>> yy = yy + (yy==0)*eps
>> [x,y] = meshgrid(xx,yy);
```

generates matrices `xx` and `yy` containing coordinates of $-10 \leq x \leq 10$ and $-10 \leq y \leq 10$ divided into intervals of 0.4. The third statement moves points along the line y = 0 to y = eps so that MATLAB will not generate a "divide by zero" in its evaluation of Equation (3.4). Finally, the `meshgrid` function returns (51×51) matrices `x` and `y` containing the x- and y-coordinates in the rectangular domain. The command

```
>> z = (x.^2 +  y.^2).*sin(y)./y
```

systematically evaluates z, a (51×51) matrix, for each of the (x-y) coordinate pairs in the rectangular domain. Finally, the commands

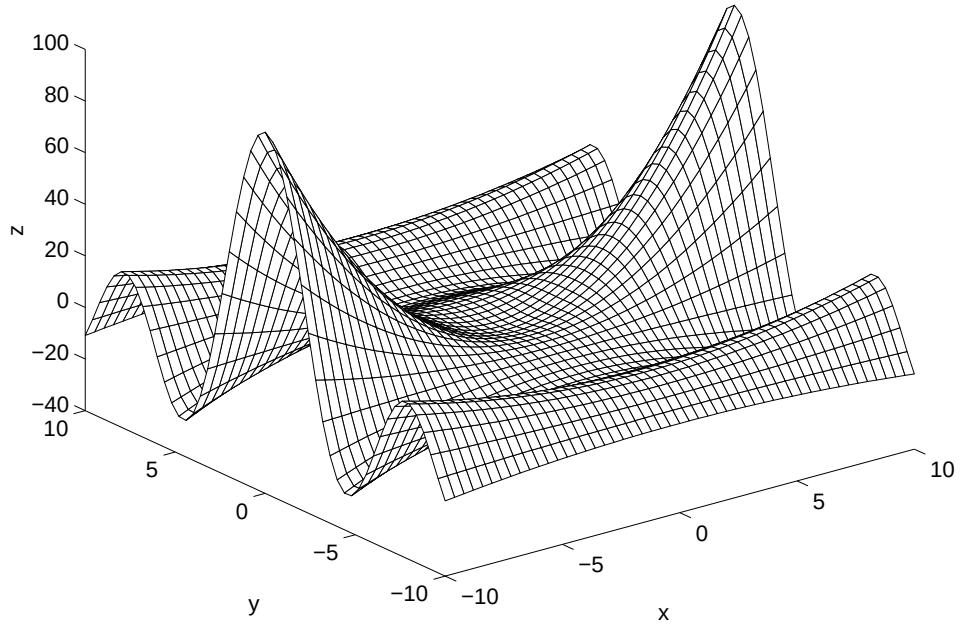


Figure 3.9. Three-dimensional mesh.

```
>> mesh(xx,yy,z)
>> xlabel('x'), ylabel('y'), zlabel('z')
```

generate a three-dimensional mesh plot, with the x- and y-axes scaled to cover the range of points contained in matrices `xx` and `yy`. The result is shown in Figure 3.3.

3.4 Contour Plots

A contour plot is an elevation map containing families of lines connecting regions of equal elevation. Sometimes it is convenient to think of a contour as a slice of a region at a particular elevation.

In MATLAB, contour plots are generated from three-dimensional elevation data:

- **contour(z)** Generates a contour plot of matrix z where the matrix elements are treated as heights above the (x-y) plane.

- **contour(x,y,z)** Generates a contour plot where `x` and `y` are vectors specifying coordinates on the x- and y- axes. Again, `z` is a matrix whose elements are treated as heights above the (x-y) plane.

- **contour(x,y,z,v)** Matrices `x`, `y` and `z` are as previously defined. Vector `v` tells contour to draw `length (v)` contour lines at the elevations specified in the elements of `v`.

Three-dimensional mesh and surface plots may be drawn with a contour diagram lying in the x-y plane. The relevant function specifications are

- **meshc(x,y,z)** This function is the same as `mesh`, except that a contour plot is drawn beneath the mesh.

- **surfc(x,y,z)** This function is the same as `surf`, except that a contour plot is drawn beneath the surface.

Example. With matrices `xx`, `yy`, `x`, `y`, and `z` in place for Figure 3.3, a two-dimensional contour map can be drawn and labeled with

```
>> contour(xx,yy,z)
>> xlabel('x'), ylabel('x')
```

Similarly, a three-dimensional surface plot with a contour map drawn beneath can be generated with the command

```
>> surfc(xx,yy,z)
```

The results are shown in Figures 3.10 and 3.11.

3.5 Subplots

MATLAB graphics windows will contain one plot by default. The `subplot` command can be used to partition the graphics screen so that either two or four plots are displayed simultaneously. Two subwindows can be displayed either "side by side" or "top and bottom." When the graphics window is partitioned into four subwindows, two are on the top and two are on the bottom.

The syntax for setting up a `subplot` is

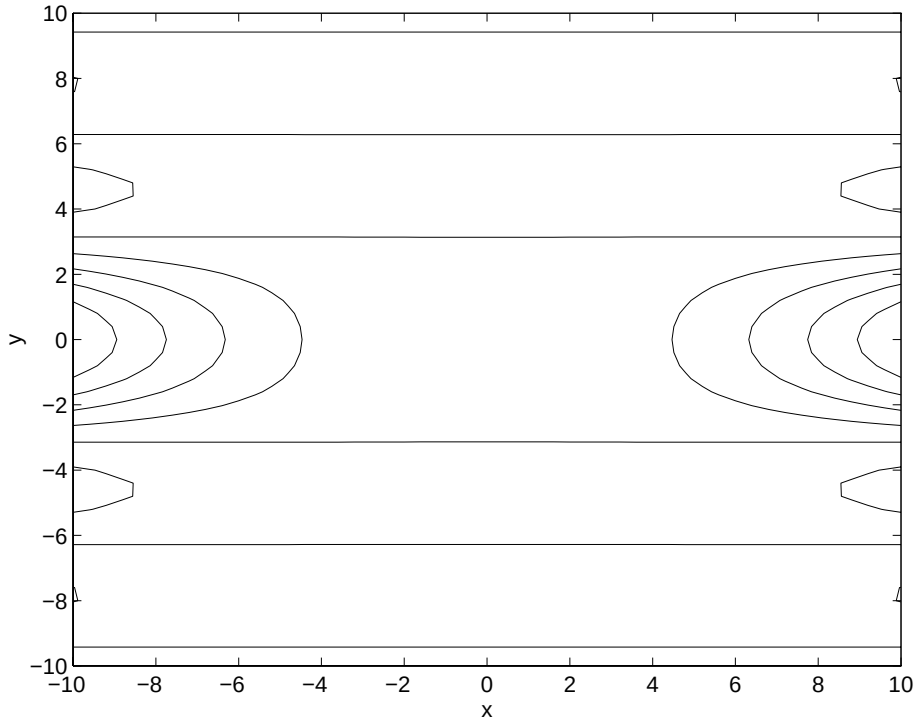


Figure 3.10. Contour plot.

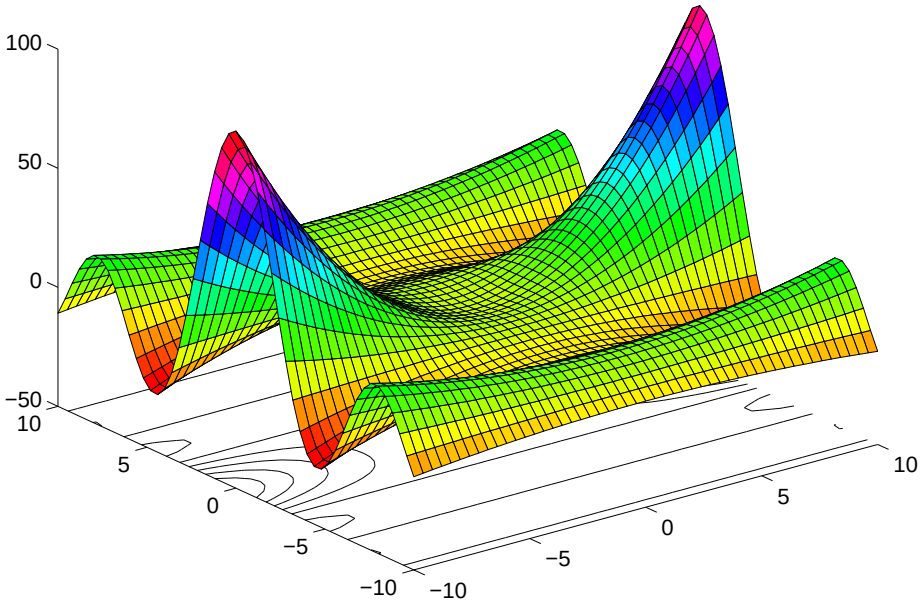


Figure 3.11. Combined surface and contour plot.

- **subplot(i,j,k)** The subplot function takes three integer arguments `i`, `j`, and `k`. Integers `i` and `j` specify that the graphics window should be partitioned into an i-by-j grid of smaller windows. The subwindows are numbered from left to right, top to bottom. Integer `k` specifies the kth subplot for the current graphics window.

Example. The sequence of commands that follow generates the array of subplots shown in Figure 3.12.

```
% Define array of temperatures in chimney cross section

  temp = [ NaN    NaN    NaN    NaN  200.0  147.3   96.5   47.7    0.0 ;
           NaN    NaN    NaN    NaN  200.0  146.9   96.0   47.4    0.0 ;
           NaN    NaN    NaN    NaN  200.0  145.2   93.9   46.1    0.0 ;
           NaN    NaN    NaN    NaN  200.0  140.4   88.9   43.2    0.0 ;
         200.0  200.0  200.0  200.0  200.0  128.2   78.8   37.8    0.0 ;
         147.3  146.9  145.2  140.4  128.2   94.3   60.6   29.6    0.0 ;
          96.5   96.0   93.9   88.9   78.8   60.6   40.2   19.9    0.0 ;
          47.7   47.4   46.1   43.2   37.8   29.6   19.9   10.0    0.0 ;
           0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0 ];

% Generate contour and surface subplots.

  subplot(2,2,1);
  contour( temp )
  title('Contour Plot');
  subplot(2,2,2);
  mesh( temp )
  title('Mesh Plot');
  subplot(2,2,3);
  surf( temp )
  title('Surface Plot');
  subplot(2,2,4);
  surfc( temp )
  title('Surface Plot with Contours');

% ================================================
% The End !
```

In the function calls `contour( temp )`, `mesh( temp )`, `surf( temp )`, and `surfc( temp )`, the elements of matrix `temp` are treated as elevations above the (x-y) plane.

You may recognize array `temp` from the C tutorial, where we computed the distribution of temperatures in one fourth of a chimney cross-section. The chimney interior is represented by the block of `NaN`s. The zero elements along the right-hand side and bottom of `temp` represent the

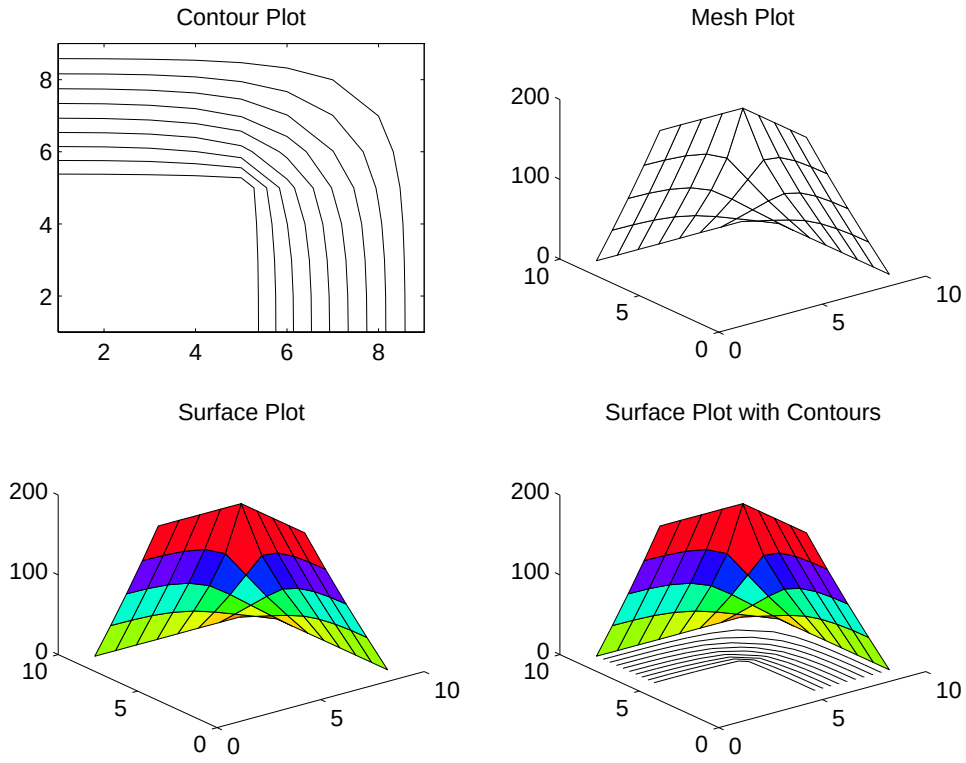


Figure 3.12. Temperature profiles in chimney cross-section

outside temperature. The row and column of 200 elements are the temperature along the inside wall of the chimney.

3.6 Hard Copies of MATLAB Graphics

To create a black-and-white postscript file of a MATLAB figure, just type

```
>> print name-of-figure.ps
```

Color postscript files can be generated with

```
>> print -dpsc name-of-figure.ps
```

Similarly, to create a color `jpg` file (i.e., Joint Photographic Experts Group file), try the following command

```
>> print -djpeg name-of-figure.jpg
```

Online information can be obtained by typing `help print`.

3.7 Preparing MATLAB Graphics for the World Wide Web

The UNIX tool `xv` can then be used to convert the postscript file into a `gif` file format, suitable for reading by WWW browsers.

3.8 Review Questions

1. How can the axes in a two-dimensional plot be constrained to retain the true shape of a figure?
2. How does MATLAB handle the plotting of

$$f(x) = \frac{(x-2)}{(x-2)} \tag{3.5}$$

   at x = 2? What is a good way of avoiding the "divide-by-zero" scenario?

3. What does the `hold` command do?
4. How would you draw a three-dimensional surface together with a contour plot of elevation levels?

3.9 Programming Exercises

3.1 **(Beginner).** Suppose that column one in the following file, `RAIN.TXT`, contains the daily rainfall for one week, and the second column contains the average rainfall for the week.

```
0.100000 0.694286
1.000000 0.694286
0.000000 0.694286
0.200000 0.694286
3.560000 0.694286
0.000000 0.694286
0.000000 0.694286
```

Write a MATLAB prorgram that will

1. Read the contents of `RAIN.TXT` into MATLAB
2. Plot a histogram of the daily rainfall measurements and a horizontal line for the average weekly rainfall.

3.2 **Beginner.** The fragment of MATLAB code

```
>> t = 0:0.01:2*pi;
>> plot( sin(t), cos(t) )
```

plots a circle of radius 1 centered at the origin.

1. Write a function file, `circle.m`, that will plot a circle of radius r centered at coordinate (x.y). A suitable function declaration is

   ```
   function circle ( x, y, r )
   ```

2. With your `circle` function in hand, write a short M-file that will generate the rectangular layout of circles shown in Figure 3.13. The circles are positioned along the x coordinates x = 5 and x = 50 and the y coordinates y = 5 and y = 50. Each circle should have a radius 2.5.

3.3 **Beginner.** Write a MATLAB script file that will plot and label the function

$$z = f(x, y) = \left[e^{-x^2 - y^2} \cdot \sin(y) \cdot \sin(y) \right] \tag{3.6}$$

over the domain $-2 \leq x \leq 2$ and $-2 \leq y \leq 2$.

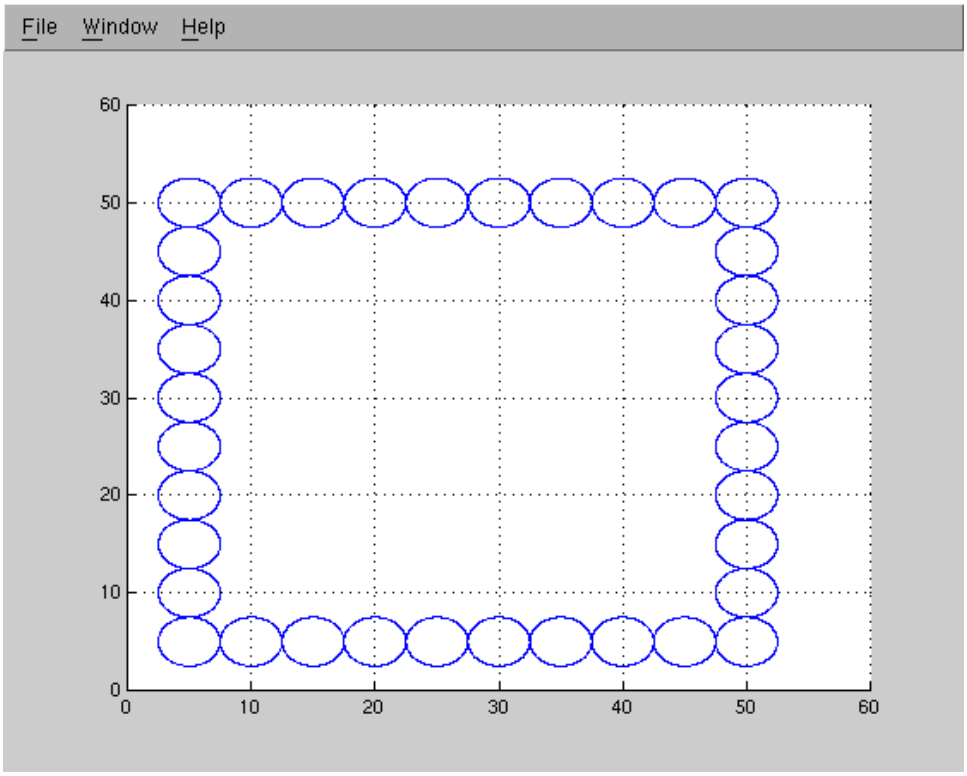


Figure 3.13. Rectangular layout of circles

3.4 **Beginner.** Write a MATLAB script file that will plot and label the function

$$z = f(x, y) = \text{int}\left[4.0 \cdot \sin^3(x) \cdot \sin^3(y) + 0.5\right] \tag{3.7}$$

for the domain $-10 \leq x \leq 10$ and $-10 \leq y \leq 10$. In Equation (3.7), `int` is a function that truncates the fractional part of a floating point number.

3.5 **Intermediate.** Figure 3.14 shows a sphere of radius r and density ρ floating in water.

The weight of the sphere will be $4/3\rho\pi r^3$. The volume of the spherical segment displacing water is $1/3\pi(3rd^2 - d^3)$.

1. Show that the depth of the sphere floating in water is given by solutions to

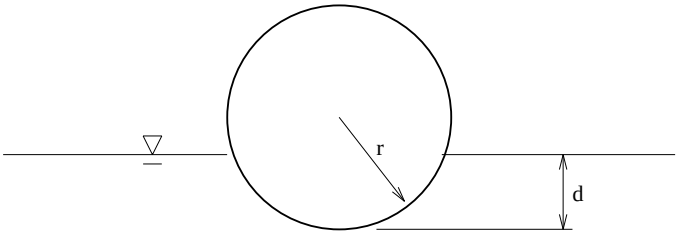


Figure 3.14. Sphere floating in water.

$$f(x, \rho) = x^3 - 3.x^2 + 4.\rho = 0 \tag{3.8}$$

where x = d/r is a dimensionless quantity.

2. Write a MATLAB program that will compute the depth to which a sphere will sink as a function of its radius for $\rho = 0$ to $\rho = 1$ in increments of 0.1.
3. Use MATLAB to plot and label the results.

Note. Perhaps the most straightforward way of solving this problem is to write a numerical procedure that computes the root of the cubic equation. This is not as hard as it might seem since $f(0, \rho)$ is always greater than zero and $f(2, \rho)$ is always less than zero. Only one solution to Equation (3.8) lies within the interval $f([0, 2], \rho)$ and so standard root finding techniques such as bisection and Newton Raphson will work. A more elegant way of solving this problem is with the `contour` function.

3.6 **Intermediate.** Write a block of MATLAB code that will evaluate and plot Equation (3.1) without error and without moving the `x = 0` coordinate to `x = eps`.

3.7 **Intermediate.** Write a MATLAB script file that will plot and label the family of ellipses defined by

$$[x(t), y(t)] = [A\sin(t), B\cos(t)] \tag{3.9}$$

for $0 \leq t \leq 2\pi$, A = 1, and B = 1, 2, $\cdots$ 5.

3.8 **Intermediate.** Write a MATLAB program that will draw a circle, and partition and label its perimeter into n equal segments. Line segments should then be drawn to connect all the

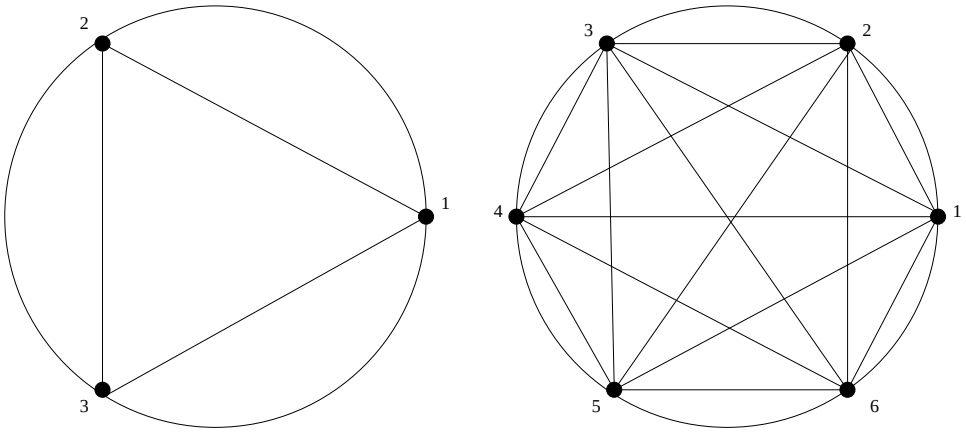


Figure 3.15. Partitioned circles.

points lying on the circle perimeter. The left- and right-hand sides of Figure 3.15 demonstrate the required partitioning and labeling for n = 3 and n = 6, respectively.

Devise an algorithm that requires no more than $n(n-1)/2$ function calls to draw all the line segments. This is not as hard as it seems – simply observe, for example, that once a line segment has connected points 1 and 6, there is no need to draw a second line connecting points 6 and 1.

3.9 **Moderate.** An experiment is conducted on 36 specimens to determine the tensile yield strength of A36 steel. The experimental results are as follows:

```
42.3    42.1    41.8    42.4    47.7    41.4
40.5    38.7    40.8    39.6    42.4    37.5
39.9    45.3    41.6    36.8    45.4    44.8
39.2    40.7    38.5    40.1    42.8    42.5
43.1    36.2    46.2    41.5    38.3    40.2
41.9    40.4    39.1    38.6    46.3    39.5
```

Write a MATLAB program and appropriate M-files that will:

1. Read the experimental test results from a file `experimental.data`.
2. Compute and print the maximum, minimum, average, and median tensile strengths.
3. Generate a histogram of “observations” versus “yield stress”
4. Construct a bar graph or stair-step graph of “cumulative frequency” versus “yield stress.”

Note. The average value of the experimental results can be computed using formulae presented in Section 8.5. After the experimental results have been numerically sorted, the median value is the midvalue (if the number of data points is odd) or the average of the two middle values (if the number of data points is even). The "cumulative frequency" versus "yield stress" is given by

$$\text{Cumulative frequency(y)} = \int_0^y p(x)dx \tag{3.10}$$

where $p(x)$ is the probability distribution of tensile yield strengths.

Chapter 4

Solution of Linear Matrix Equations

In this chapter, we learn how MATLAB can be used to solve engineering problems involving linear systems of matrix equations. Linear systems of equations arise in many areas of engineering including structural mechanics, circuit simulation, control of electrical and mechanical devices, and finite element analysis. Moderate to large families of equations commonly represent the state of a system (e.g., equations of equilibrium, energy and momentum conservation).

4.1 Definition of Linear Matrix Equations

We begin with the basics. In expanded form, a system of m linear equations with n unknowns may be written

$$\begin{array}{rrrrr} a_{11}x_1 & + a_{12}x_2 & + a_{13}x_3 + & \cdots + a_{1n}x_n & = b_1 \\ a_{21}x_1 & + a_{22}x_2 & + a_{23}x_3 + & \cdots + a_{2n}x_n & = b_2 \\ a_{31}x_1 & + a_{32}x_2 & + a_{33}x_3 + & \cdots + a_{3n}x_n & = b_3 \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ a_{m1}x_1 & + a_{m2}x_2 & + a_{m3}x_3 + & \cdots + a_{mn}x_n & = b_m \end{array} \tag{4.1}$$

The constants a_{11}, a_{21}, a_{31}, $\cdots$ a_{mn} and b_1, b_2, $\cdots$ b_m are called the equation coefficients. Most often the coefficients will be real numbers, but they could also be complex numbers. The variables x_1, x_2 $\cdots$ x_n are the unknowns in the system of equations. The matrix equivalent of Equation 4.1 is $[A] \cdot \{X\} = \{B\}$, where

$$[A] = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & & \vdots \\ \vdots & & & \vdots \\ a_{m1} & \cdots & \cdots & a_{mn} \end{bmatrix}, \{X\} = \begin{Bmatrix} x_1 \\ x_2 \\ \\ \vdots \\ x_n \end{Bmatrix}, \text{ and } \{B\} = \begin{Bmatrix} b_1 \\ b_2 \\ \\ \vdots \\ b_m \end{Bmatrix} \tag{4.2}$$

Equations 4.2 are said to be homogeneous when the right-hand side vector {B} = 0. A system of equations is said to be **under-determined** when there are more unknowns than equations (i.e., m < n). Conversely, a system of equations is termed **over-determined** when there are more equations than unknowns (i.e., m > n). Over-determined systems of equations arise in linear optimization methods, and in the problem of finding the best fit of a low-order equation to experimental data. A well-known name for the latter application is least squares analysis.

Geometry of Two- and Three-dimensional Systems

Linear algebra plays a central role in the development of numerical equation solvers because it allows for the classification solutions to Equation 4.2. Indeed, before the development of numerical equation solvers can proceed, we need to understand under what circumstances a system of equations will have a unique solution. For those cases where a system of equations has more than one solution, we also need to know how many solutions there will be and how they can be characterized.

A good way of gaining insight into these issues is to study systems of equations whose solutions are simple enough to be graphically displayed. This will occur for two- and three- dimensional problems (i.e., when $m = n = 2$ and $m = n = 3$).

Two-Dimensional Example. When $m = n = 2$, the matrix equations $[A] \cdot \{X\} = \{B\}$ can be interpreted as a pair of straight lines in the (x_1, x_2) plane. That is,

$$a_{11}\, x_1 \quad + a_{12}\, x_2 \;=\; b_1 \tag{4.3}$$

$$a_{21}\, x_1 \quad + a_{22}\, x_2 \;=\; b_2 \tag{4.4}$$

The pair of Equations 4.3 and 4.4 may be interpreted as a linear transformation from two-dimensional coordinate space (x_1, x_2) into a two-dimensional right-hand side vector space (b_1, b_2) (see Figure 4.1).

The problem of finding solutions to Equations 4.3 and 4.4 is equivalent to finding points $X = (x_1, x_2)$ in the (X_1, X_2) plane that will be mapped via the transformation $A \cdot X$ into the (B_1, B_2) plane.

As we soon see, this problem is complicated by three types of solutions, namely, (1) no solutions to A.X = B, (2) a unique solution to A.X = B, or (3) an infinite number of solutions to A.X = B.

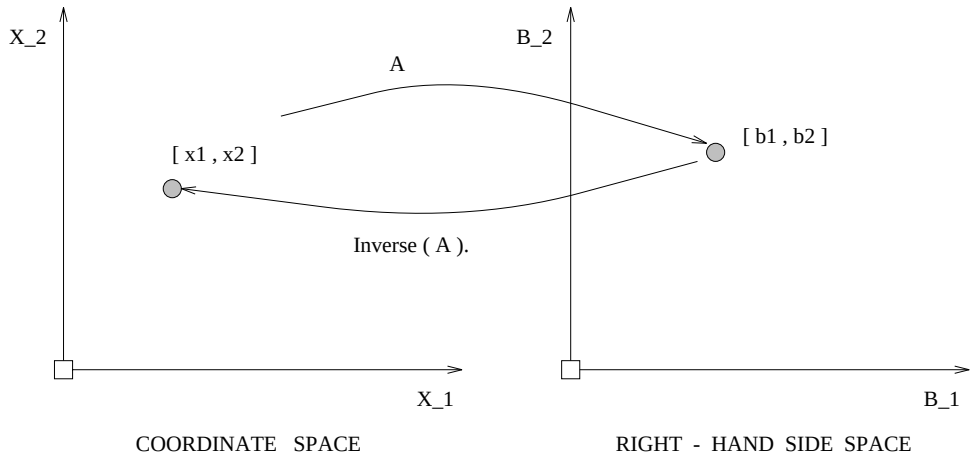


Figure 4.1. Matrix transformations.

4.2 Hand Calculation Procedures

There are two hand calculation procedures for computing a solution(s) to the system of equations. In the **graphical method**, plots of lines in two dimensions are constructed. Solutions to the system of equations correspond to points where the lines intersect. An example of such a solution is shown in the left-hand schematic of Figure 4.2, labeled "Unique Solution."

In the second method, the equations are premultiplied by constants in such a way that when they are combined variables will be eliminated. For example, if Equation 4.3 is multiplied by a_{21} and Equation 4.4 is multiplied by a_{11}, then we have the system of equations

$$a_{21} \cdot a_{11} \cdot x_1 \quad + a_{21} \cdot a_{12} \cdot x_2 \; = \; a_{21} \cdot b_1 \tag{4.5}$$

$$a_{11} \cdot a_{21} \cdot x_1 \quad + a_{11} \cdot a_{22} \cdot x_2 \; = \; a_{11} \cdot b_2 \tag{4.6}$$

Subtracting Equation 4.5 from Equation 4.6 gives

$$x_2 = \left[\frac{a_{11} \cdot b_2 - a_{21} \cdot b_1}{a_{11} \cdot a_{22} - a_{12} \cdot a_{21}} \right]. \tag{4.7}$$

Now x_2 can be back-substituted into either Equation 4.3 or Equation 4.4 for the corresponding value of x_1. You may may have noticed that this procedure is not unique. If Equation 4.3 is multiplied by a_{22} and Equation 4.4 is multiplied by a_{12}, subtraction of equations gives

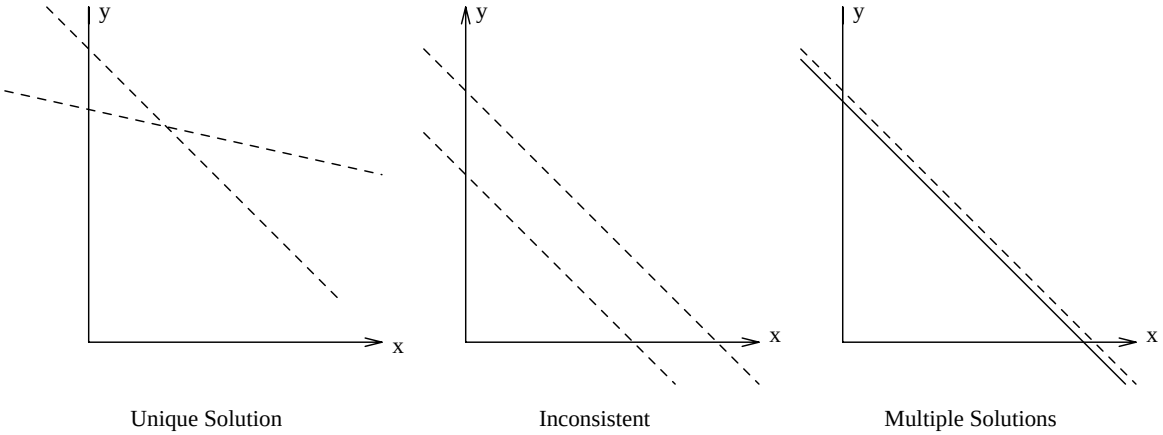


Figure 4.2. Types of solutions to matrix equations.

$$x_1 = \left[\frac{a_{22} \cdot b_1 - a_{12} \cdot b_2}{a_{11} \cdot a_{22} - a_{12} \cdot a_{21}} \right] \tag{4.8}$$

Now x_2 can be back-substituted into either Equation 4.3 or and x_1 can be back-substituted into either Equation 4.3 or 4.4 for the corresponding value of x_2.

4.3 Types of Solutions for Systems of Linear Matrix Equations

We observe that the denominators of Equations 4.7 and 4.8 are the same, and correspond to the **determinant** of a (2×2) matrix, namely:

$$det(A) \; = \; det \left[\begin{array}{cc} a_{11} & a_{12} \\ a_{21} & a_{22} \end{array} \right] \; = \; a_{11} \cdot a_{22} \; - \; a_{12} \cdot a_{21}. \tag{4.9}$$

The same principles apply to families of equations containing three unknowns (i.e., $m = n = 3$). When the equations are written in the form $[A] \cdot \{X\} = \{B\}$, A is a (3×3) matrix. From a geometric point of view, each equation describes a plane in three-dimensional space, as shown in Figure 4.3. The solution to the system of equations corresponds to those points in three-dimensional space that lie in all three planes. There will only be one such point when `det(A)` $\neq 0$. The mathematical condition `det(A)` $= 0$ occurs when two or more planes are parallel. As with the two-dimensional case, there will be either an infinite number of solutions or no solutions. An infinite number of

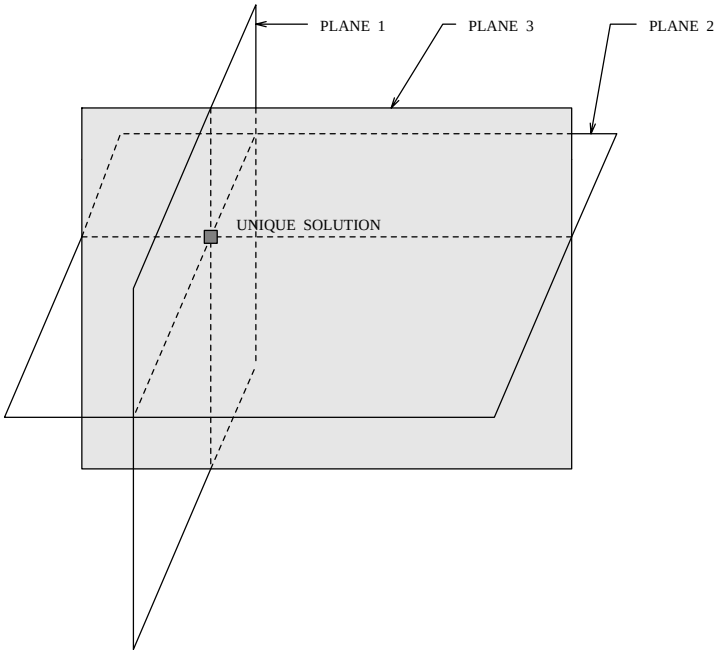


Figure 4.3. Intersection of planes in three-dimensional space.

solutions occurs when all three planes have the same equation (or a nonzero constant multiplied by the same equations). No solutions occur when at least two of the planes are parallel but do not have the same equation.

When problem sizes increase above three unknowns, we can no longer rely on graphical methods to find a solution(s). The nature of solutions to linear equations is given instead by theoretical results from linear algebra. The results are

1. A unique solution $\{X\} = [A^{-1}] \cdot \{B\}$ exists when $[A^{-1}]$ exists (i.e., $\det [A] \neq 0$).
2. The equations are inconsistent when $[A]$ is singular and rank $[A|B] \neq$ rank $[A]$.
3. If rank $[A|B]$ equals rank $[A]$, then there are an infinite number of solutions.

For the case of a unique solution, $[A^{-1}]$ is the **inverse** of matrix `A`, and such a matrix will exist when the determinant of `A` is nonzero. An equivalent way of detecting a unique solution is with the rank of matrix `A`, that is, the number of linearly independent rows in matrix `A`. In a nutshell, a unique solution to the matrix equations will exist when the `rank (A)` equals the number of rows

or columns in A. In cases 2 and 3 above, the notation $[A|B]$ stands for the matrix A juxtaposed with matrix B.

The MATLAB function `inv (A)` will return the inverse $\left[A^{-1}\right]$ of matrix A when it exists. The MATLAB function `det (A)` will return the determinant of a square matrix A. The MATLAB function `rank (A)` will return the rank of A, otherwise known as the number of linearly independent rows (or columns) in A.

4.4 Case Study Problem : Three Linear Matrix Equations

Suppose that the following three equations describe the equilibrium of a simple structural system as a function of external loads and computed displacements.

$$\begin{array}{rrrl} 3x_1 & -1x_2 & +0x_3 & = 1 \\ -1x_1 & +6x_2 & -2x_3 & = 5 \\ 0x_1 & -2x_2 & +10x_3 & = 26 \end{array} \tag{4.10}$$

This family of equations can be written in the form `A.X = B` where

$$[A] = \begin{bmatrix} 3 & -1 & 0 \\ -1 & 6 & -2 \\ 0 & -2 & 10 \end{bmatrix}, \{X\} = \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix}, \text{ and } \{B\} = \begin{Bmatrix} 1 \\ 5 \\ 26 \end{Bmatrix} \tag{4.11}$$

In a typical application, matrices A and B will be defined by the parameters of the engineering problem, and the solution matrix X will need to be computed. In this particular case, the solution matrix

$$\{X\} = \begin{Bmatrix} 1 \\ 2 \\ 3 \end{Bmatrix}, \tag{4.12}$$

makes the left- and right-hand sides of matrix Equation 4.11 equal. The following script of MATLAB code

```
>> A = [ 3 -1 0; -1 6 -2; 0 -2 10 ];
>> B = [ 1; 5; 26 ];
>> X = A\B
```

defines matrices A and B, and computes and prints the solution to A.X = B, namely,

```
>> X = A\B
X =
    1.0000
    2.0000
    3.0000
>>
```

Of course, this solution can be verified by first computing the inverse of A, and then postmultiplying it by matrix B. The relevant details of MATLAB code are

```
>> Ainv = inv(A)
Ainv =
    0.3544    0.0633    0.0127
    0.0633    0.1899    0.0380
    0.0127    0.0380    0.1076
>> X = Ainv*B
X =
    1.0000
    2.0000
    3.0000
>>
```

We can use MATLAB to check that this result is consistent with the rank and determinant of matrix A. First, the script of code

```
>> rank (A)
ans =
     3
>>
```

shows that the matrix rank is equal to 3, the number of rows and columns in matrix A. We therefore expect that the matrix equations will have a unique solution. A second indicator of the solution type is the matrix determinant. The script of code

```
>> det (A)
ans =
   158
>>
```

shows that the matrix determinant is nonzero, again indicating the presence of a unique solution to the matrix equations.

4.5 Singular Systems of Matrix Equations

A family of matrix equations A is said to be **singular** when the individual equations are dependent. That is, one or more of the matrix equations can be written as a linear combination of the remaining equations. For example, matrix

```
>> A = [ 1 2 3;
         2 4 6;
         4 5 6 ];
```

is singular because the elements of the second row are simply two times those in the first row. For a general matrix A, such a relationship may be far from evident, and so we must rely on the rank and determinant functions to identify singular systems. In this case, the function call

```
>>  rank (A)
ans =
     2
>>
```

highlights the presence of a singular system because the `rank(A) = 2` is less than 3, the number of rows and columns in A. Consequently, the inverse of A will not exist. Should a situation of this type be accidentally overlooked in an engineering computation, MATLAB will display an error message and return a solution vector containing NaNs or $\pm\infty$, depending on the values elements in matrix A. Consider, for example

```
>> Ainv = inv(A)
Warning: Matrix is singular to working precision.

Ainv =
   Inf   Inf   Inf
   Inf   Inf   Inf
   Inf   Inf   Inf
>>
```

The error condition `Inf` will be propagated to all subsequent arithmetic computations involving `Ainv`.

Generally speaking, if a matrix determinant is computed to be exactly zero, then there is no difficulty in identifying the three types of matrix solutions mentioned in the previous section. But what about matrices that are nearly singular? Many practical implementations are complicated by factors such as round-off error, finite precision, and limited ranges of numbers that may be stored in

a computer (i.e., underflow and overflow of numbers). Perhaps matrix A is singular and a nonzero calculation is due to numerical problems, or perhaps it is not singular. Resolving these issues is far from a trivial matter.

4.6 Engineering Applications

Now that we are familiar with MATLAB's matrix and graphics capabilities, this section works step by step through the design and implementation of four engineering applications that require the solution of linear matrix equations. They are:

1. A MATLAB program for the structural analysis of a cantilever truss.
2. A MATLAB program for the electrical analysis of a circuit containing resistors and batteries.
3. A MATLAB program that computes a least squares analysis of experimental data.
4. A MATLAB program that computes and plots the distribution of temperature in a chimney cross-section.

Each problem description is accompanied by a brief discussion of the theory needed to set up the relevant matrix equations.

Structural Analysis of a Cantilever Truss

Problem Statement. In the design of highway bridge structures and crane structures, engineers are often required to compute the member forces and support reactions in planar truss structures. The analysis of cantilever truss structures is governed by the following principles

1. At each joint, the sum of internal and external forces in the horizontal and vertical directions must equal zero.
2. The sum of external forces and support reactions in the horizontal and vertical directions must equal zero.
3. For the entire structure and all possible substructures, the sum of moments must equal zero.

Figure 4.4 shows a six-bar cantilever truss carrying `10 kN` loads at joints `B` and `C`. The analysis begins with the arbitrary assignment of element numbers in the truss, numbers for the joints, and reaction components. We assume that

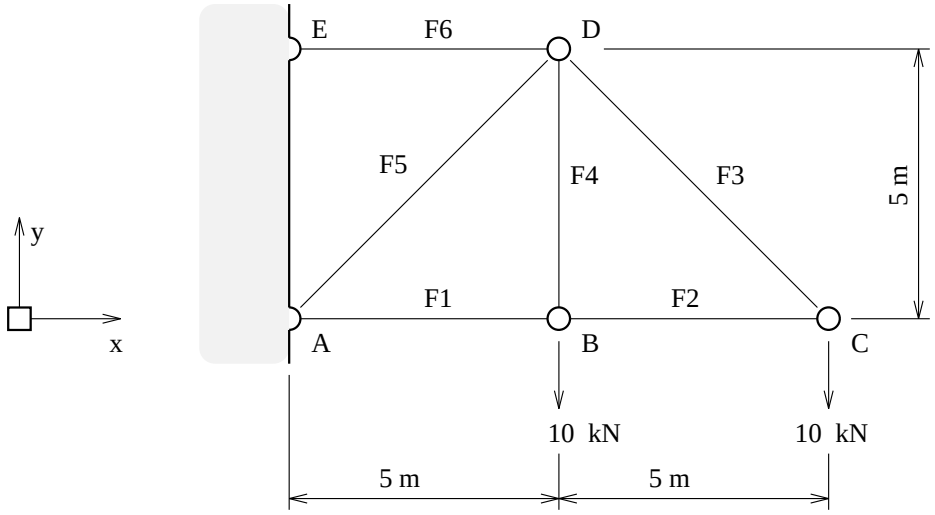


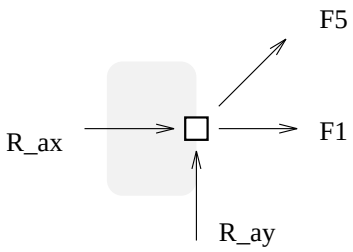
Figure 4.4. Six-bar cantilever truss.

1. The truss elements can only carry axial forces, with tensile axial forces being positive and compressive axial forces being negative.

2. All the joints are pinned (in other words, the joints cannot transfer moments; only axial forces from the truss elements).

The forces F_1, F_2, $\cdots$ F_6 represent the axial forces in truss members 1 through 6, and R_{ax}, R_{ay}, R_{ex}, and R_{ey} the support reactions at joints A and E.

The cantilever truss is "statically determinate" in the sense that axial forces in the elements and the support reactions can be computed without a knowledge of material properties. Equations of equilibrium in the horizontal and vertical directions are written for each of the five joints. The equations are then expressed in matrix form, and solved for the member axial forces and horizontal and vertical reaction components. For example, the forces acting on joint A are



The equations of equilibrium in the horizontal and vertical directions are

$$\sum F_x = F_1 + \frac{F_5}{\sqrt{2}} + R_{ax} = 0 \; \sum F_y = \frac{F_5}{\sqrt{2}} + R_{ay} = 0 \tag{4.13}$$

The equations of equilibrium for joint B are

$$\sum F_x = F_1 - F_2 = 0 \; \sum F_y = F_4 - 10 = 0 \tag{4.14}$$

The equations of equilibrium for joint C are

$$\sum F_x = F_2 + \frac{F_3}{\sqrt{2}} = 0 \; \sum F_y = \frac{F_3}{\sqrt{2}} - 10 = 0 \tag{4.15}$$

The equations of equilibrium for joint D are

$$\sum F_x = \frac{F_3}{\sqrt{2}} - \frac{F_5}{\sqrt{2}} - F_6 = 0 \; \sum F_y = F_4 + \frac{F_3}{\sqrt{2}} + \frac{F_5}{\sqrt{2}} = 0 \tag{4.16}$$

and for joint E, the equations of equilibrium are

$$\sum F_x = F_6 + R_{ex} = 0 \; \sum F_y = R_{ey} = 0 \tag{4.17}$$

Equations 4.13 to 4.17 are solved in two steps. First, we put Equations 4.14 to 4.16 in matrix form and solve for the member forces F_1 , $F_2 \cdots F_6$. Then we back-substitute the member forces into equations 4.13 and 4.17 to get the support reactions. In matrix form, Equations 4.14 to 4.16 are

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1/\sqrt{2} & 0 & 0 & 0 \\ 0 & 0 & 1/\sqrt{2} & 0 & 0 & 0 \\ 0 & 0 & 1/\sqrt{2} & 0 & -1/\sqrt{2} & -1 \\ 0 & 0 & 1/\sqrt{2} & 1 & 1/\sqrt{2} & 0 \end{bmatrix} \cdot \begin{Bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \\ F_5 \\ F_6 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 10 \\ 0 \\ 10 \\ 0 \\ 0 \end{Bmatrix} \tag{4.18}$$

In step two, the support reactions are given by

$$\begin{Bmatrix} R_{ax} \\ R_{ay} \\ R_{ex} \\ R_{ey} \end{Bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 0 & -1/\sqrt{2} & 0 \\ 0 & 0 & 0 & 0 & -1/\sqrt{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{Bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \\ F_5 \\ F_6 \end{Bmatrix} \tag{4.19}$$

Program Source Code. Program 4.1 allocates memory for matrix Equations 4.18 and 4.19, initializes the nonzero matrix element values, and computes a solution to Equations 4.18 and 4.19. It prints matrices of axial forces in the truss members and components of reaction at joints A and E.

Computer Program 4.1 : Structural Analysis of Cantilever Truss

```
% ============================================================================
% truss.m : Compute internal forces and reactions in cantilever truss structure.
%
% Matrices : Truss     = Represents truss geometry and element connectivity.
%          : Load      = External forces acting on the truss nodes.
%          : Force     = Internal member forces acting in truss elements.
%          : Support   = Relationship between truss and support reactions
%          : Reactions = Matrix of support reaction forces.
% ============================================================================

% Define problem parameters

  NoElmts     = 6;
  NoReactions = 4;

% Setup matrix for "truss connectivity"

  Truss = zeros( NoElmts, NoElmts );
  Truss( 1 , 1 ) =  1;
  Truss( 1 , 2 ) = -1;

  Truss( 2 , 4 ) =  1;

  Truss( 3 , 2 ) =  1;
  Truss( 3 , 3 ) =  1/sqrt(2);

  Truss( 4 , 3 ) =   1/sqrt(2);
```

```
  Truss( 5 , 3 ) =   1/sqrt(2);
  Truss( 5 , 5 ) =  -1/sqrt(2);
  Truss( 5 , 6 ) =  -1;

  Truss( 6 , 3 ) =   1/sqrt(2);
  Truss( 6 , 4 ) =   1;
  Truss( 6 , 5 ) =   1/sqrt(2);

% Setup matrix for "external loads" on truss nodes

  Load = zeros( NoElmts, 1 );
  Load( 2 , 1 ) =  10;
  Load( 4 , 1 ) =  10;

% Print matrices for "truss connectivity" and "external loads"

  Truss
  Load

% Solve equations and print "internal member" forces

  Force = Truss\Load

% Setup matrix for "support reactions"

  Support = zeros( NoReactions, NoElmts );

  Support( 1 , 1 ) = -1;
  Support( 1 , 5 ) = -1/sqrt(2);
  Support( 2 , 5 ) = -1/sqrt(2);
  Support( 3 , 6 ) = -1;

% Compute and print "support reactions"

  Reactions = Support*Force

% ============================================================
% the end!
```

Running the Program. Assume that Program 4.1 is contained the M-file `truss.m`. The script of input/output (I/O):

```
>> format compact
>> truss
Truss =
    1.0000   -1.0000         0         0         0         0
```

```
         0         0         0    1.0000         0         0
         0    1.0000    0.7071         0         0         0
         0         0    0.7071         0         0         0
         0         0    0.7071         0   -0.7071   -1.0000
         0         0    0.7071    1.0000    0.7071         0
Load =
     0
    10
     0
    10
     0
     0
Force =
  -10.0000
  -10.0000
   14.1421
   10.0000
  -28.2843
   30.0000
Reactions =
    30
    20
   -30
     0
>>
```

shows the command needed to run the program (i.e., `truss`) and the output that is generated by the analysis.

For programming convenience, we define the variables `NoElmts` and `NoReactions` for the number of frame elements and reactions, respectively. The matrices in Equations 4.18 and 4.19 are defined by first allocating memory for zero matrices of the appropriate size and then filling in the nonzero matrix elements.

Validating the Results. The member `"Forces"` matrix contains the axial forces in elements `1` through `6`. A quick examination of the matrix reveals that $F_1 = F_6 = -10$ kN (i.e., compression) and that $F_4 = 10$ kN (i.e., tension). Element `6` carries a tensile force of 30 kN. If you take moments about joint `A`, then you will see that the axial force in element 6 times a lever arm of `5 m` is balanced by the 10 kN loads at lever arms `5 m` and `10 m`.

The `"Support"` reactions matrix contains the horizontal and vertical support reactions at joints `A` and `E`. Two points should be noted. First, because the horizontal component of externally applied loads is zero, we expect that the sum of the horizontal reactions at joints `A` and `E` will be zero. They are. Second, you should also note that truss element 2 transfers all the externally applied vertical loads to support `A`. The vertical reaction at support `A` is 20 kN, which is the sum of the two

externally applied loads.

Analysis of an Electrical Circuit

Problem Statement. The analysis of electrical networks composed of resistance and voltage supplies is governed by three basic principles:

1. **Ohm's Law.** The drop in voltage across a resistor R in the direction of an assumed current is proportional to the current. In other words, $V = I \cdot R$.
2. **Kirchoff's Law.** The sum of all currents entering and exiting a node must equal zero.
3. **Kirchoff's Voltage Law.** The sum of voltage drops around any closed loop must sum to zero.

Consider the circuit shown in Figure 4.5, consisting of three loops, nine resistors, and one battery.

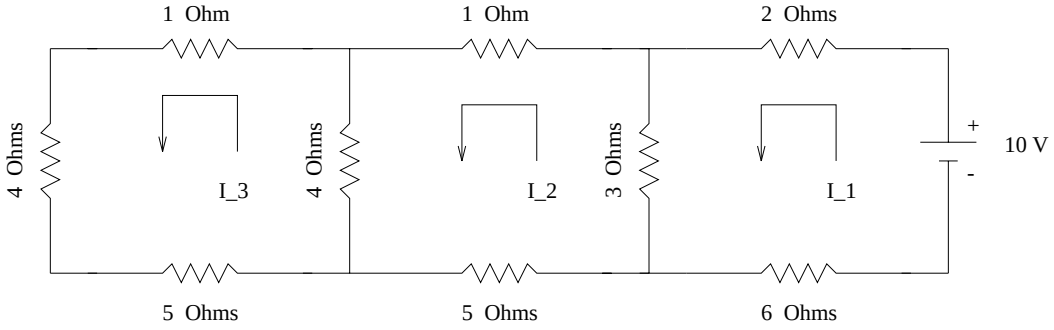


Figure 4.5. Three-loop voltage-resistance circuit.

The analysis begins with the arbitrary assignment of current directions in each of the three loops. The current in each loop will be deemed to be positive when it flows in the direction shown. For loop 1, a positive voltage change occurs between the negative and positive battery terminals. The voltage drops across the upper and lower resistors in loop one are $2I_1$ and $6I_1$, and $3 \cdot (I_1 - I_2)$ in the left-most resistor. Applying Kirchoff's Voltage law to loop 1 gives

$$6 \cdot I_1 + 2 \cdot I_1 + 3 \cdot (I_1 - I_2) = 10 \text{ V}. \tag{4.20}$$

For loop 2, Kirchoff's voltage law gives

$$I_2 + 5 \cdot I_2 + 3 \cdot (I_2 - I_1) + 4 \cdot (I_2 - I_3) = 0 \text{ V}. \tag{4.21}$$

and for loop 3, Kirchoff's voltage law gives

$$I_3 + 4 \cdot I_3 + 5 \cdot I_3 + 4 \cdot (I_3 - I_2) = 0 \text{ V}. \tag{4.22}$$

Putting Equations 4.20 to 4.22 in matrix form gives

$$\begin{bmatrix} 11 & -3 & 0 \\ -3 & 13 & -4 \\ 0 & -4 & 14 \end{bmatrix} \cdot \begin{Bmatrix} I_1 \\ I_2 \\ I_3 \end{Bmatrix} = \begin{Bmatrix} 10 \\ 0 \\ 0 \end{Bmatrix} \tag{4.23}$$

Program Source Code. Program 4.2 defines and initializes matrix Equation 4.23, and solves and prints the solution to the currents in each of the three loops.

Computer Program 4.2 : Analysis of an Electrical Circuit

```
% ========================================================================
% electrical.m -- Compute currents in an electrical circuit
%
% Matrices : Resist  = Rows represents resistors in each loop.
%          : Voltage = Voltage gain in each loop provided by battery.
%          : Current = Current in each loop.
%
% ========================================================================

% Setup matrix for "resistances in circuit loops"

  Resist = [ 11  -3   0;
             -3  13  -4;
              0  -4  14 ]

% Setup matrix for "voltage gains" in circuit loops

  Voltage = [ 10; 0; 0 ]

% Solve equations and print currents
```

```
  Current = Resist\Voltage

% ============================================================
% the end!
```

Running the Program. Assume that Program 4.2 is stored in `electrical.m`. The MATLAB script

```
>> format compact
>> electrical
Resist =
    11    -3     0
    -3    13    -4
     0    -4    14
Voltage =
    10
     0
     0
Current =
    0.9765
    0.2471
    0.0706
>>
```

shows the command needed to execute the program and the output that is generated. A (3×3) matrix of system resistances is explicitly defined and initialized in one MATLAB statement. A (3×1) matrix of system voltages is defined in second statement. With these matrices in place, solutions to Equation 4.23 are computed and printed by simply writing:

```
>> Current = Resist\Voltage
```

Of course, we could have computed the currents in each loop of the circuit by writing `inv(Resist)*Voltage`

Validating the Results. You should verify that the solution is correct by multiplying

```
>> Resist*Current
```

and checking that the result is equal to `Voltage`.

Least Squares Analysis of Experimental Data

Problem Statement. Engineers are often faced with the practical problem of having to model complex physical processes and phenomena that are not fully understood. The lack of understanding may be due to the overwhelming size of the system, or perhaps, because information about the system is missing. In an effort to better understand system behavior, many engineers design and conduct laboratory experiments, and use the experimental data in the construction of simplified empirical models. The simplified models will be based on numerous assumptions in behavior and may contain parameters that can be adjusted or modified to provide a "best fit" to the experimental data.

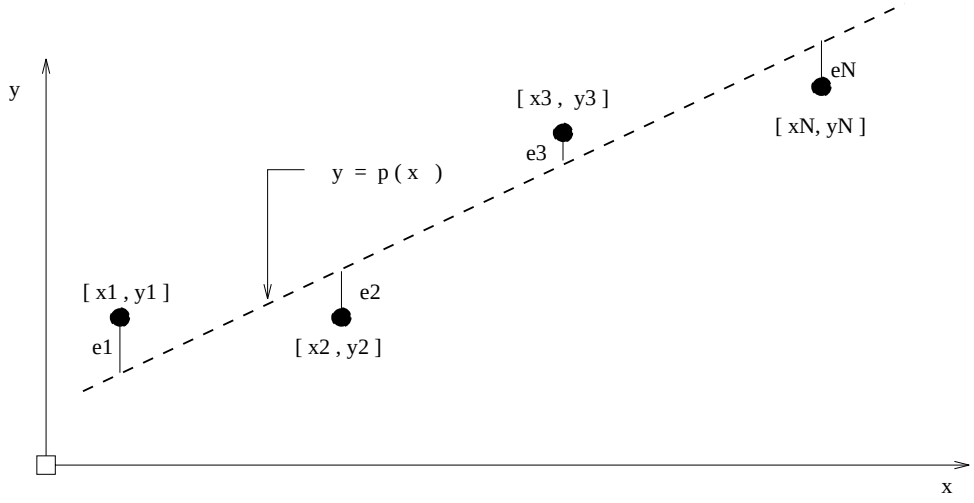


Figure 4.6. Experimental data points and line of best fit.

Figure 4.6 is a plot of experimental data points (x_1, y_1), (x_2, y_2), $\cdots$ (x_N, y_N), and a dashed-line polynomial $y = p(x)$ of best fit that has been drawn by hand. The polynomial might predict, for example, the relationship between an input signal, x, and an output signal, $y = p(x)$. Notice that approximately half of the data points deviate from the line in a positive direction (i.e., are above the line) and approximately half the data points are below the line (i.e., negative deviation).

Although it is certainly possible to find a high-order polynomial that will interpolate the data points exactly, in many engineering applications, theoretical considerations and good engineering judgment indicate that a low-order polynomial (or simpler curve) will provide a good approximation to the data. Unfortunately, there are no hard-and-fast rules for selecting a best functional form. An engineer should look for obvious trends, such as a linear, quadratic, cubic polynomial, for symmetries and antisymmetries in the data, and for periodic behavior, suggesting functional forms

containing `sin()` and `cos()` terms.

Derivation of Discrete Least Squares Equations. In this section, we formulate a mathematical procedure for fitting a polynomial curve

$$p(x) = a_o \ + \ a_1 \cdot x \ + \ a_2 \cdot x^2 \ + \ \cdots \ + \ a_n \cdot x^n \tag{4.24}$$

of degree n through N data points $(x_1, y_1), \ (x_2, y_2), \ \cdots \ (x_N, y_N)$, where $(N \ > \ n + 1)$. To avoid mathematical difficulties with positive/negative distances in error deviations, the objective of discrete least squares is to minimize the sum of the squares of the distance between the y values of the data, and $y = p(x)$. In mathematical terms, we want to minimize

$$\sum_{i=1}^{N} e_i^2 \ = \ \sum_{i=1}^{N} (y_i \ - \ p(x))^2 \tag{4.25}$$

or

$$S(a_o, a_1, \cdots, a_n) \ = \ \sum_{i=1}^{N} [y_i \ - \ a_o \ - \ a_1 \cdot x_i \ \cdots \ a_n x_i^n]^2 \tag{4.26}$$

The optimal parameter settings are given by the solution to the linear equations

$$\frac{\partial S}{\partial a_0} \ = \ 0, \ \frac{\partial S}{\partial a_1} \ = \ 0, \ \cdots \ \frac{\partial S}{\partial a_n} \ = \ 0 \tag{4.27}$$

The partial derivatives of Equations 4.27 are:

$$\left[\sum_{i=1}^{N}\right] \cdot a_o \ + \ \left[\sum_{i=1}^{N} x_i\right] \cdot a_1 \ + \ \cdots \ + \ \left[\sum_{i=1}^{N} x_i^n\right] \cdot a_n \quad = \ \textstyle\sum_{i=1}^{N} y_i \tag{4.28}$$

$$\left[\sum_{i=1}^{N} x_i\right] a_o \ + \ \left[\sum_{i=1}^{N} x_i^2\right] a_1 \ + \ \cdots \ + \ \left[\sum^{N} x_i^{n+1}\right] a_n \ = \ \textstyle\sum_{i=1}^{N} y_i \cdot x_i \tag{4.29}$$

$$\left[\sum_{i=1}^{N} x_i^n\right] a_o \ + \ \left[\sum_{i=1}^{N} x_i^{n+1}\right] a_1 \ + \ \cdots \ + \ \left[\sum^{N} x_i^{n+m}\right] a_n \ = \ \textstyle\sum_{i=1}^{N} x_i^n \cdot y_i \tag{4.30}$$

The family of $n+1$ Equations 4.28 through 4.30 is linear in the parameters $a_o, \ a_1, \cdots a_n$. In matrix form, the equations may be written

$$\begin{bmatrix} N & \sum_{i=1}^{N} x_i & \cdots & \sum_{i=1}^{N} x_i^n \\ \sum_{i=1}^{N} x_i & \sum_{i=1}^{N} x_i^2 & \cdots & \sum_{i=1}^{N} x_i^{n+1} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^{N} x_i^n & \sum_{i=1}^{N} x_i^{n+1} & \cdots & \sum_{i=1}^{N} x_i^{2n} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{N} y_i \\ \sum_{i=1}^{N} x_i \cdot y_i \\ \vdots \\ \sum_{i=1}^{N} x_i^n \cdot y_i \end{bmatrix} \tag{4.31}$$

Analysis of Experimental Data. Suppose that you have been asked to formulate an engineering model to describe the force-displacement relationship for the simple spring shown on the left-hand side of Figure 4.7. The purpose of the model is to describe the functional relationship between an applied force F and a measured displacement x.

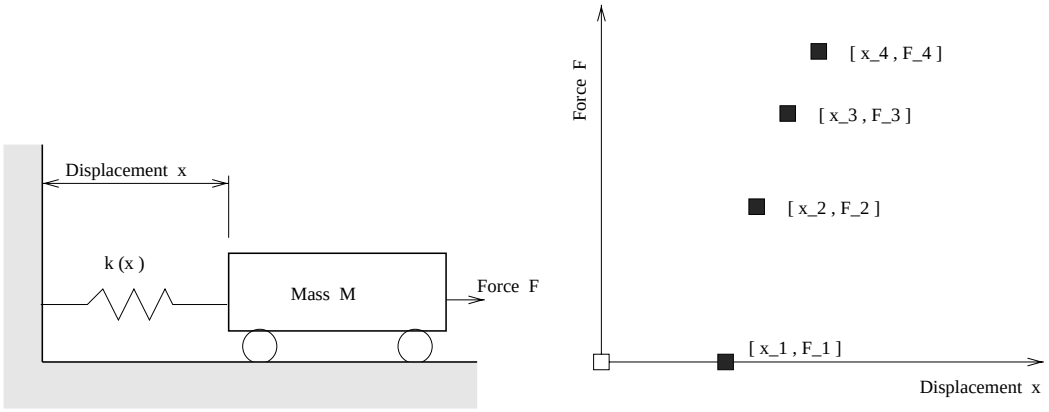


Figure 4.7. Force-displacement experiment for mass-spring system

The force-displacement model is calibrated by conducting a displacement-controlled experiment, where displacement of the mass is increased to a prescribed level and the force is measured. The experimental results are

Data Point	Displacement (cm)	Force (N)
1	5.0	0.0
2	5.5	47.5
3	6.0	90.0
4	6.5	127.5
5	7.0	160.0

```
6                7.5          187.5
7                8.0          210.0
```

A schematic of coordinate pairs (measured displacement, applied force) is plotted on the right-hand side of Figure 4.7. Theoretical considerations indicate that the force-displacement relationship is mildly nonlinear and is closely approximated by the quadratic:

$$\text{Force}(x) \;=\; a_o \;+\; a_1 \cdot x \;+\; a_2 \cdot x^2 \tag{4.32}$$

where a_o, a_1 and a_2 are coefficients to be determined via experiment and least squares analysis. When $n = 2$, Equation 4.31 takes the form:

$$\begin{bmatrix} N & \sum_{i=1}^{N} x_i & \sum_{i=1}^{N} x_i^2 \\ \sum_{i=1}^{N} \cdot x_i & \sum^{N} x_i^2 & \sum_{i=1}^{N} x_i^3 \\ \sum_{i=1}^{N} \cdot x_i^2 & \sum^{N} x_i^3 & \sum_{i=1}^{N} x_i^4 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{N} y_i \\ \sum_{i=1}^{N} x_i \cdot y_i \\ \sum_{i=1}^{N} x_i^2 \cdot y_i \end{bmatrix} \tag{4.33}$$

Program Source Code. Program 4.3 stores the experimental data in a matrix `data` and then assembles and solves the matrix Equation 4.33.

Computer Program 4.3 : Least Squares Analysis of Experimental Data

```
% ========================================================================
% leastsq.m -- Compute least squares polynomial fit on experimental data
%
% Experiment : x = displacement of spring (cm).
%              f = force in spring (N).
%
% Least squares fit : p(x) = a + b.x + c.x^2
% ========================================================================

% Store force-displacement relationship in matrix "data"

data = [ 5.0    0.0;
         5.5   47.5;
         6.0   90.0;
```

```
         6.5  127.5;
         7.0  160.0;
         7.5  187.5;
         8.0  210.0 ];

% Compute terms in least squares matrix and right-hand vector

N = 7;
sumx   = sum(data(:,1));
sumy   = sum(data(:,2));
sumxy  = sum(data(:,1).*data(:,2));

sumx2  = sum(data(:,1).*data(:,1));
sumx2y = sum(data(:,1).*data(:,1).*data(:,2));
sumx3  = sum(data(:,1).*data(:,1).*data(:,1));
sumx4  = sum(data(:,1).*data(:,1).*data(:,1).*data(:,1));

A = [      N   sumx   sumx2
        sumx  sumx2   sumx3
       sumx2  sumx3   sumx4 ]

B = [ sumy; sumxy; sumx2y]

% Compute and print constants a,b,c

Coefficients = A\B

% ============================================================
% the end!
```

Running the Program. Assume that Program 4.3 is contained the M-file `leastsq.m`. The script of MATLAB I/O

```
>> format compact
>> leastsq
A =
   1.0e+04 *
    0.0007    0.0046    0.0303
    0.0046    0.0303    0.2059
    0.0303    0.2059    1.4282
B =
   1.0e+04 *
    0.0823
    0.5836
    4.1891
Coefficients =
```

```
 -750.0000
  200.0000
  -10.0000
>>
```

shows the commands used to run the program and the output that is generated. Matrices `A` and `B` represent Equation 4.33. The matrix `Coefficients` contains the results of the least squares analysis. That is, the values for a_o, a_1, and a_2 that provide the best fit of Equation 4.32 to the experimental data.

Assembling the Least Squares Matrix Equations. Program 4.3 stores the system displacements and applied forces in columns 1 and 2 of matrix `data`. The most straightforward and, admittedly, inefficient way of computing the matrix element terms in Equation 4.33 is with blocks of MATLAB code that look similar to the following:

```
sumx2 = 0.0;
for i = 1:N
    sumx2 = sumx2 + data(i,1)*data(i,1);
end,
```

Here `sumx2` holds the sum of x_i^2 terms needed for matrix elements A(1,3), A(2,2), and A(3,1). Evaluation of this looping structure in MATLAB will be slow because it is interpreted. The same numerical result can be obtained in much less time with the single statement

```
sumx2  = sum( data(:,1).*data(:,1) );
```

Now the MATLAB function `sum` is applied to matrix element-level multiply operations on all the items in column one of matrix data. The `.*` syntax signifies matrix element-level multiply operations, and the colon (`:`) operation implies all of the items within a column of matrix `data`.

It is possible, in fact, to completely eliminate the `sumx2`-type terms from the calculation and to form the (3x3) least squares matrix in one statement block involving matrix element-level multiplies on `data`. Similar expressions can be written for matrix `B`.

Validating the Results. The polynomial coefficients $a_o = -750$, $a_1 = 200$, and $a_2 = -10$ define the force-displacement relationship

$$\text{Force}(x) = -750 + 200 \cdot x - 10 \cdot x^2 \tag{4.34}$$

$$= -10 \cdot (x - 5) \cdot (x - 15) \tag{4.35}$$

The sum of deviations

$$\sum_{i=1}^{N} e_i^2 \; = \; \sum_{i=1}^{N} (y_i \; - \; p(x))^2 \; = \; 0.0 \tag{4.36}$$

indicating that our second-order polynomial passes through the seven data points exactly (okay, we confess, we set it up that way).

Distribution of Temperature in Chimney Cross-Section

Problem Statement. In this example, we use a finite difference approximation of Laplace's equation to compute the distribution of temperature in a chimney cross-section. We select this problem because it is typical of many that naturally occur in the analysis of equilibrium states of physical systems. For example, with suitable simplifying assumptions, Laplace's equation can also describe (1) the irrotational flow of an incompressible fluid, (2) electrostatic and magnetostatic potentials, and (3) the hydraulic head associated with the steady-state flow of ground water in a uniform porous medium.

Figure 4.8 shows the front elevation and square-shaped cross-section A-A for the tall chimney. The chimney is constructed from a material that is homogeneous and isotropic (i.e., the material has the same material properties in all directions). At the cross-section, the inside and outside temperatures are $200^o\ C$ and $0^o\ C$, respectively, and there is neither a net flow of heat to or from the chimney (i.e., it is in thermal equilibrium). Finally, we assume that at the chimney cross-section, the distribution of temperature is constant along the z axis.

Because the chimney is constructed from a material that is homogeneous, thermal conductivity will not vary with position, and because the material is isotropic, thermal conductivity will not vary with direction. The steady-state distribution of temperature $T = T(x, y)$ throughout the chimney cross-section is given by solutions to Laplace's equation

$$\frac{\partial^2 T(x,y)}{\partial x^2} + \frac{\partial^2 T(x,y)}{\partial y^2} = 0. \tag{4.37}$$

with boundary conditions T = $0^o\ C$ along the exterior of the chimney, and T = $200^o C$ along the chimney interior.

Finite Difference Mesh for Chimney Cross-Section

The chimney cross-section is symmetric about the x- and y-axes, and the two diagonal axes. Our computational model takes advantage of symmetries about the x- and y-axes by modeling only

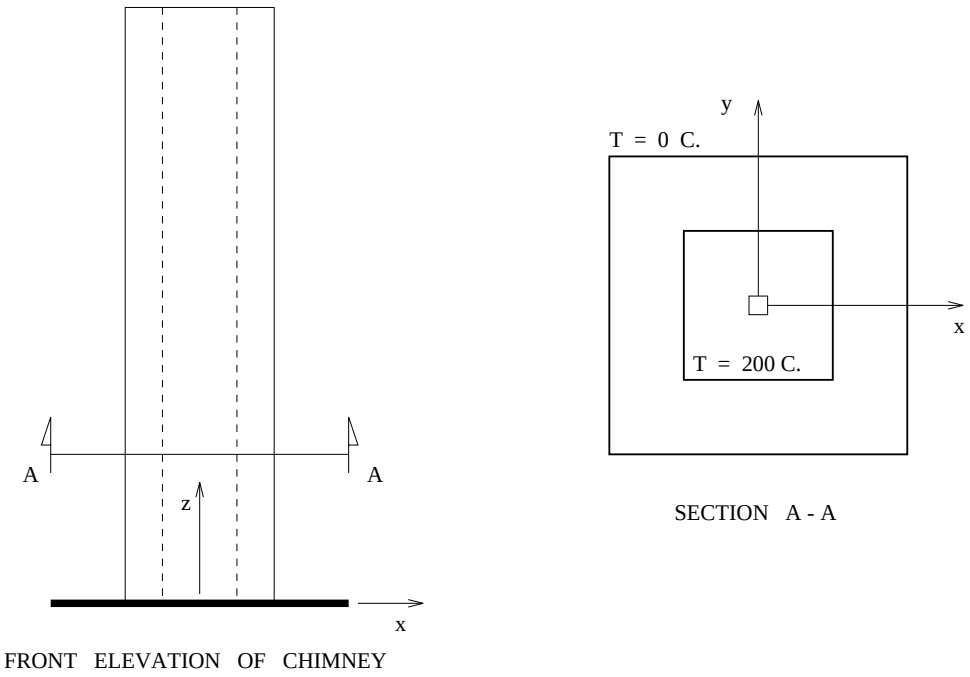


Figure 4.8. Front elevation and cross-section of tall chimney.

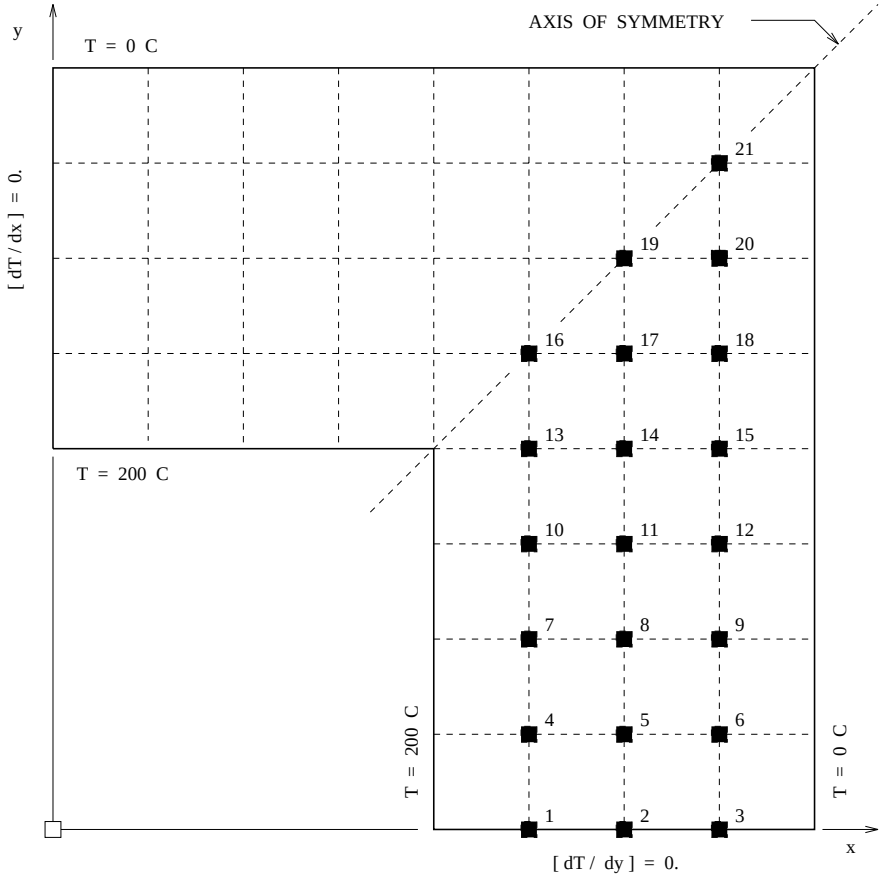
one quarter of the chimney cross section, as shown in Figure 4.9a. Two new boundary conditions are needed for this model; along the y-axis the temperature gradient $dT/dx = 0$, and along x-axis $dT/dy = 0$.

If dx and dy are the mesh distance in the x-axis and y-axis directions, then a suitable finite difference approximation to Equation 4.37 is

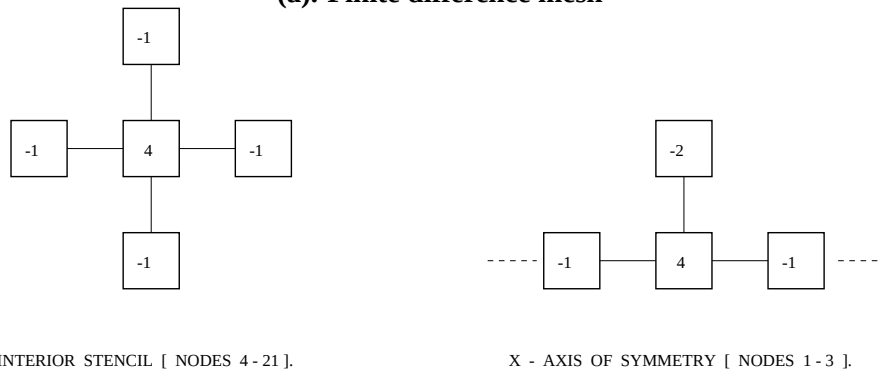
$$\left[\frac{T(x+dx,y) - 2T(x,y) + T(x-dx,y)}{dx^2}\right] + \left[\frac{T(x,y+dy) - 2T(x,y) + T(x,y-dy)}{dy^2}\right] = 0. \tag{4.38}$$

Equation 4.38 is simply the two-dimensional counterpart of the finite difference approximation derived for the cable profile problem. If dx = dy, then Equation 4.38 can be rearranged to give

$$4T(x,y) - T(x-dx,y) - T(x+dx,y) - T(x,y+dy) - T(x,y-dy) = 0. \tag{4.39}$$



(a). Finite difference mesh



(b). Finite difference stencils

Figure 4.9. Finite difference mesh and stencils for Equation 4.37

The leftmost schematic of Figure 4.9b shows the weighting of discrete temperatures in the finite difference approximation. The nodes along the x-axis (i.e., y = 0) satisfy the finite difference equation

$$4 \cdot T(x,0) - T(x-dx,0) - T(x+dx,0) - 2T(x,dy) = 0 \tag{4.40}$$

The leftmost schematic of Figure 4.9b shows the and along the y-axis (i.e., x = 0)

$$4 \cdot T(0,y) - 2T(dx,y) - T(0,y+dy) - T(x,y-dy) = 0 \tag{4.41}$$

The finite difference mesh has 65 nodes, 26 of them being on the interior (i.e., T = O^o C) and exterior (i.e., T = 200^o C) boundaries. This leaves 39 nodes on the chimney interior for evaluation. Instead of evaluating the temperature stencils at all 39 interior nodes, we compute temperature only at the 21 nodes labeled with small filled black boxes and fill in the remaining unknowns by noting the symmetry in temperature along the line $x = y$. The four-node stencil is used at nodes 1 to 3, and the five-node stencil is used at nodes 4 to 21.

Program Source Code. Our solution to this problem uses the "method of iteration" to compute the steady-state temperature profile at the internal nodes. Once the temperature profile is known we create a two-dimensional color contour plot of the temperature distribution inside the chimney wall.

Computer Program 4.4 : Distribution of Temperature in Chimney Cross-Section

```
%  ============================================================
%  chimney.m -- Compute and displace profiles of temperature in
%               chimney cross section.
%  ============================================================

%  Setup working array and boundary conditions along
%  internal/external walls.

 T = zeros(9,9);

 for i = 5:9;
     T(i,5) = 200;
 end;
 for i = 1:5;
     T(5,i) = 200;
 end;
 for i = 6:9;
```

```
 for j = 1:4;
     T(i,j) = NaN;
 end;
 end;

% Loop over internal nodes and compute new temperatures

 counter   = 0;
 maxchange = 200;
    while (maxchange > 1)
        counter = counter+1;
        maxchange = 0;
        k=5;
        l=4;
        for c = 6:8;
            newtemp    = 0.25*(2*T(l+1,c)+2*T(l,c+1));
            tempchange = newtemp - T(l,c);
            maxchange  = max(maxchange,abs(tempchange));
            T(l,c)=newtemp;
           for r=k:8
               newtemp    = 0.25*(T(r,c-1)+T(r,c+1)+T(r-1,c)+T(r+1,c));
               tempchange = newtemp - T(r,c);
               maxchange  = max(maxchange,abs(tempchange));
               T(r,c)= newtemp;
           end
           newtemp    = 0.25*(T(9,c-1)+T(9,c+1)+2*T(8,c));
           tempchange = newtemp - T(9,c);
           maxchange  = max(maxchange,abs(tempchange));
           T(9,c)=newtemp;
           l=l-1;
           k=k-1;
        end
        counter;
        maxchange;       % to view counter or maxchange remove
   end

% Compute reflected temperature

  for i = 2:4
      for j = 1: 11-i
          T(i,j) = T(10-j,10-i);
      end
  end

% Print temperature array.

T
```

```
% Plot temperature contours.

contour(T)
hold;

% Now overlay perimeter of chimney section on contours.

perim = [ 1 , 1;
          9 , 1;
          9 , 9;
          5 , 9;
          5 , 5;
          1 , 5;
          1 , 1 ];

plot(perim(:,1),perim(:,2),'w');
text(1.1,5.3,'Temp = 200.0');
text(7.0,1.3,'Temp = 0.0');

%  ============================================================
%  The End!
```

Running the Program. Assume that Program 4.4 is contained in the M-file `chimney.m`. The abbreviated script of MATLAB I/O:

```
>> format compact
>> chimney

   .... lots of program output removed ....

T =

  Columns 1 through 7

         0         0         0         0         0         0         0
   47.9527   47.4495   45.9241   42.8111   37.2518   28.7560   19.0375
   96.9120   96.1208   93.6699   88.3435   77.7632   59.1105   38.4899
  147.6536  147.0166  144.9543  139.9098  127.2653   92.5014   59.1105
  200.0000  200.0000  200.0000  200.0000  200.0000  127.2653   77.7632
       NaN       NaN       NaN       NaN  200.0000  139.9098   88.3435
       NaN       NaN       NaN       NaN  200.0000  144.9543   93.6699
       NaN       NaN       NaN       NaN  200.0000  147.0166   96.1208
       NaN       NaN       NaN       NaN  200.0000  147.6536   96.9120

  Columns 8 through 9
```

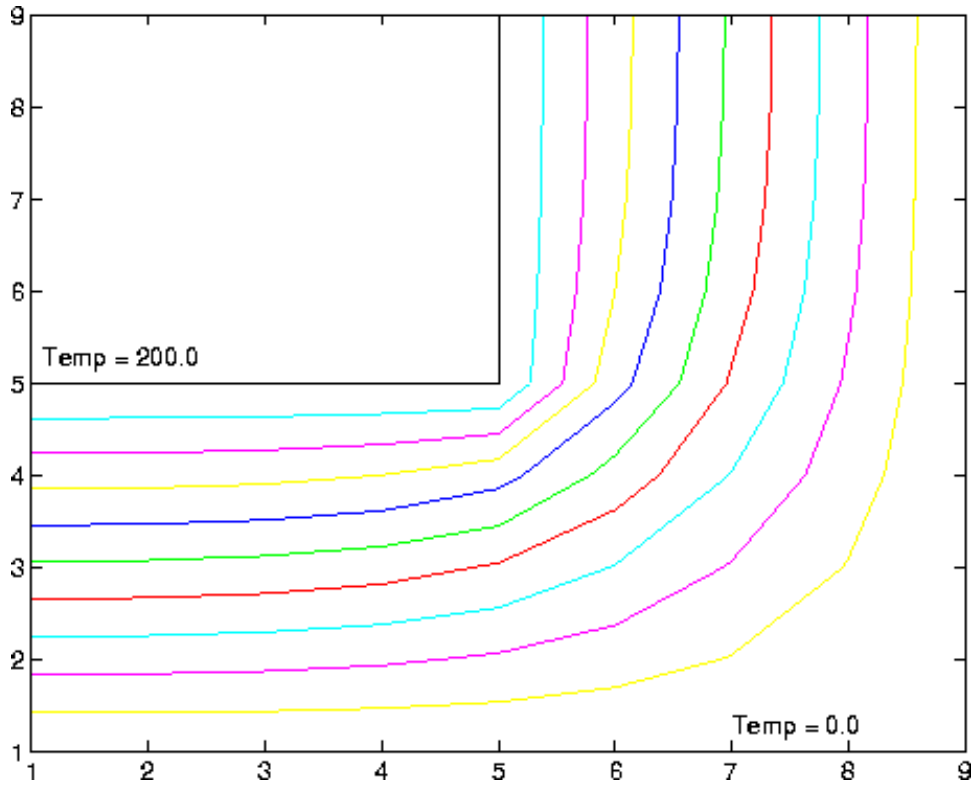


Figure 4.10. Temperature profile in chimney.

```
         0         0
    9.3215         0
   19.0375         0
   28.7560         0
   37.2518         0
   42.8111         0
   45.9241         0
   47.4495         0
   47.9527         0
>>
```

shows the commands needed to run the chimney analysis and compute the distribution of temperature throughout the chimney cross-section. Figure 4.10 is two-dimensional contour plot of chimney temperature.

Computational Procedure. The first block of code sets up a (9x9) matrix for modeling one fourth of the chimney cross-section. The temperature along the interior and exterior walls is set to 200 and 0 degrees, respectively. In MATLAB the interior region of the chimney can be represented with NaNs. That is, a missing data item. At this point in the program execution, the contents of matrix T are

```
T =

     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0
   200   200   200   200   200     0     0     0     0
   NaN   NaN   NaN   NaN   200     0     0     0     0
   NaN   NaN   NaN   NaN   200     0     0     0     0
   NaN   NaN   NaN   NaN   200     0     0     0     0
   NaN   NaN   NaN   NaN   200     0     0     0     0
```

The main block of code walks along columns 6 through 8 and evaluates the finite difference stencils for

```
Column No            Row Nos
============================
        6        4 through 9
        7        3 through 9
        8        2 through 9
============================
```

For example, after the algorithm has walked along column 6 for the first time, the contents of T are

```
T =

  Columns 1 through 7

       0         0         0         0         0         0         0
       0         0         0         0         0         0         0
       0         0         0         0         0         0         0
       0         0         0         0         0         0         0
 200.0000  200.0000  200.0000  200.0000  200.0000   50.0000         0
     NaN       NaN       NaN       NaN  200.0000   62.5000         0
     NaN       NaN       NaN       NaN  200.0000   65.6250         0
     NaN       NaN       NaN       NaN  200.0000   66.4062         0
     NaN       NaN       NaN       NaN  200.0000   83.2031         0

  Columns 8 through 9
```

```
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
```

Bear in mind that the temperature at stencil T(4,6) is still zero because all the neighboring stencils are initially zero. The three-line block of code

```
newtemp    = 0.25*(T(9,c-1)+T(9,c+1)+2*T(8,c));
tempchange = newtemp - T(9,c);
maxchange  = max(maxchange,abs(tempchange));
```

computes the new temperature estimate at the node, the change in node temperature from the previous iteration, and the maximum change in temperature occurring over rows 6 through 8 for the current iteration.

The outermost loop of the algorithm will iteratively refine the temperature profile until satisfactory convergence occurs. For this example, we stop refining the temperature profile when the maximum change in temperature over rows 6 through 8 is less than 1 degree. At the conclusion of the main block of code, the temperature profile is

```
T =

   Columns 1 through 7

         0         0         0         0         0         0         0
         0         0         0         0         0         0         0
         0         0         0         0         0         0   38.4899
         0         0         0         0         0   92.5014   59.1105
  200.0000  200.0000  200.0000  200.0000  200.0000  127.2653   77.7632
       NaN       NaN       NaN       NaN  200.0000  139.9098   88.3435
       NaN       NaN       NaN       NaN  200.0000  144.9543   93.6699
       NaN       NaN       NaN       NaN  200.0000  147.0166   96.1208
       NaN       NaN       NaN       NaN  200.0000  147.6536   96.9120

   Columns 8 through 9

         0         0
    9.3215         0
```

```
19.0375         0
28.7560         0
37.2518         0
42.8111         0
45.9241         0
47.4495         0
47.9527         0
```

The final temperature profile is obtained by reflecting the temperatures along the line y = x.

Of course, the temperature profile may also be computed by writing and solving the finite difference equations in matrix form (see Problem 4.8).

4.7 Review Questions

1. Explain how a system of m linear equations containing n unknowns can be represented in matrix form.

2. What are the three types of solutions matrix equations can have?

3. What role does the matrix determinant play in determining whether a family of matrix equations will have a unique solution.

4. Let A be a $(n \times n)$ matrix and B be a $(n \times 1)$ matrix. Under what conditions will the solution to $A.X = B$ have an infinite number of solutions? How would you use MATLAB to detect this situation?

5. Suppose that a family of three equations, each having three unknowns, is graphed in three-dimensional space and that it is immediately apparent that one of the equations is a linear combination of the remaining two. If the equations are written in matrix form, what can you say about (1) the matrix rank, (2) the matrix determinant, and (3) the matrix inverse?

4.8 Programming Exercises

4.1 **Beginner.** Suppose that the cable profile of a small suspension bridge carrying a uniformly distributed load

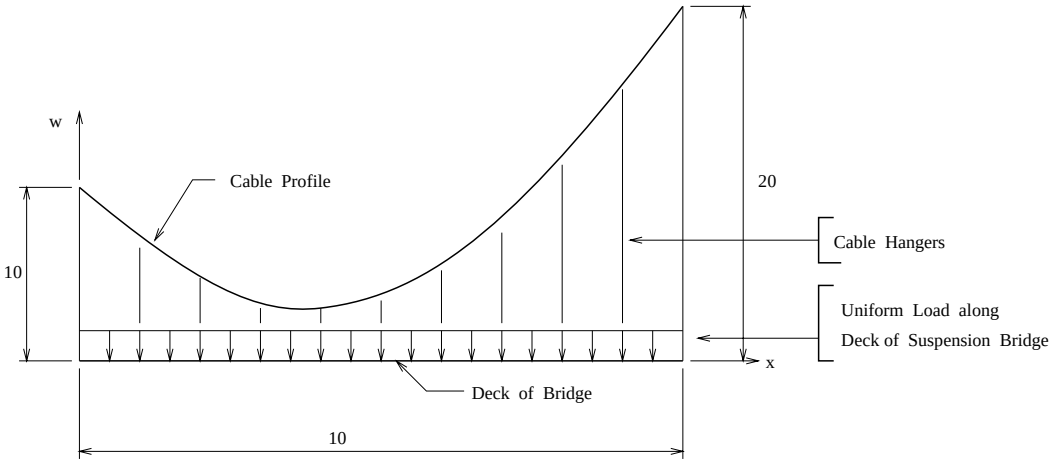


DIAGRAM NOT TO SCALE

corresponds to the solution of the differential equation

$$\frac{d^2w}{dx^2} = 1.0 \tag{4.42}$$

with the boundary conditions $w(0) = 10$ and $w(10) = 20$. It is easy to show that the analytic solution to the cable profile is

$$w(x) = \frac{1}{2}x^2 - 4x + 10 \tag{4.43}$$

Now solve Equation 4.42 via the method of finite differences.

1. What is a suitable finite difference approximation to Equation 4.42?.
2. If the cable profile is divided into five regions along the x-axis, with four internal nodes, write down the family of finite difference equations that you would solve for the cable profile (do not try to find a solution to these equations).

3. Write a MATLAB program to solve the family of equations by the "method of iteration."
4. Write down the family of linear matrix equations corresponding to this finite difference problem. Write a MATLAB program that computes the solution to these equations, and then displays the numerical solution and Equation 4.43 on the same graph.

4.2 **Beginner.** Figure 4.8 shows a three-loop voltage-resistance circuit, containing one battery and seven resistors.

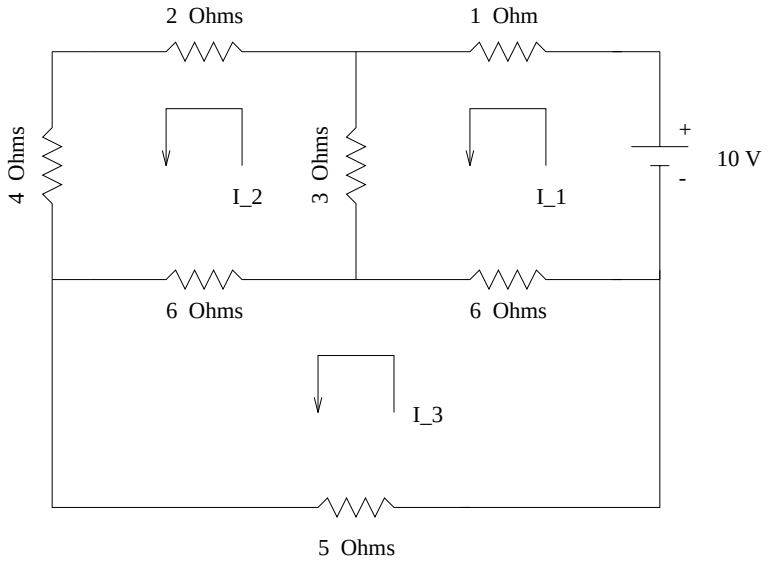


Figure 4.11. Three-loop voltage-resistance circuit.

Write a MATLAB program to compute and print the magnitude of current flows in each of the three loops. For each loop, assume that anticlockwise current flow is positive.

4.3 **Intermediate.** In the solution of many fluid mechanics and chemical engineering problems, conservation of mass is a central principle. Briefly stated, conservation of mass accounts for all sources and sinks of a material that pass in and out of a control volume (see the left-hand side of Figure 4.12).

For a specified interval of time, the accumulation of substance is simply the sum of the inputs minus the sum of the outputs. When the sum of the inputs equals the sum of the outputs, accumulations are zero, and the mass within the volume will be constant. Since the mass within the volume does not change with time, we say that such a system is in steady state.

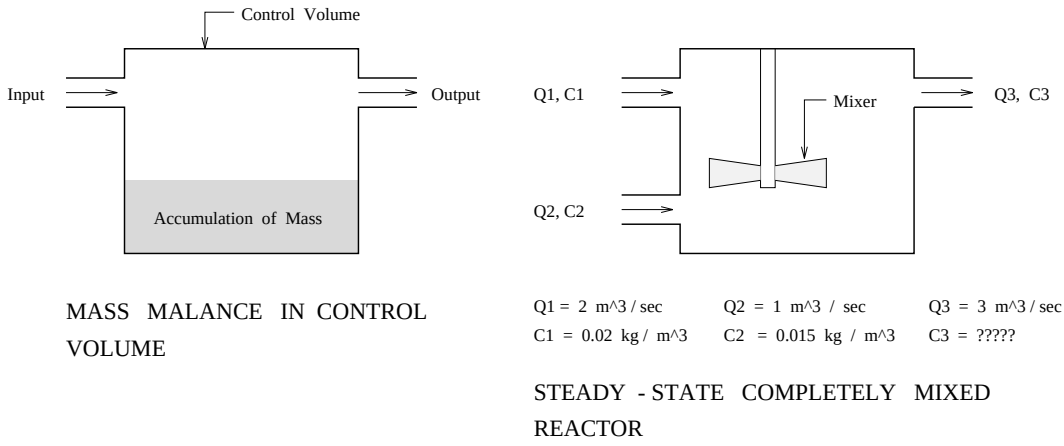


Figure 4.12. Conservation of mass in fully mixed reactor.

The principle of conservation of mass can be used to determine the concentrations of substances in system of coupled fully mixed reactors. To see how the analysis proceeds, let's first look at the single fully mixed reactor shown on the right-hand side of Figure 4.12. The reactor has one input pipe and two output pipes. You should observe that the concentration at the output pipe is not shown because it can be computed via the principle of conservation of mass.

The mass of substance passing through each pipe is simply the flow rate, Q (m^3/sec), multiplied by the concentration of substance C (kg/m^3). For a system in steady state (where the mass does not increase or decrease due to chemical reactions), conservation of mass requires

$$C_1 \cdot Q_1 + Q_2 \cdot C_2 = Q_3 \cdot C_3. \tag{4.44}$$

Hence, the concentration of mass in the output pipe is

$$C_3 = \left[\frac{(Q_1 \cdot C_1 + Q_2 \cdot C_2)}{Q_3}\right] = \left[\frac{0.055}{3}\right] kg/m^3. \tag{4.45}$$

Exactly the same principles can be used to compute the concentration of substances in the network of fully mixed reactors shown in Figure 4.13. The concentrations of mass in reactors

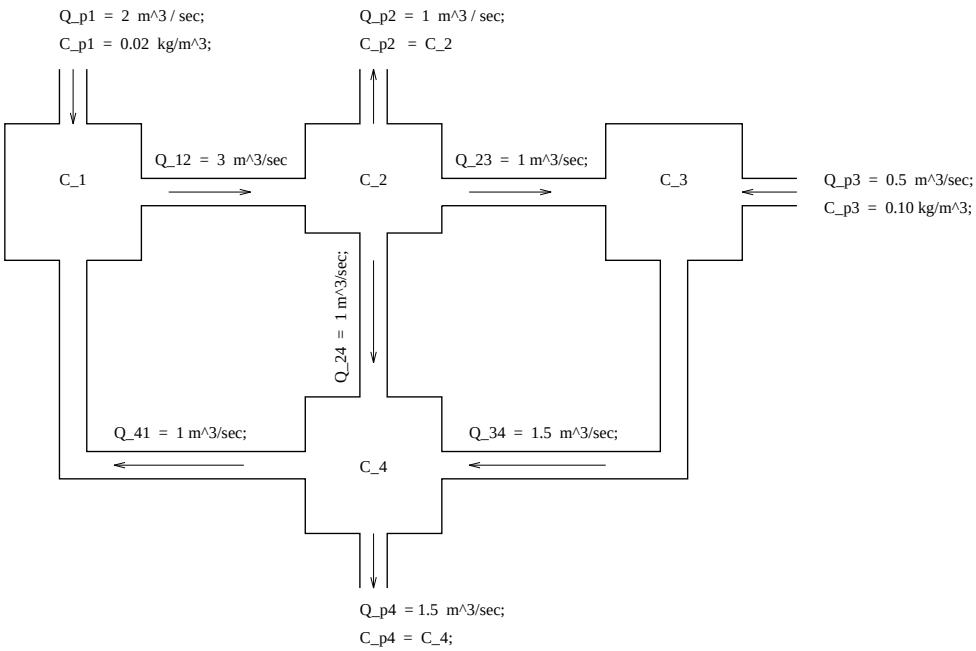


Figure 4.13. Network of four fully mixed reactors.

1 through 4 are denoted by the symbols C_1, C_2, C_3 and C_4. Because there are four reactors, four simultaneous mass-balance equations are needed to describe the distribution of substance concentrations.

1. Show that the mass-balance equations may be written

$$\begin{bmatrix} Q_{12} & 0 & 0 & -Q_{41} \\ Q_{12} & -Q_{12} & 0 & 0 \\ 0 & -Q_{23} & Q_{34} & 0 \\ 0 & Q_{24} & Q_{34} & -(Q_{p4}+Q_{41}) \end{bmatrix} \cdot \begin{Bmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \end{Bmatrix} = \begin{Bmatrix} Q_{12} \cdot C_{p1} \\ 0 \\ Q_{p3} \cdot C_{p3} \\ 0 \end{Bmatrix} \quad (4.46)$$

2. Develop a MATLAB program to solve Equations 4.46 for the concentrations in each reactor.

4.4 **Intermediate.** In the design of highway bridge structures and crane structures, engineers are often required to compute the maximum and minimum member forces and support reactions due to a variety of loading conditions.

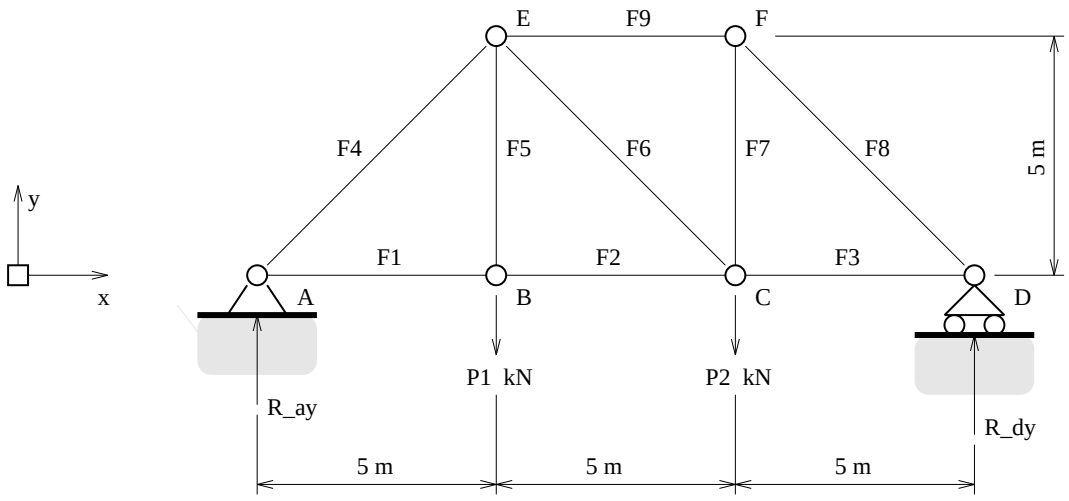


Figure 4.14. Front elevation of pin-jointed bridge truss.

Figure 4.14 shows a nine bar pin-jointed bridge truss carrying vertical loads P_1 kN and P_2 kN at joints B and C. The symbols F_1, F_2, $\cdots$ F_9 represent the axial forces in truss members 1 through 9, and R_{ay} and R_{dy} are the support reactions at joints A and D. (Notice that because support D is on a roller and there are no horizontal components of external loads, horizontal reactions will be zero.)

Write down the equations of equilibrium for joints B through F and put the equations in matrix form. Now suppose that a heavy load moves across the bridge and that, for engineering purposes, it can be represented by the sequence of external load vectors

$$\begin{bmatrix} P_1 \\ P_2 \end{bmatrix} = \begin{bmatrix} 10 \\ 0 \end{bmatrix}, \begin{bmatrix} P_1 \\ P_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 5 \end{bmatrix}, \begin{bmatrix} P_1 \\ P_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 10 \end{bmatrix} \tag{4.47}$$

Develop a MATLAB program that will solve the matrix equations for each of the external load conditions, and compute and print the minimum and maximum axial forces in each of the truss members.

4.5 **Intermediate-Advanced.** In the detailed stages of a petroleum refinery design, an experiment is conducted to determine the empirical relationship between solubility weight (%) of n-butane in anhydrous hydrofluoric acid at high pressures and temperature. A plot of the experimental data

Data Point	Temperature (C)	Solubility (%)
1	25	2.5
2	38	3.3
3	85	7.1
4	115	11.0
5	140	19.7

on semilog graph paper indicates that solubility and temperature follow the nonlinear relationship

$$\text{Solubility } s(t) \;=\; a_o e^{a_1 \cdot t}. \tag{4.48}$$

where a_o and a_1 are parameters to be determined. A linear least squares problem can be obtained by applying the transformation $\log_e(s(t)) = \log_e(a_o) + a_1 \cdot t$.

1. Show that the least squares estimate of parameters a_o and a_1 is given by solutions to the matrix equations

$$\begin{bmatrix} N & \sum_{i=1}^{N} t_i \\ \sum_{i=1}^{N} t_i & \sum_{i=1}^{N} t_i^2 \end{bmatrix} \cdot \begin{bmatrix} \log_e(a_o) \\ a_1 \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{N} \log_e(s_i) \\ \sum_{i=1}^{N} t_i \cdot \log_e(s_i) \end{bmatrix} \tag{4.49}$$

2. Write a MATLAB program to compute parameters a_o and a_1 by solving matrix Equation 4.49.

4.6 **Intermediate.** Figure 4.15 is a three-dimensional view of a 2 by 2 km site that is believed to overlay a thick layer of mineral deposits.

To create a model of the mineral deposit profile and establish the economic viability of mining the site, a preliminary subsurface exploration consisting of 16 bore holes is conducted. Each bore hole is drilled to approximately 45 m, with the upper and lower boundaries of mineral deposits being recorded. The bore hole data is as follows:

Borehole	[x, y] coordinate	[upper, lower] mineral surfaces
1	[10.0 m, 10.0 m]	[-30.5 m, -40.5 m]
2	[750.0 m, 10.0 m]	[-29.0 m, -39.8 m]
3	[1250.0 m, 10.0 m]	[-28.0 m, -39.3 m]
4	[1990.0 m, 10.0 m]	[-26.6 m, -38.5 m]
5	[10.0 m, 750.0 m]	[-34.2 m, -41.4 m]

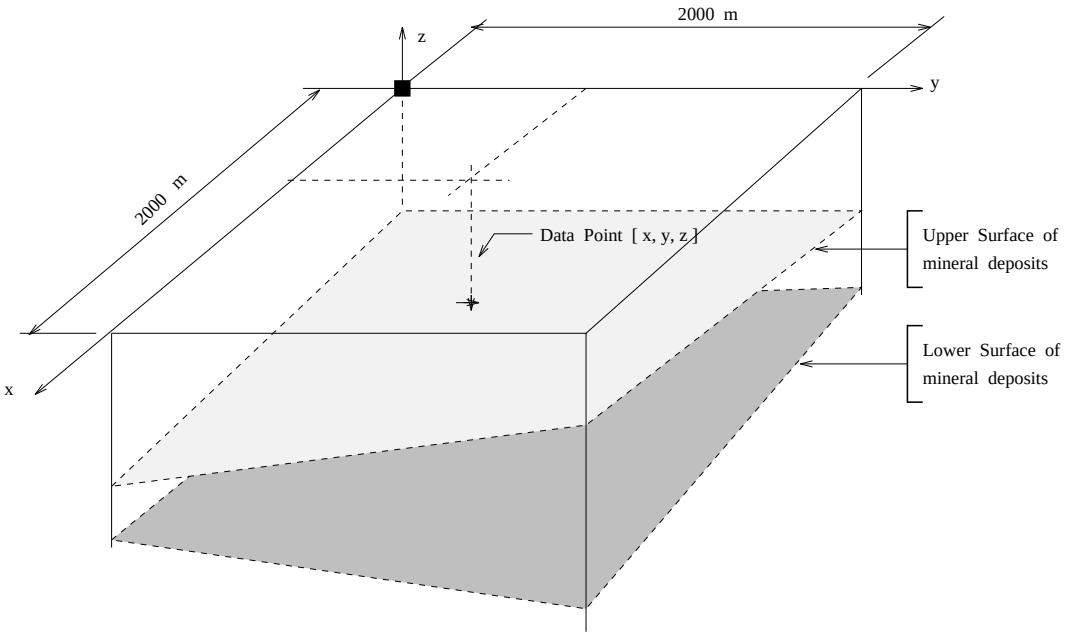


Figure 4.15. Three-dimensional view of mineral deposits.

```
 6     [  750.0 m,  750.0 m ]       [ -32.8 m,  -40.6 m ]
 7     [ 1250.0 m,  750.0 m ]       [ -31.8 m,  -40.1 m ]
 8     [ 1990.0 m,  750.0 m ]       [ -30.3 m,  -39.4 m ]
 9     [   10.0 m, 1250.0 m ]       [ -36.7 m , -42.0 m ]
10     [  750.0 m, 1250.0 m ]       [ -35.2 m,  -41.2 m ]
11     [ 1250.0 m, 1250.0 m ]       [ -34.2 m,  -40.7 m ]
12     [ 1990.0 m, 1250.0 m ]       [ -32.8 m,  -40.0 m ]
13     [   10.0 m, 1990.0 m ]       [ -40.4 m,  -42.8 m ]
14     [  750.0 m, 1990.0 m ]       [ -39.0 m,  -42.1 m ]
15     [ 1250.0 m, 1990.0 m ]       [ -38.0 m,  -41.6 m ]
16     [ 1990.0 m, 1990.0 m ]       [ -36.5 m,  -40.9 m ]
```

With the bore hole data collected, the next step is to create a simplified three-dimensional computer model of the site and subsurface mineral deposits. The mineral deposits will be modeled as a single six-sided object. The four vertical sides are simply defined by the boundaries of the site. The upper and lower sides are to be defined by a three-dimensional plane

$$z(x, y) = a_o + a_1 \cdot x + a_2 \cdot y \tag{4.50}$$

where coefficients a_o, a_1, and a_2 correspond to minimum values of

$$S\left(a_o,\ a_1,\ a_2\right) = \sum_{i=1}^{N} \left[z_i - z(x_i, y_i)\right]^2 \tag{4.51}$$

Things to do:

1. Show that minimum value of $S(a_o, a_1, a_2)$ corresponds to the solution of the matrix equations

$$\begin{bmatrix} N & \sum_{i=1}^{N} x_i & \sum_{i=1}^{N} y_i \\ \sum_{i=1}^{N} x_i & \sum^{N} x_i^2 & \sum_{i=1}^{N} x_i \cdot y_i \\ \sum_{i=1}^{N} y_i & \sum^{N} x_i \cdot y_i & \sum_{i=1}^{N} y_i^2 \end{bmatrix} \cdot \begin{bmatrix} a_o \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{N} z_i \\ \sum_{i=1}^{N} x_i \cdot z_i \\ \sum_{i=1}^{N} y_i \cdot z_i \end{bmatrix} \tag{4.52}$$

2. Write an M-file that will create three-dimensional plots of the borehole data at the lower and upper surfaces.
3. Write an M-file that will set up and solve the matrix equations derived in part 1 for the upper and lower mineral planes.
4. Compute and print the average depth and volume of mineral deposits enclosed within the site.

Note. The least squares solution corresponds to the minimum value of function $S(a_o, a_1, a_2)$. At the minimum function value, we will have

$$\frac{dS}{da_o} = \frac{dS}{da_1} = \frac{dS}{da_2} = 0 \tag{4.53}$$

Matrix Equation 4.52 is simply the three equations 4.53 written in matrix form. You should find that the equation of the upper surface is close to `z(x,y) = -30.5 + x/500 - y/200` and the lower surface close is to `z(x,y) = -40.5 + x/1000 - y/850`.

4.7 **Intermediate.** Repeat the "chimney temperature" problem using the following problem-solving procedure:

1. Write an M-file that sets up the finite difference equations in matrix form and then computes a solution by solving A.T = B, where T is the temperature at the internal nodes of the chimney.
2. Create a three-dimensional mesh (or surface) plot of the temperature distribution in one fourth of the chimney cross section.

4.8 **Intermediate.** Figure 4.16 shows the cross-section of a long conducting metal box with a detached lid.

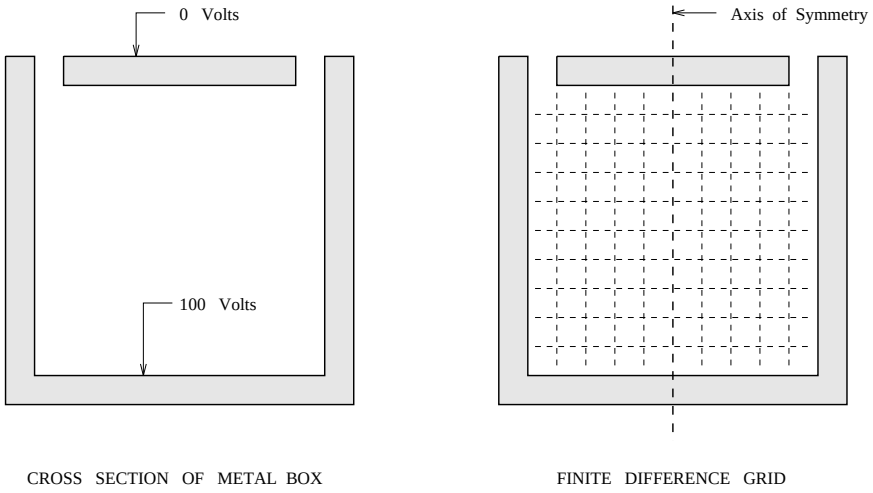


Figure 4.16. Cross-section of tall (infinite) metal box.

The sides and bottom of the box are at 100 Volts potential, and the lid is at ground (0 V) potential. Laplace's equation and the method of finite differences can be used to compute the distribution of potential inside the box. The solution procedure is almost identical to the chimney problem described in the chapter, but with temperature changed to voltage.

The box is assumed to extend to infinity in the z direction (so that there are no "edge effects" to consider). This simplifies the problem to two dimensions (x and y). The static distribution of voltage, $V = V(x, y)$, inside the metal box is given by solutions to Laplace's equation

$$\frac{\partial^2 V(x,y)}{\partial x^2} + \frac{\partial^2 V(x,y)}{\partial y^2} = 0 \tag{4.54}$$

with boundary conditions V = 0 V on the top or lid of the box and V = 100 V along both sides and the bottom of the box. Write a MATLAB program that will

1. Compute the voltage distribution inside the box via the method of finite differences described in Chapter 8 of the C tutorial.
2. Plot a contour map of the voltage potential.
3. **Optional.** Change the potential on the walls and lid of the box relative to each other and show how the voltage distribution changes.

Bibliography

[1] Arnold, K., Gosling, J. *The Java Programming Language*. Addison-Wesley, Reading, MA 01867, 1996.

[2] Boehm, B.W. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[3] Booch, G. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, Redwood City, CA 94065, 2nd edition, 1994.

[4] Brooks, F. *The Mythical Man-Month*. Addison-Wesley, 1975.

[5] Clements, P. C., Parnas, P. L., Weiss, D. M. The modular structure of complex systems. *Proc 7th International Conf. on Software Engineering*, pages 408–417, March 1984.

[6] Dongarra, J.J., Bunch, J.J., Moler, C.B., Stewart, G.W. LINPACK User's Guide. *SIAM*, 1979.

[7] East, S. *Systems Integration – A Management Guide for Manufacturing Engineers*. McGraw-Hill, 1994.

[8] Linton M. dbx. Technical report, Berkeley, CA 94720, 1982.

[9] Meyer, B. *Object-oriented Software Construction*. Prentice-Hall International Series in Computer Science, Hertfordshire, UK, 1988.

[10] Nievergelt, J., Hinrichs, K.H. *Algorithms and Data Structures : With Applications to Graphics and Geometry*. Prentice-Hall, Englewood Cliffs, NJ 07632, 1993.

[11] Osterhout, J.K. *Tcl and the Tk Toolkit*. Addison-Wesley Professional Computing Series, Reading, MA 01867, 1994.

[12] Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:330–336, December 1972.

[13] Press, L. Personal computing : Technetronic education : Answers on the cultural horizon. *Communications of the ACM*, 36(5):17–22, May 1993.

[14] Royce, W.W. Managing the development of large software systems. In *Proceedings of the IEEE WESCON*, August 1970.

[15] Smith, B.T., Boyle, J.M., Ikebe, Y., Klema, V.C., Moler, C. *Matrix Eigensystem Routines : EISPACK Guide*. Springer-Verlag, 2nd edition, 1970.

[16] Tesler, L.G. Networked computing in the 1990's. *Scientific American*, 265(3):86–93, September 1991.

[17] Wall, L., Christiansen, T., Schwartz, R. *Programming Perl*. O'Reilly and Associates, Sebastopol, CA 95472, 2nd edition, 1993.