MAT2409

High Performance Numerical Computing

Faculty of Health, Engineering and Sciences

Study Book

Written by

Dr Leigh Brookshaw, Dr Harry Butler, Prof. Tony Roberts, Dr Walter Spunde
School of Agricultural, Computational and Environmental Sciences
mailto:harry.butler@usq.edu.au
Faculty of Health, Engineering and Sciences
The University of Southern Queensland

Distributed by

The University of Southern Queensland
Toowoomba, Queensland 4350
Australia
http://www.usq.edu.au

Produced using XeLaTeX by the School of Agricultural, Computation & Environmental Sciences, http://www.usq.edu.au, in a style with minor adaptations from a base of the refman package (v2.0e) by Kielhorn & Partl, to implement Wendy Priestly's *Instructional typographies using desktop publishing techniques to produce effective learning and training materials,* http://www.ascilite.org.au/ajet/ajet7/priestly.html.

Version control information
Git Version: 0bf0313
Git Commit Date: 2018-02-09 13:53:56 +1000

Table of Contents

Index

Preface

This course in High Performance Numerical Computing deals with generic issues in efficient practical computation.

To develop and illustrate techniques we use the specific problems of finding the values of functions and their integrals and derivatives when they can not be described exactly in terms of the standard functions of mathematics. The function values may come from raw data or the solution of algebraic or differential equations and we will describe processes for interpolating between data points and for finding approximate solutions for algebraic or differential equations. The computations are typically beyond the capacity of a person with a simple calculator and require significant computing power. The tool we use for this is the software package MATLAB.

A study of MATLAB and the methods of programming are included in the Appendix. However, you should integrate your study of MATLAB programming techniques with the numerical methods throughout the course.

MATLAB's evolution in response to commercial opportunities has made it a tool that is very easy to use, with spectacular graphic capabilities and many toolboxes for various engineering applications. MATLAB's pedigree ensures that its numerical routines have been thoroughly tested and improved through many years of use. However, that does not mean that they can be used without concern. Having a knowledge of numerical techniques and a proficiency in MATLAB will greatly enhance the power and confidence of any person working in a scientific or engineering environment. Mastering this course is well worth the effort.

Bibliography

[Ano01] Anonymous. IA-32 Intel architecture software developer¹s manual volume 1: Basic architecture. Technical report, Intel Corporation, 2001. 15, 16

Chapter 1 Computer architecture and its influence on program efficiency

Chapter contents

1.1 Introduction

There are several different computer architectures in use today. The most common of these are found in home workstations. The most common computer architecture is based around the INTEL™ (and AMD™) type processors. Due to the wide diversity of architectures, programmers focus on the portability of their programs from one architecture to another by using a high level language like MATLAB.

While portability issues are important considerations, in most applications, such as games and modelling, program efficiency and speed of execution are crucial. A smart programmer who understands the underlying hardware can always squeeze out a little more performance.

Some of the code modification to achieve peak performance are generic and come from writing good code, others come from understanding the hardware the program is running on. The purpose of this module, is to empower you to recognise hardware components that play a role in determining the performance of a computer system.

The fastest known man-made digital computers are recorded on the web page `http://performance.netlib.org` using a standard computation called the Linpack Benchmark. These and other sources have monitored the exponential increase in computer speed for decades. Nonetheless, the amazing increases in raw speed have been matched by improvements in algorithm design. Although we start

by briefly looking at some hardware, always remember that *algorithms are equally as important as hardware in humans being able to solve large and challenging computational problems.*

1.2 Computer architecture overview

The system speed is ultimately determined by hardware components, such as the Central Processing Unit (CPU) and memory, and how these interact. Over the last twenty years significant improvements in system speed has been by improving electronic components. For example the valves in the early computers have been replaced by more efficient transistors. Each technology imposes its own physical limits on performance and within those bounds, further improvements can only come from writing programs that run more efficiently. Just as sports teams are coached differently depending on the relative strengths of the players, so too programs for a computer must consider the strengths of the component parts. In order to write efficient programs, it is important to understand the architecture that the program is to run on: Figure 1.1 shows one example of the complex structure of a modern vector supercomputer. This section, examines link between system performance and the various hardware components.

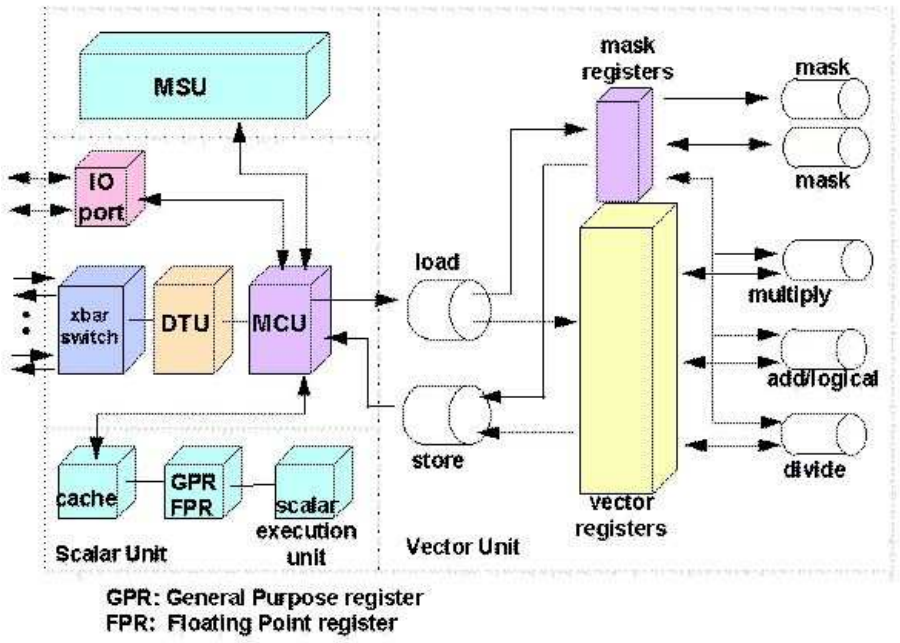


Figure 1.1: complex architecture of a vector supercomputer.

1.2.1 Central processing unit

The Central Processing Unit (CPU) is the heart of any computer system. It is the only component that can create or modify information. Most home computers only have a single CPU, but it may contain two or more "cores" High performance systems have two or more CPU's which have two or more "cores". However, the CPU's in all systems, operate in a similar fashion.

In one operation cycle, a simple view of the CPU is that it performs the following:

1. it fetches an instruction from memory,
2. decodes this instruction returned from memory and,
3. it executes the instruction.

The last of these steps may require further memory requests. For example, the decoded instruction from memory may instruct the processor to fetch two values from memory and subtract them and store the result in a new memory location. At the end of the operation cycle, the process is repeated with the CPU fetching the next instruction.

However, the following quote shows the situation is always much more complicated.

> The Pentium Pro processor is three-way super-scalar, permitting it to execute up to three instructions per clock cycle. It also introduced the concept of dynamic execution (micro-data flow analysis, out-of-order execution, superior branch prediction, and speculative execution) in a super-scalar implementation. Three instruction decode units worked in parallel to decode object code into smaller operations called micro-ops (micro-architecture op-codes). These micro-ops are fed into an instruction pool, and (when interdependencies permit) can be executed out of order by the five parallel execution units (two integer, two FPU and one memory interface unit). The Retirement Unit retires completed micro-ops in their original program order, taking account of any branches. *[Ano01, p.2–4]*

In most multi-processor or multi-core system, the Operating System (OS) does not split a program over multiple processors or cores. On these systems the OS simply assigns the program to one CPU or core, thus, freeing the others to run different programs. It is this load balancing that results in the extra performance of these systems.

The frequency of the CPU clock cycle is used to determine the speed at which the above instructions are executed. However, one cannot assume that the higher the frequency the faster the computer. For example, consider two computers systems, one has a AMD™ chip running at 1545 Mhz, the other has a Pentium™ chip running at 1800 Mhz. The actual speed of these two systems in benchmark testing is similar despite the difference in clock frequency. This difference can be accounted for by the difference in how the CPU implements instructions, and how it interacts with other components such as memory. The following discussion examines how the CPU interacts with other components.

1.2.2 Random access memory and cache

There are two types of Random Access Memory (RAM) in today's computers. They are referred to as *cache memory* and *Main memory*: as shown schematically in Figure 1.2, cache lies between the CPU and the main memory and aims to provide

a fast intermediate store. The performance of each is determined by the *access* and *cycle* time.

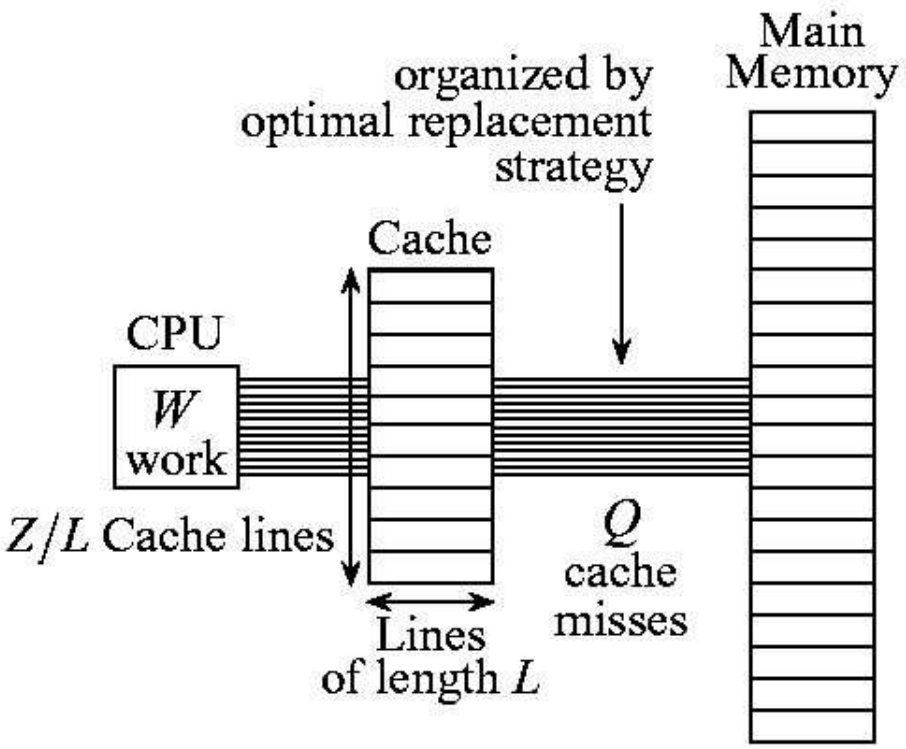


Figure 1.2: a cache lies between main memory and the CPU: actually most modern computers have multiple layers of cache!

> The power of the Pentium Pro processor was further enhanced by its caches: it had the same two on-chip 8-KByte 1st-Level caches as did the Pentium processor, and also had a 256-KByte 2nd-Level cache that was in the same package as, and closely coupled to, the processor, using a dedicated 64-bit backside (cache-bus) full clock speed bus.
>
> *[Ano01, p.2–4]*

Access time is the theoretical response time of the memory to read/write requests from the CPU. However, the true CPU/memory access time depends on several other factors, such as the time to forward the contents of memory to the CPU over the system bus. Access time is the main difference between cache and normal RAM. In particular, cache is normally high speed memory, and thus has a short access time. To further speed up access time, cache memory is normally located physically close to the CPU. Consequently, it takes less time to forward information between cache and the CPU.

The *Cycle time* is the time that must elapse between any two read/write requests being made to the memory. Thus there is a time overhead in waiting for memory to respond to repeated CPU requests. The actual time overhead involved depends on the physical specifications of the various types of memory used within the system.

The other important feature of memory, is that memory assignment is not random: memory is assigned in clusters (or blocks or lines). Figure 1.2 displays "lines" of cache that each store temporarily a block of consecutive locations from memory. This style of assignment has one major implication for high performance programming. Namely, that if your program is optimised to grab blocks of memory and store them in cache, it is possible to increase execution speeds substantially. Fortunately, we do not have to build this into our code, as most modern compilers, will do this optimisation for us. However, you should be aware that the efficiency of this optimisation depends both on the computer and the compiler. Thus, it is still important to be careful in how you assign and access memory in your code, and not rely on the compiler alone.

Example 1.1: To illustrate the improvement in performance given by cache, let us consider a system which has cache memory with access time of $t_c = 10\,\text{ns}$[1] and main memory which has a access time of $t_m = 120\,\text{ns}$. We assume that the CPU finds what it requires in cache 95% of the time. The effective memory access time (t_e) for the system is calculated as

$$\begin{aligned} t_e &= 0.95t_c + 0.05t_m \\ &= 0.95 \times 10 + 0.05 \times 120 \\ &= 15.5\,\text{ns}\ . \end{aligned} \tag{1.1}$$

Thus, the average access time for the system is 15.5 ns.

1.2.3 The system bus

The system bus is used to transfer information between the various hardware components. Small to mid-range home workstations have a single system bus, while many high performance systems have a more complex arrangement. The performance of the system bus is described by two parameters: the *transfer time* and the *bandwith* (or throughput).

The *transfer time* is the amount of time it takes the system bus to deliver data. In contrast, the *bandwidth* is a measure of how much information can be passed over the bus in one second. As such, the bandwidth, depends on the width of the bus (that is, the number of data lines that make up the bus) and how packets of information can be sent over the bus in one second. The higher the bandwith the more information that can be transferred simultaneously over the bus. The bandwith of the system bus is an important parameter in high performance computing in that determines how quickly information can be passed between Memory/cache and the CPU.

Over the last forty years, significant progress has been made in processor speed. However, processor development reached a point where the processor was no longer the limiting performance factor. The time to pass information over the system bus meant that the processor was idle much of the time, waiting to get information from memory. The ideal situation is for the CPU, cache and system bus to be balanced in such a way that the processor is fully utilised. There is no point in exceptional processor speed, if the processor cannot be supplied with information fast enough.

[1] The abbreviation 'ns' denotes nanoseconds

1.2.4 Input/Output

Tape/Disk and other storage devices

Real world computer applications require data to be read/written from Tape/Disk or other external storage devices. The performance of these devices depend heavily on the device in question and the system bus. As with the system bus, the performance of these devices is measured in terms of the *bandwidth* of the device (that is, the volume of data per unit time that can be transferred between the device and memory). Consequently, if large amounts of data are to be read from or written to such devices, it is important to choose devices with appropriate *bandwidth* and storage capacity to handle such data transfers. However, we will not explore these issues in this course.

Graphics output

At the human/computer interface most users are concerned about the speed of the graphical output. A high performance games machine requires a high performance graphics card. Computer modellers, also require a high performance graphics card to run 3D and real time computer models. To understand the advantage that high end graphics cards provide, one must look at how the various types of graphics devices work.

Some home computers use the main system CPU and RAM to do the calculations required to display the various video images. The advantage from the manufacturers point of view is that it cuts down costs. However, from the users point of view, it has the disadavantage that the main CPU has to do all the calculations to update the screen, as well as the program. In addition, the use of the main RAM for video storage, reduces the amount of system RAM that is available for program storage. The nett result of this is to slow the system down.

High performance computer systems improve graphics performance in two ways. The first of these is to add a dedicated graphics CPU (often called a GPU—Graphics Processing Unit) to the system. This frees the main CPU from having to do the extensive graphical calculations required in games and 3D modelling, and increases the amount of information that can be processed by the system. However, it is still possible that by sharing the system RAM, the graphics and system performance are not optimised. In particular, it is possible that the graphics performance is hampered by the speed of the system bus and that overall system speed is degraded by sharing the system RAM between the graphics and system CPU's. Thus, the optimum solution to this problem is to add dedicated graphics memory, in addition to a dedicated graphics CPU. This enables system RAM to be solely used by the system CPU. It also enables the graphics CPU dedicated access to memory, without placing additional overhead on the system, thereby freeing the system bus for other data traffic.

Exercise 1.2: The following specifications about the Silicon Graphics™ Fuel workstation, was obtained from the Silicon Graphics™. It lists important details about the Fuel workstation. Extract from this overview the following information: processor, speed of the processor, the amount of cache, the memory bandwidth and bus speed.

```
Technical Info

Leveraging industry-leading technology from Silicon Graphics
Fuel features the latest MIPS R16000A(TM) processor, the
unparalleled VPro(TM) 3D graphics system for IRIX, and a
high-bandwidth design based on the SGI 3000 family systems
architecture. Due to its high-performance architecture, this
workstation is perfectly designed to execute demanding applications
for creative and technical users, while the 48-bit RGBA provides the
highest level of precision available on any desktop system today. The
Silicon Graphics Fuel visual workstation is complementary to the
Silicon Graphics Tezro(TM) workstation and is binary compatible with
current IRIX applications.

The Silicon Graphics Fuel visual workstation is powered by the new
64-bit MIPS R16000A processor. With out-of-order execution, large
flexible caches, and super-scalar design, R16000A brings top
performance to real-world codes:
    * 800 MHz
    * Four-way super-scalar, 64-bit architecture
    * Out-of-order instruction execution
    * Five separate execution units
    * MIPS 4 instruction set
    * 32KB two-way set-associative on-chip instruction cache
    * 4MB fast secondary cache

Key Architecture Features
    * 3.2GB/sec main memory peak bandwidth
    * 1.6GB/sec system-to-graphics interconnect
    * VPro graphics
    * MIPS RISC processing, 64K primary cache, 4MB secondary cache
    * Optimized 200 MHz front-side bus
    * 32- or 64-bit binaries
    * Priority I/O
    * Integrated PCI
```

The following specifications on the Silicon Graphics™ Octane2 was also obtained from Silicon Graphics™. Once again extract from this page information about the: processor, speed of the processor, the amount of cache, the memory bandwidth and bus speed.

```
Technical Info

The unsurpassed crossbar switch design of the Silicon Graphics
Octane2 (TM) system provides excellent levels of interactivity with
critical data transfers, such as loading a 3D model from memory to the
screen. The 1.0GB per second peak main memory bandwidth and 1.6GB per
second peak bandwidth between subsystems ensure that you experience
smooth, fluid operations while completing even the most complex
tasks. The Octane2 workstation's 8GB memory capacity gives you
increased power to handle bigger data sets for more accurate analysis
in less time. Configured with single or dual MIPS processors, Octane2
delivers the power to quickly complete one task or simultaneously
tackle multiple tasks such as design and analysis or motion modeling
and behavior scripting. You can perform more tasks on more data than
ever before at the desktop level. The exceptional system
responsiveness means that you can focus completely on the problem you
are solving, while working intuitively to get the job done better and
faster.

Key Architecture Features
    * 1.0GB/sec main memory peak bandwidth
    * 1.6GB/sec peak
    * 64K primary cache, 2MB secondary cache
    * 32 or 64-bit binaries
    * Symmetric multiprocessing
    * Priority I/O
    * Optimized front side bus

Octane2 I/O
    * Two full- and one half-size optional industry-standard PCI slots
      for 32- and 64-bit wide PCI devices
    * Three 3.5-inch Ultra SCSI drive bays
    * Four XIO slots for graphics, networking, and storage cards

XIO Option Cards
All have the same form factor and cover one or two slots.
    * Storage XIO Option Cards
          o Fibre Channel--two ports (one slot)
          o UltraSCSI--four ports (one slot)
    * Networking XIO Option Cards:
          o Fast Ethernet--four ports (one slot)

Processors

The Silicon Graphics Octane2 visual workstation is powered by the
64-bit MIPS R14000A(TM) processor.
    * R14000A--The New Computational Leader, R14000A brings top
      performance to real-world codes.
    * R14000A CPU:
          o 600 MHz
          o Four-way super-scalar, 64-bit architecture
          o Out-of-order instruction execution
          o Five separate execution units
          o MIPS 4 instruction set
          o 32KB two-way set-associative on-chip instruction cache
          o 2MB fast secondary cache
```

Use the information you have extracted to compare and contrast the two workstations in turns of the likely performance of each workstation. Justify your reasoning.

1.3 Different high performance architectures

The most common way to classify high performance architectures is to use *Flynn's taxonomy*. This taxonomy classifies architectures by how information flows into the processing unit within the CPU (processor). It divides the flow of information up into two streams. These streams are called the *data stream* and the *instruction stream*. The *data stream* carries information from RAM and cache (DATA memory) to the processor, while the *instruction stream* carries instructions from the instruction memory (that is the area of RAM and cache that stores the actual program instructions) that to processor. Using this classification, computer architectures are classified according to how many of these streams are used.

1.3.1 Single instruction stream, Single data stream (SISD)

Single instruction, single data stream computers consist of a single processor that receives a single instruction stream that operates on a single data stream (Figure 1.3). That is, at each step during program execution one instruction is processed by the processor on the data obtained from memory. For example, the instruction may tell the processor to add the data and put it back in memory.

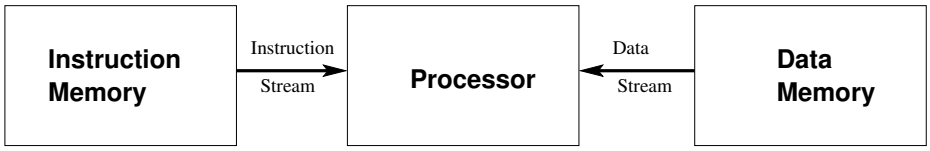


Figure 1.3: Schematic of a SISD computer

Most home computers fall into this category. High performance vector processors also fall into this category. Modern super-scalar processors achieve their high performance by passing successive elements of vectors to separate pieces of hardware which handle different parts of complex operations (see Example 1.3). We concentrate on SISD computation in this course.

Example 1.3: To add two numbers, such as 3.4×2^5 and 1.6×2^4, the numbers must have the same exponent. To do this the processor must alter the mantissa and the exponent of one number until it makes the other. In the above example 3.4×2^5 is changed to 6.8×2^4. The computer then adds 6.8×2^4 and 1.6×2^4 to get 8.4×2^4. A super-scalar processor is constructed so that the single data stream is feed into the processor at a high rate, so that one part of the processor adds the mantissas and another part adjusts the exponents.

1.3.2 Multiple instruction stream, Single data stream (MISD)

In MISD computers, there are N processors, each with its own instruction memory (Figure 1.4). These processors access a common data area. Thus, at each step, at a

block of data is operated upon by all processors simultaneously. This class of computers lend itself to applications requiring data to be subject to several operations at the same time (see Example 1.4).

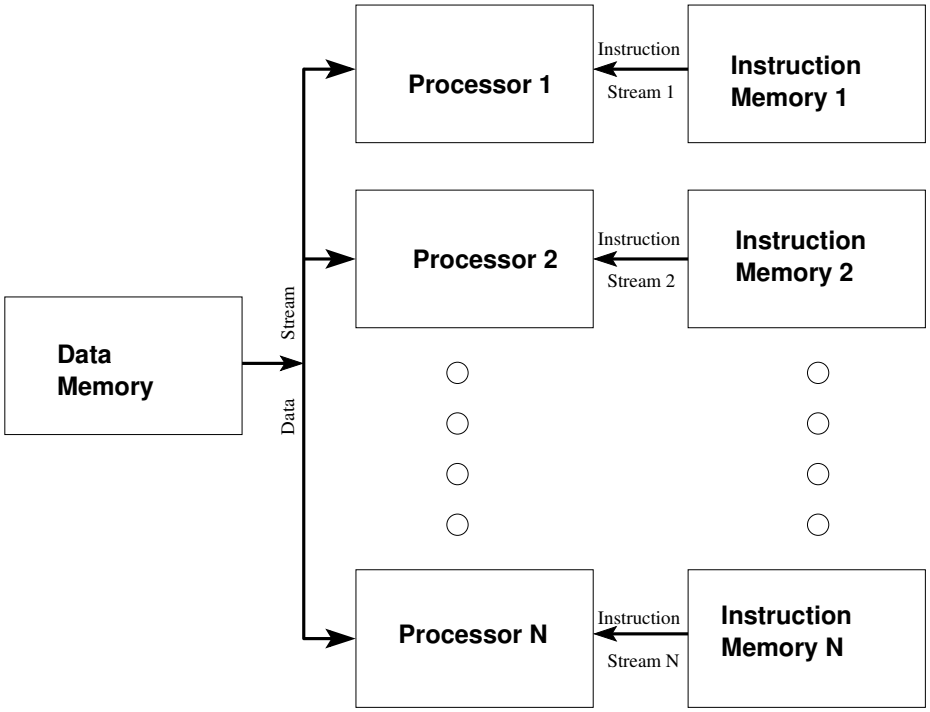


Figure 1.4: Schematic of a MISD computer

Example 1.4: One of the common computer applications is to classify certain objects into classes. For example, a robot may need to distinguish between circle, squares and triangles. Typically this classification is done by subjecting it to a number of different tests. For our robot this can be done by counting vertices. Such comparisons can be done quickly on MISD computer. Each processor is associated with a class and can recognise members of that class through a computational test. Data on the object is sent simultaneously to all processors, where it is tested in parallel. The processor than reports the success of its test.

The above example shows that MISD computers can be useful in many applications. However, the type of calculations that can be carried out efficiently are rather specialised. Parallel architectures that are more flexible and hence suitable to a wider range of problems are discussed in the following two sections.

1.3.3 Single instruction stream, Multiple data stream (SIMD)

In SIMD (Figure 1.5) each of N processors has its own data memory where it can store data. All these processors operate under instructions from a single instruction bank. An advantage of this style of architecture is the savings in logic that needs to be built into each chip. Anywhere from 20% to 50% of the logic on a typical processor is devoted to fetching, decoding and scheduling instructions (this area of the chip is called the *Control Unit*). The remainder is used for storage, and logic to

implement data processing. In an SIMD computer, only one of these control units is required, so more areas of each chip can be dedicated to storage and data processing.

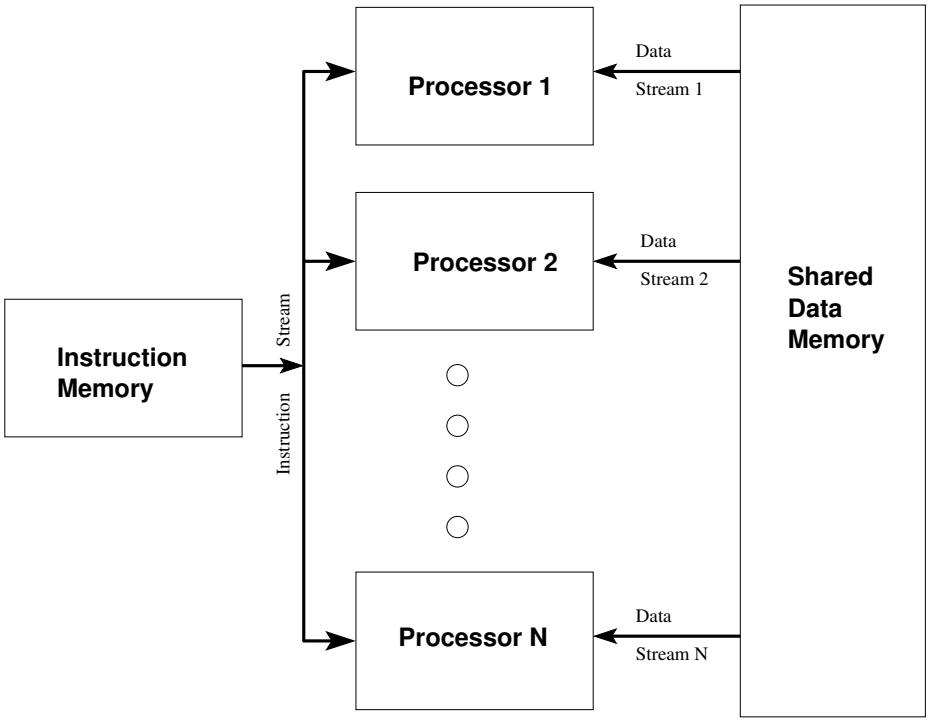


Figure 1.5: Schematic of a SIMD computer

Example 1.5: Vector processing on a SIMD machine is performed by distributing elements of the vectors across all processors. As example, consider adding two vectors a and b. Processor 1 would get the first elements of a and b, store the result back in memory. Processor 2 would get the second elements of a and b, add them and store them back in memory. The other processors would do the same thing until all elements have been added. As long as the number of processors is larger than the length of the vectors, the processing does not depend on the length of the vectors. On SISD vector processors, the time taken is a linear function of the length of the vectors.

For most problems that we want to solve on SIMD systems, it is desirable that the processors are able to communicate amongst themselves, in order to exchange data and intermediate calculation results. This can be achieved in either by using *shared memory* or an *interconnected network*. How this is done gives rise to various different topographies of parallel computers.

Quantum computing Within the next decade, further miniaturization of computer chips will cease as fundamental physical limits are reached. Quantum computing offers the distant promise of amazing further increases in computing power. The multiple entangled states of a quantum computer's processor effectively act as a vast reservoir of data which is all acted upon simultaneously by quantum processing logic. Quantum computers may be viewed as a type of SIMD computer.

1.3.4 Multiple instruction stream, Multiple data stream (MIMD)

MIMD machines are the most diverse of Flynn's four classifications, and include such things as local networks of workstations and units specifically designed for parallel processing. In MIMD machines we have N processors, with N data streams and N instruction streams (Figure 1.6). Each processor under its own set of instructions, that arrive via a separate instruction stream. Thus the processors are potentially executing different programs on different data while solving different subproblems of a specific problem. This means that the processors typically operate asynchronously.

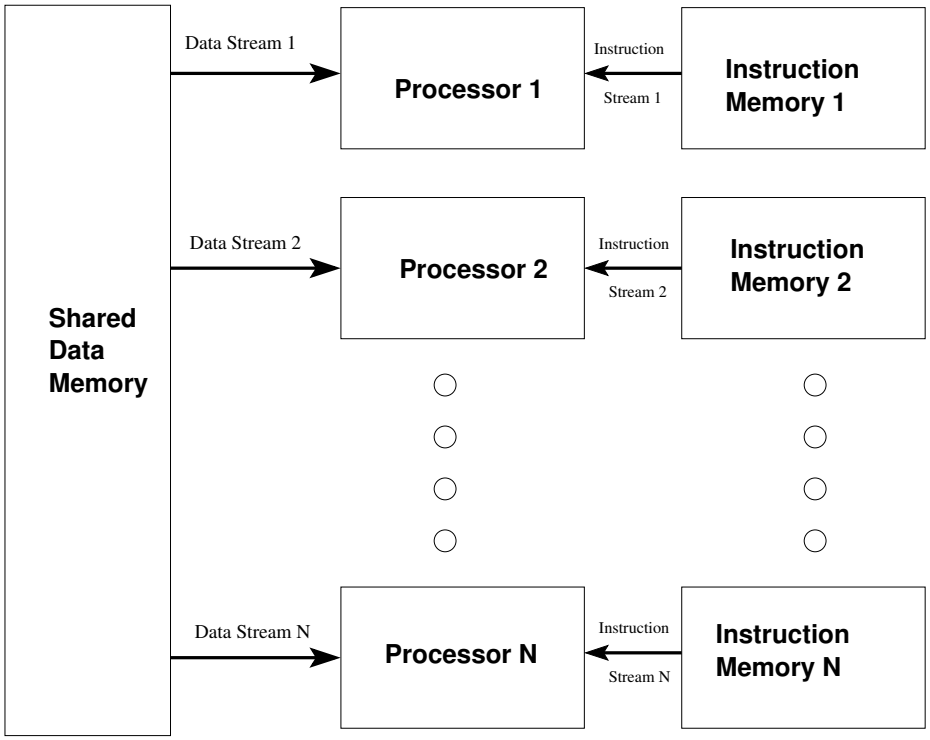


Figure 1.6: Schematic of a MIMD computer

With "dual core" processors now becoming standard, most new home computers are MIMD computers with $N = 2$ processors.

As with SIMD computers, it is desirable that the processors can communicate amongst themselves. Such communication is done using *shared memory* or *interconnected network*.

1.4 Summary

In this module, some general issues in the basis of High Performance Computing have been discussed. The next module, uses MATLAB to look at the vector and super-scalar programming issues in detail. In particular, it will focus on how making use of the vector mode in MATLAB (or on vector or super-scalar processing hardware) improves performance remarkably.

1.4.1 Glossary

Computer architecture The organizational structure of a computer system, including hardware and software. In particular, it covers how the various different parts of a computer are designed to interact.

Workstations a desktop computer.

Real time the actual wall clock time.

Clock cycle A clock is used in computers to coordinate the actions of several circuits. The clock cycle is the interval at which a signal is generated. The clock cycle is normally measured in nano seconds (ns) (that is, 10^{-9}s). The number of clock cycles per instruction (CPI) determine the speed of the computer.

Clock frequency As should expected the clock frequency relates to the clock cycle. It is the number of clock cycles executed in 1 s. That is,

$$\text{clock frequency} = \frac{1}{\text{clock cycle time}}$$

It is normal expressed in Hertz (Hz). These days the clock frequency for AMD and Pentium chips is normally stated in Giga (Billion) Hertz (GHz).

System bus the network of wires on the motherboard that carry between the CPU and other devices within the computer.

Chapter 2 Computer Computation Basics

Chapter contents

2.1 Introduction

This module begins with a review of some important concepts associated with computer representation of floating point numbers, and the inherent error in computer calculations. It then follows with a discussion of an important result in calculus "Taylor's Series", which is used extensively to construct many useful numerical methods. It finishes with a discussion of *vector* and *scalar* computations—that is, the ability of some computer architectures to run multiple computations simultaneously.

2.2 Binary Numbers

All computers deal with real numbers using the binary system, as distinct from the decimal system that humans prefer. The binary system uses 2 as the base in the same way that the decimal system uses 10.

When a real number such as 427.325 is written out in detail, we have

$$27.325 = 2 \times 10^1 + 7 \times 10^0 + 3 \times 10^{-1} + 2 \times 10^{-2} + 5 \times 10^{-3}$$

If we admit the possibility of having an infinite number of digits standing to the right of the decimal point, then *any* real number can be represented. Thus $-\pi$ is

$$-\pi = -3.14159265358979323846264338327 6\ldots$$

The last 6 written stands for 6×10^{-30}

In the binary system, only the digits 0 and 1 are used. A typical number in the binary system can also be written in detail, for example

$$\begin{aligned} 11011.01010011 \quad = \quad & 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ & + 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} \\ & + 0 \times 2^{-5} + 0 \times 2^{-6} + 1 \times 2^{-7} + 1 \times 2^{-8} \end{aligned}$$

This is the decimal number 27.325 in binary notation.

Since computers communicate with humans using the *decimal* system but work internally in the *binary* system, the computer must convert between the two at input/output time. Unfortunately, the binary system cannot represent fractions as easily as the decimal system—most simple decimal fractions can only be represented exactly in binary as an infinite series. For example, the simple decimal fraction 1/10 in binary notation becomes

$$0.0001100110011001100110011\ldots \tag{2.1}$$

As computers can only represent real numbers with a fixed number of digits there is a restriction on the precision with which real numbers can be represented.

2.3 Normalised Scientific Notation

In the decimal system, any real number can be expressed in *normalised scientific notation*. This means that the decimal point is shifted and appropriate powers of 10 are supplied so that only one non-zero digit is to the left of the decimal point. For example—

$$\begin{aligned} 5198.6013 &= 5.1986013 \times 10^3 \\ -0.0009274 &= -9.274 \times 10^{-4} \end{aligned}$$

in general, a non-zero real number x can be represented in the form

$$x = \pm r \times 10^n, \tag{2.2}$$

where r is a number in the range $1 \leq r < 10$ and n is an integer. The number r is called the **mantissa** or **significand** and the integer n is called the **exponent**.

In exactly the same way, we can use normalised scientific notation to represent a non-zero *binary* number—

$$x = \pm q \times 2^m, \tag{2.3}$$

where the **mantissa** q is $1 \leq q < 2$ and the **exponent** m is an integer.

In a computer both the mantissa and exponent will be represented as binary numbers.

2.4 IEEE Floating-Point Representation

Within a typical computer, numbers are represented using normalised scientific notation—but with length restrictions placed on the mantissa q and the exponent m.

The IEEE[1] floating point standard is the most common representation for real numbers in computers today.

The floating-point standard defines how a number is to be stored. That is, it defines how many bits of storage are to be used for the mantissa and the exponent. Table 2.1 lists the IEEE floating-point standard for single precision and double precision. Single precision consists of representing a floating-point number in one 32-bit word and double precision represents a floating-point number in two 32-bit words[2]

Table 2.1: IEEE layout for single (32-bit) and double (64-bit) precision floating-point values. The number of bits for each field are shown with the bit ranges in square brackets

	Sign	Exponent	Mantissa	Bias
Single Precision	1 [31]	8 [30–23]	23 [22–0]	127
Double Precision	1 [63]	11 [62–52]	52 [51–0]	1023

The Sign Bit

0 denotes a positive number; 1 denotes a negative number. Changing the value of this bit changes the sign of the number.

The Exponent

The exponent field has to represent both positive and negative exponents. To do this, a *bias* is added to the stored exponent in order to get the true exponent. For IEEE single-precision floats, this value is 127. Thus, a stored exponent of 126 means that the actual exponent is -1. A stored value of 200 means an actual exponent of 73.

For reasons discussed below, actual exponents of -127 (all 0s) and +128 (all 1s) are reserved for special numbers.

The Mantissa

As the floating-point number is stored in normalised form, the first digit of the mantissa must be non-zero. As we are dealing with binary digits this means that

[1] Institute of Electrical and Electronics Engineers—the leading developer of computer hardware standards.

[2] Be aware that "64-bit architecture" still uses a 32-bit word length. The 64-bits refers not to the word length but to the length of a memory address. This means that "64-bit architecture" increases the amount of memory that can be used from ~ $4GB$ to ~ $16EB$.

the mantissa's first digit must always be 1. Thus, we can always assume a leading digit of 1, and don't need to store it explicitly. As a result, the single precision mantissa is effectively represented by 24 bits, with only 23 stored.

Special Values

Zero Due to the assumption of a leading 1 in the mantissa—zero is not directly representable. We would need to specify a true zero mantissa to yield a value of zero. So zero is a special value denoted with an exponent field of zero and a fraction field of zero.

Note the definition of zero means that -0 and +0 are distinct values, though they both compare as equal.

Infinity The values *+infinity* (+∞) and *-infinity* (−∞) are denoted with an exponent of all 1s and a fraction of all 0s. The sign bit distinguishes between the negative and positive infinity.

Being able to denote infinity as a specific value is useful because it allows operations to continue past overflow situations (see below).

Not A Number The value NaN (*Not a Number*) is used to represent a value that does not represent a real number. NaN's are represented by a bit pattern with an exponent of all 1s and a non-zero fraction.

Special Operations Operations on special numbers are well-defined by IEEE. In the simplest case, any operation with a NaN yields a NaN result. Other operations are defined in Table 2.2

Table 2.2: Operations with special numbers

Operation	Result
$n \div \pm$ inf	0
$\pm$ inf $\times \pm$ inf	$\pm$ inf
$\pm$nonzero $\div$ 0	$\pm$ inf
inf + inf	inf
$\pm 0 \div \pm 0$	NaN
inf − inf	NaN
$\pm$ inf $\div \pm$ inf	NaN
$\pm$ inf $\times 0$	NaN
Any with NaN	NaN

Exercise 2.1: Using Table 2.2 experiment in MATLAB with the special values NAN and INF.

When MATLAB attempts to plot a vector that contains NAN's, they are silently ignored with breaks in the plot. This can be used to split a vector into multiple plots. Experiment with this feature.

Converting Integer to a Floating-Point Number

Converting an integer to a single-precision float may entail a loss of precision—as the 32-bit integer may have to be reduced to fit into the 24-bits of the mantissa. To do this the integer is truncated. For example—

$$\begin{aligned} 4039944719 &= 11100001100110010101010000001111 \\ &\approx 1.11100001100110010101010 \times 2^{31} \\ &= 11110000110011001010101000000000 \\ &= 4039944704 \end{aligned}$$

The last 8 bits of the integer are lost when converted to floating-point representation which means the floating-point number could be smaller by a maximum of 255 (in this case only the last four bits are non zero so the floating-point number is smaller by 15).

Range of Floating-Point Numbers

The largest and smallest positive numbers that can be represented by IEEE floating-point representation are given in Table 2.3. Since the sign of floating-point numbers is given by the special leading bit, the range for negative numbers is given by the negation of the numbers in the table.

Table 2.3: The largest and smallest positive numbers in the IEEE standard

	Maximum Binary	**Maximum Decimal**	**Minimum Binary**	**Minimum Decimal**
Single Precision	$(2 - 2^{-23}) \times 2^{127}$	$\sim 3.4028 \times 10^{38}$	2^{-126}	$\sim 1.1754 \times 10^{-38}$
Double Precision	$(2 - 2^{-52}) \times 2^{1023}$	$\sim 1.7976 \times 10^{+308}$	2^{-1022}	$\sim 2.2250 \times 10^{-308}$

There are five distinct numerical ranges that single-precision floating-point numbers cannot represent:

1. Negative numbers less than $\sim -3.4028 \times 10^{38}$ and is called *Negative Overflow*. Any attempt to store a number in this range will result in a value of – inf.
2. Negative numbers greater than $\sim -1.1754 \times 10^{-38}$ and is called *Negative Underflow*. Any attempt to store a number in this range will result in a value of −0.
3. Zero
4. Positive numbers less than $\sim 1.1754 \times 10^{-38}$ and is called *Positive Underflow*. Any attempt to store a number in this range will result in a value of +0.
5. Positive numbers greater than $\sim 3.4028 \times 10^{38}$ and is called *Positive Overflow*. Any attempt to store a number in this range will result in a value of + inf.

Overflow means that values have grown too large for the representation. Underflow is a less serious problem because it denotes a loss of precision, which is guaranteed to be closely approximated by zero.

Precision of Floating-Point Numbers

The restriction that the **mantissa** of the floating-point number is of fixed length means that computer numbers have limited precision. The least significant bit in the single-precision mantissa is 2^{-23} which is .00000012. This means that numbers expressed in normalised scientific notation with a decimal mantissa of more than 7 digits will be *approximated* when converted to binary. In double-precision the least significant bit of the mantissa is 2^{-52} which is .00000000000000022. This means that mantissae with more than 15 decimal digits are approximated.

Also, some *simple* decimal numbers such as 1/10 are not representable by a binary number of limited length (see Equation 2.1 above) so are *always* approximated on a computer.

Floating-point numbers that can be exactly represented in a computer are distributed rather unevenly, more of them being concentrated near zero. There are only a finite number of floating-point numbers that can be represented in a computer exactly, and between adjacent powers of two there are always the same number of machine numbers (because of the fixed length mantissa). Since gaps between powers of 2 are smaller near zero and larger away from zero this produces a non-uniform distribution of floating-point numbers, with a high density near the origin.

2.5 Arithmetic Round off Error

As well as the *inexact representation* of numbers in computers the arithmetic operations performed are also inexact, even if the operands happen to be exactly represented. For example, to add or subtract two numbers the following operations are performed:

1. Select the smaller number (in magnitude) of the two. Right shift (divide by two) the mantissa of the smaller number, simultaneously increasing its exponent.
2. When the exponent of the two numbers are identical subtract or add the mantissae.
3. Normalise the answer if necessary (left shift the mantissa).

In the process of "right shifting" the low-order (or least significant) bits of the smaller operand may be lost. If the two operands differ too greatly in magnitude, then the smaller operand will be right shifted to zero!

The smallest (in magnitude) floating-point number which, when added to the floating-point number 1.0, produces a floating-point result different from 1.0 is termed the *machine accuracy* ϵ_m. This definition of *machine accuracy* (there are others) just represents spacing between floating-point numbers that can be represented near 1.0. For IEEE single-precision floating-point numbers the machine accuracy is

$\epsilon_m = 2^{-23} = 1.1921 \times 10^{-07}$ and for double-precision it is $\epsilon_m = 2^{-52} = 2.2204 \times 10^{-16}$. Which unsurprisingly, is the value of the least significant bit in the mantissa.

It is important to understand that the machine accuracy is given by the number of bits in the *mantissa* and that the smallest or largest floating-point number that can be represented on the machine is given by the number of bits in the *exponent*.

Round off errors accumulate with increasing amounts of calculation. If you perform N calculations and the round off errors accumulate preferentially in one direction then the total error will be of order $N\epsilon_m$—this is the worst case scenario. If, as is more likely round off errors occur randomly—positive and negative—then the total round off error will be of order $\sqrt{N}\epsilon_m$. (This result is calculated assuming a *random* walk—see Module 5 on Monte Carlo methods.)

Sometimes however calculations can conspire to vastly increase the round off of single operations. Generally these can be traced to the subtraction of two very nearly equal numbers, giving a result whose only significant bits are those few low-order bits in which the operands differed. This occurs far more frequently than you may suppose—and requires vigilance even when coding simple mathematical expressions. For example, consider the seventh order polynomial:

$$y = x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1. \tag{2.4}$$

Using the MATLAB code below will plot the polynomial in the range $0.998 \leq x \leq 1.012$.

```
x = 0.988:.0001:1.012;
y = x.^7 - 7*x.^6 + 21*x.^5 - 35*x.^4 + ...
  35*x.^3 - 21*x.^2 + 7*x - 1;
plot( x, y)
```

Figure 2.1 is the result—a smooth polynomial is not the result—round off is dominating this simple calculation. From the plot, it can be seen that $y \sim 10^{-14}$—these small values for y are being calculated by taking the sums and differences of numbers as large as ~ 35—15 orders of magnitude larger than the final result.

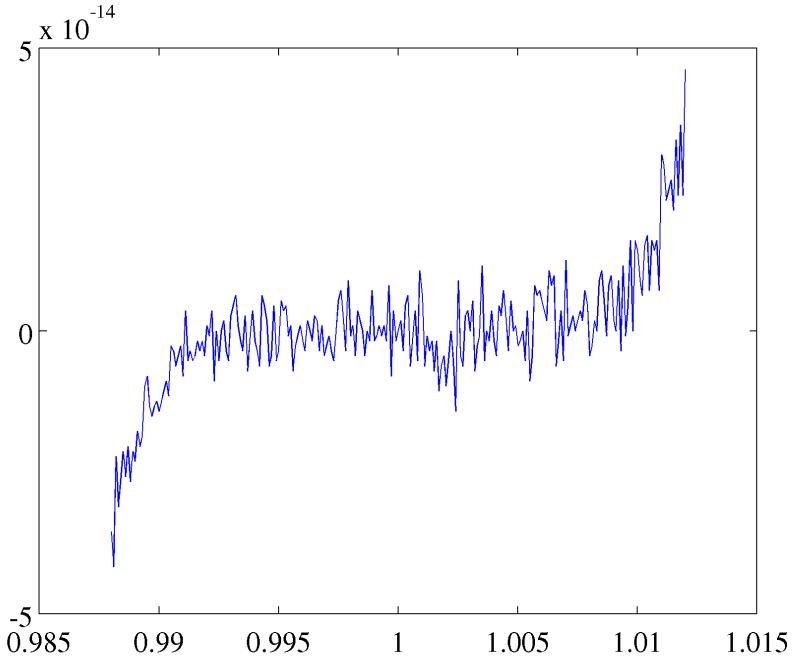


Figure 2.1: Plot of a seventh order polynomial using Equation 2.4

Eliminating rounding problems is not always easy or possible. Rearranging the equation may reduce or even eliminate the problem. In the case of the polynomial of Equation 2.4, near $x = 1$ the polynomial is close to zero, no matter how the terms in the sum are rearranged the final operation will involve subtraction of nearly equal quantities. The round off error here is small (note the scale in Figure 2.1), so the problem may not be serious—but that critically depends on how the solution is to be used. For example, using a computer to find the roots of Equation 2.1, it would be useless to ask for high precision in the value of the root, because round off error "noise" makes it appear that almost any value in the interval [0.99,1.01] could be the root. A user should be aware of the potential hazard of round off error, to avoid asking for "impossible precision".

From Figure 2.1 it is clear that a root to the polynomial exists around 1—it is also clear that the derivative is also tending to zero around 1—this means that a possible multiple factor exists at $x = 1$. First step in plotting a smooth version of this polynomial is to try and factorise it around 1. Looking for factors around 1 will soon show that Equation 2.4 can be written as:

$$y = (x - 1)^7. \tag{2.5}$$

When Equation 2.5 is plotted with the script code below.

```
x = 0.988:.0001:1.012;
y = (x-1).^7;
plot( x, y)
```

Figure 2.2 is the result.

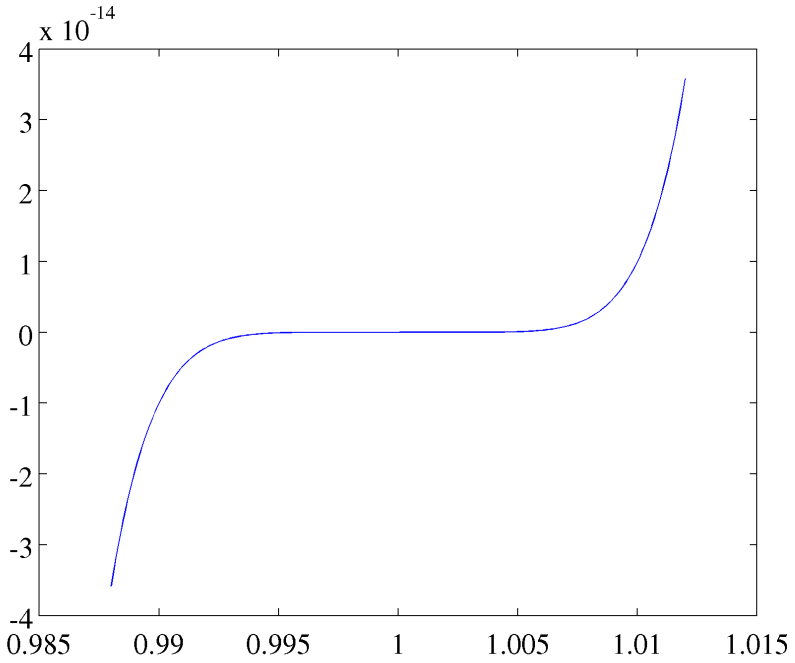


Figure 2.2: Plot of a seventh order polynomial using Equation 2.5

Example 2.2: The solution of the quadratic equation

$$ax^2 + bx + c = 0,$$

where the coefficients a, b and c are real can be written in two ways—

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

and

$$x = \frac{2c}{-b \pm \sqrt{b^2 - 4ac}}.$$

Using *either* of the solutions above to get the two roots is very dangerous. If either a or c or both are small, then one of the roots will involve the subtraction of b from a nearly equal quantity (the discriminant)—and will be very inaccurate. The correct way to calculate the roots is to avoid the subtraction and that can be done by calculating—

$$q = \frac{1}{2}\left[b + \text{sgn}(b)\sqrt{b^2 - 4ac}\right],$$

where sgn is the sign function.

Then the two roots are

$$x_1 = \frac{q}{a} \qquad \text{and} \qquad x_2 = \frac{c}{q}.$$

2.6 Taylor's Series

An important result for many methods in numerical computing is Taylor's series. Taylor's series tells us that any complex function that is differentiable can be represented by an infinite polynomial series. In many numerical methods it is found that the Taylor's series is far better to use than the original function.

There are a number of ways to derive Taylor's series—the most straight forward is to use integration by parts. Assume that the function f has continuous derivatives on $[a, b]$ then for any points x and $x + h$ in $[a, b]$ we can write—

$$f(x + h) - f(x) = \int_x^{x+h} f'(t)\, dt, \tag{2.6}$$

where $f'(t)$ is the derivative of f with respect to its argument t.

Recalling the formula for integration by parts—

$$\int udv = uv - \int vdu, \tag{2.7}$$

we can apply it to Equation 2.6.

Choosing

$$u = f'(t), \quad \text{and} \quad dv = dt,$$

leads to

$$du = f"(t)\, dt, \quad \text{and} \quad v = t - x - h,$$

where we have introduced $x + h$ as the constant of integration. Equation 2.7 now becomes

$$\int_x^{x+h} f'(t)\, dt = \Big[(t - x - h)f'(t)\Big]_{t=x}^{t=x+h} + \int_x^{x+h} f"(t)(x + h - t)\, dt. \tag{2.8}$$

Noting that the left-hand side is $f(x + h) - f(x)$ and that the first term on the right-hand side vanishes at the upper limit, we have

$$f(x + h) = f(x) + f'(x)h + \int_x^{x+h} f"(t)(x + h - t)\, dt. \tag{2.9}$$

We can again integrate by parts the integral of Equation 2.9. Choosing

$$u = f''(t), \quad \text{and} \quad dv = (x + h - t)dt,$$

leads to

$$du = f'''(t)\,dt, \quad \text{and} \quad v = -\frac{1}{2}(t - x - h)^2,$$

Equation 2.9 becomes

$$f(x + h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \int_x^{x+h} f'''(t)(x + h - t)^2\,dt. \tag{2.10}$$

Repeating the integration by parts produces the series—

$$\begin{aligned} f(x + h) &= f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \ldots + \frac{1}{n!}f^{(n)}h^n \\ &\quad + \int_x^{x+h} f^{(n+1)}(t)(x + h - t)^n\,dt. \end{aligned} \tag{2.11}$$

The expression for $f(x + h)$ given here is called Taylor's series with an integral remainder. The integral remainder gives us the error we make in approximating $f(x + h)$ when we truncate the series at the term in h^n.

The Taylor series with remainder can be written in a number of ways, some of which are more useful than the integral remainder form given above. For our purposes, the most useful form is that derived by J. L. Lagrange, resulting in the Taylor series with derivative remainder:

$$f(x + h) = \sum_{k=0}^{n} \frac{1}{k!}f^{(k)}(x)h^k + E_n(h), \tag{2.12}$$

$$E_n(h) = \frac{1}{(n + 1)!}f^{(n+1)}(\xi)h^{n+1}, \tag{2.13}$$

where the point ξ lies somewhere between x and $x + h$.

The Taylor series is an important result for numerical computing as it is used to develop many of the standard numerical methods.

The Taylor series tells us that in the vicinity of a given point any function can be approximated just by knowing its value and the value of its derivatives at that point. **The error of the approximation is a function of the number of terms in the series and how far away from the point were the series terms are calculated.**

2.6.1 Maclaurin Series

When the point, x, about which we form the Taylor series is $x = 0$, the expansion is called the Maclaurin series. The Maclaurin series for any function can be written as (were we have renamed h to x for convenience.):

$$f(x) = \sum_{k=0}^{n} \frac{1}{k!} f^{(k)}(0) x^k + E_n(x), \tag{2.14}$$

$$E_n(h) = \frac{1}{(n+1)!} f^{(n+1)}(\xi) x^{n+1}, \tag{2.15}$$

where when $n \longrightarrow \infty$ the error term $E_n(x) \longrightarrow 0$ the series is said to converge.

Example 2.3: Consider the Maclaurin expansion of the function $\sin(x)$. Starting with

$$\frac{d}{dx} \sin x = \cos x, \quad and \quad \frac{d}{dx} \cos x = -\sin x,$$

we have $f(0) = 0$, $f'(0) = 1$, $f"(0) = 0$, $f^{(3)}(0) = -1$, ...The series expansion of $\sin x$ about 0 is

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + E_n(x). \tag{2.16}$$

For $\sin x$ we know that the derivative lies in the range $[-1, 1]$, therefore the error term can be written

$$|E_n(x)| \leq \frac{|x|^{n+1}}{(n+1)!}.$$

Since the denominator on the right grows faster then numerator, as $n \longrightarrow \infty$ the error $|E_n| \longrightarrow 0$. Therefore the Taylor series expansion of $\sin x$ around 0 will converge.

Figure 2.3 shows the Maclaurin series for $\sin(x)$ around zero using 1, 2, 3, and 4 terms. Notice that the accuracy of the series improves with more terms. Remember that the terms of the series are calculated using information at zero only—yet with only four terms the accuracy is not bad all the way out to π.

The Taylor series provides a method of evaluating a function over an interval, but using only information about that function and its derivatives at one point. In addition, the function is written as a polynomial plus an error term, and when the Taylor series converges, the error term can be made as small as required—simply by taking a sufficient number of terms in the polynomial expansion. This is particularly useful because a lot is known about polynomial functions, and leads to the basis for any methods employed in numerical computing.

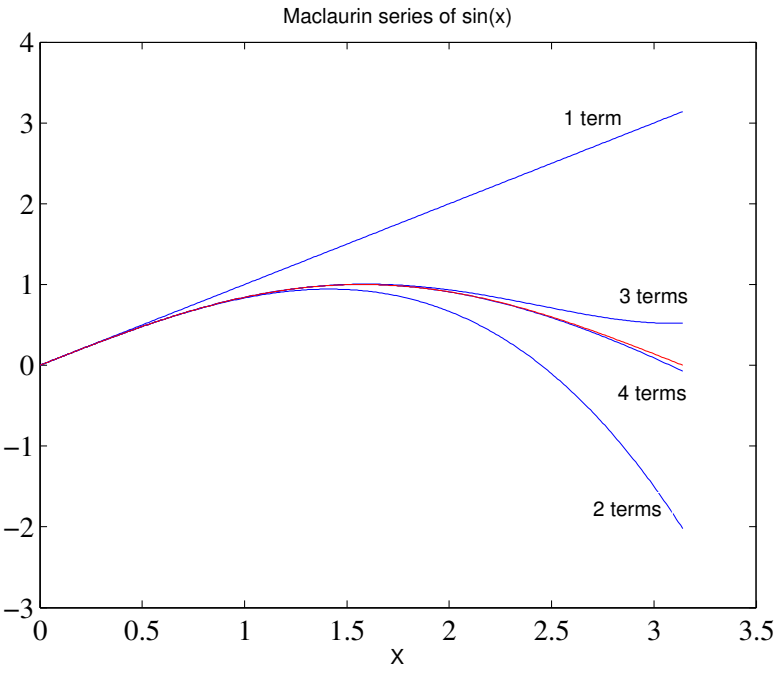


Figure 2.3: Comparison of a Taylor series expansion of $\sin(x)$ with increasing number of terms.

There are related techniques that represent a function not as a polynomial but as a series of basic functions. For example, a Fourier series represents a function as a series of trigonometric functions, sin and cos. It is particularly useful in the analysis of periodic systems, such as vibrating strings, electro-magnetism, etc.

Example 2.4: What is the value of the function $(\sin x)/x$ around zero?

Clearly both the denominator and the numerator approach zero as $x \longrightarrow 0$, but which one approaches zero faster? Is the function singular at zero?

We can use the Taylor series for $(\sin x)$ to find the Taylor series for the function $(\sin x)/x$—

$$\frac{\sin x}{x} = 1 - \frac{x^2}{3!} + \frac{x^4}{5!} - \frac{x^6}{7!} + \ldots \tag{2.17}$$

This series shows that as $x \longrightarrow 0$, $\sin x/x \longrightarrow 1$. So in the vicinity of $x = 0$ the function is well behaved.

To calculate this function using a computer requires that the Taylor series of the function be used in the vicinity of zero not the function itself. The number of terms in the series is dictated by the accuracy required of the result.

Exercise 2.5: Find the Taylor series of the polynomial $f(x) = x^4 - 5x^3 + 6x^2 - x + 3$

What is the Taylor series of any polynomial?

Exercise 2.6: Find the Taylor series for $f(x) = \cosh x$ about the point 0.

Exercise 2.7: Find the Taylor series of the function $f(x) = \sin x \cos x$ about the point 0 by multiplying the Taylor series of $\sin x$ about 0 to the Taylor series of $\cos x$ about 0 and gathering together all the terms of equal power in x.

This method is used extensively when calculating a Taylor series of complex functions.

Exercise 2.8: Find the Taylor series of the function $f(x) = 1/(1-x)$ about the point 0.

Using the Taylor series for $\sin x$ and the result above what is the first three terms in the Taylor series of $f(x) = 1/(1-\sin x)$ when x is small.

Note: The Taylor Series for $f(x) = 1/(1-x)$ will only converge when $x < 1$ this is called the Radius of Convergence. What happens if $x = 1$?

2.7 Truncation Errors

Round off error is characteristic of computer hardware. As a general rule there is not much a programmer can do about round off error—except to choose algorithms that do not magnify it unnecessarily. There is another source of error that is characteristic of the algorithm used and is independent of the hardware on which the program is executed. Most numerical algorithms compute "discrete" approximations to some desired "continuous" function. For example, a function may be evaluated using a Taylor series expansion with a "finite" number of terms. Or, an integral may be evaluated numerically by computing a function at a discrete set of points, rather at all points. In cases such as these there is always an adjustable parameter, the number of terms or points such that the "true" answer is obtained when the parameter goes to infinity. Any practical calculation is done with a finite number of points or terms, finite but sufficiently large for the accuracy required.

The discrepancy between the true answer and the answer obtained in a "practical" calculation is called the *truncation error*. Truncation error would persist even in a "hypothetical" computer that has infinite floating-point accuracy and no round off error. As a general rule—round off error is difficult to overcome. Truncation error, on the other hand, is entirely in the hands of the programmer. The simple solution of *truncation error* is to increase the number of terms in the series or just take more points—unfortunately there are problems with this approach—more terms or points will slow the calculation considerably—this is a major consideration when solutions may take days to produce. The inclusion of more points or terms may mean that round off errors start to dominate the solution! In fact it is not an exaggeration to say that the entire field of numerical analysis is the production of methods that minimise truncation error without increasing the number of calculations.

Example 2.9: Consider the Taylor series expansion of e^x about $x = 0$—

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots$$

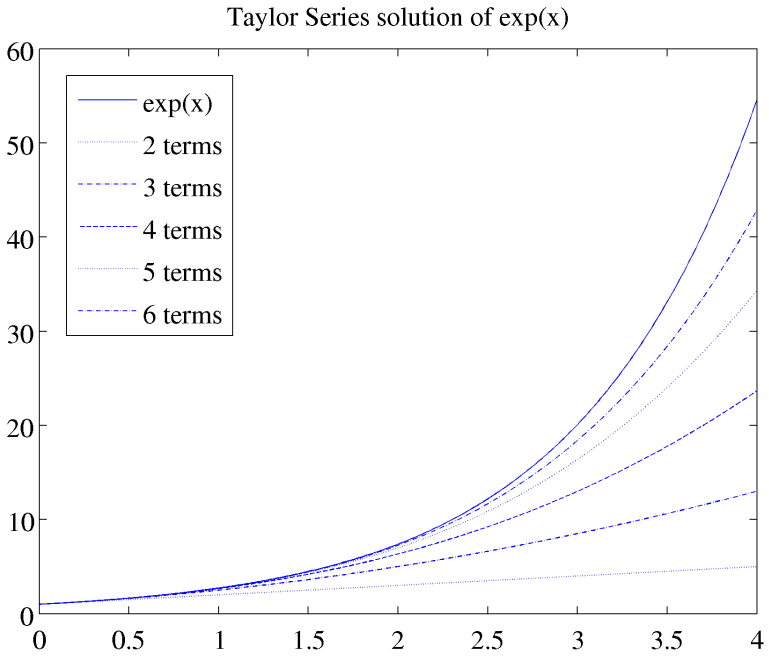


Figure 2.4: Comparison of a Taylor series expansion of e^x with increasing number of terms.

Figure 2.4 plots the solution using the computer's supplied function $\exp(x)$ (which will solve the function to the precision of the machine) and successive Taylor series approximations.

A number of things are apparent:

- For a given x increasing the number of terms in the Taylor series expansion and the error (which is proportional to x^n) decreases.
- The closer to the point of the expansion ($x = 0$ in this case) the fewer terms required in the expansion for a given error. That is, more terms are required for a given error the further you move away from the point of the expansion.

  This is not surprising as the point of expansion contains all the information you have about the function —the further you move away from that point the more information (more derivatives) required to maintain accuracy.

Exercise 2.10: For small values of x, the approximation $\sin x \sim x$ is often used. Using the Taylor's series for $\sin x$ estimate the error in using this approximation. For what range of values of x will this approximation give results correct to six decimal places?

Exercise 2.11: Derive the Taylor series around 0 for the function $f(x) = \ln(x + 1)$. Write this series in summation notation with a derivative remainder. Determine the smallest number of terms that must be taken in the series to yield $\ln 1.5$ with an error of less than 10^{-10}

2.8 Vector and super-scalar Computers

Seymour Cray was the first to incorporate a vector mode into computer processors. A vector mode means that the next computation starts before the previous one finishes. Modern vector supercomputers, based on SISD design, can have at least 20 computations running simultaneously. All of which are at different stages. There are major improvements in computation speed as a result of doing this. Home workstations cannot offer the same level of performance. Instead, the standard Intel and AMD processors have fewer computations running simultaneously. These processors are referred to as *super-scalars*, due to their vector-like mode of operation.

To take advantage of the *super-scalar* nature of these processes, we use programming techniques developed for vector computers. Such vector programming using MATLAB is covered here.

2.8.1 MATLAB as a vector computer

Rather than use a real supercomputer, requiring remote and inconvenient access, we explore many aspects of vectorisation using the interpretative computer language MATLAB. The interactive nature of interpretative computing makes exploration very convenient.

The analogy between MATLAB and high performance computers is the following:

- MATLAB's slow interpretive mode corresponds to the slow scalar mode of supercomputers, or to slow poorly organised computations on a super-scalar workstation, where only one computation is done at a time;
- whereas the MATLAB's fast vector-slash matrix instructions correspond to the fast vector mode of supercomputers, or fast well organised computations on a super-scalar workstation, where many computations are done simultaneously.

In this course, we use MATLAB to learn about vector programming.[3]

Reading 2.A! → Appendix A provides a detailed overview of programming in MATLAB.

2.8.2 Vector operations

The MATLAB vector mode is much faster than its standard interpretative mode. To see this we need to measure processing time.

Functions `tic` and `toc` measure elapsed time These two functions enable the user to measure the amount of time that elapses between the issue of the two commands. The `tic` command starts the stopwatch, while the `toc` command stops the stopwatch and prints out the elapsed time. Use these functions to time the execution of the whole program or a section of code within a program.

[3] Another reason for using MATLAB rather than C, is that the compilers for languages, such as C, often optimise their compiled executable code.

Example 2.12: Using the MATLAB commands `tic` and `toc` shows that the following commands took a total of 3.4172 seconds to execute.

```
>> tic
>> cos(0:.1:10);
>> toc

elapsed_time =
3.4172
```

Actually, almost all of this time is the time it took me to type the commands: `tic` and `toc` measure elapsed time, not the actual compute time. `tic` and `toc` are unreliable if the computer or MATLAB are simultaneously doing other system tasks.

Observe that typing on one line the commands

```
tic; cos(0:.1:10); toc
```

results in very mush faster execution time because MATLAB does not have to wait for your typing.

Compare vector and scalar times To illustrate the difference between MATLAB's fast vector mode and its slow interpretative mode type the following into a file called `test_speedup.m` and execute it in MATLAB. Note: 1) to see the real difference in the R2016b version of MATLAB we need to run MATLAB as a single computational thread. You can do this by starting MATLAB with the option `-singleCompThread` (i.e. typing the command `matlab -singleCompThread` at the DOS or shell prompt); and 2) should execute it several times to get good timing results.

Exercise 2.13: Type the code in Algorithm 1 into a script called `test_speedup.m` and run it several times, noting the results each time.

The vector instruction, `c=a+b`, is many times faster than the interpretive mode involving a loop. On a real supercomputer with dedicated vector hardware the vector mode is typically twenty times faster than the scalar mode. On workstations well written code effectively utilising the vector nature of the super-scalar architecture typically is about four or more times faster than poorly written code (badly written code may be horribly slower).

Note: the command `feature accel off` tells MATLAB to not recognise the vector nature of the simple loop in Algorithm 1. Thus this command forces MATLAB to use the slow interpretive mode for the `for` loop.

By comparing its vector instructions instead of its interpretive mode, MATLAB indicates clearly, indeed exaggerates, the speed improvements attainable using algorithms appropriate for modern computer architecture.

Algorithm 1 compare scalar and vector code.

```
    feature accel off
    n=20000
    a=rand(n,1);
    b=rand(n,1);
% vector code is fast
    tic; c=a+b; tvec=toc()
% using a for loop is slow
    tic
    for i=1:n
      c(i)=a(i)+b(i);
    end
    tsca=toc()
% Speedup
    speedup=tsca/tvec
```

2.9 Loop interchange

Consider multiplying an $n \times n$ array $\boldsymbol{A} = [a_{ij}]$ by an n-vector $\boldsymbol{x}$ to give an n-vector $\boldsymbol{y}$ as the result. Mathematically,

$$\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}, \quad \text{that is} \quad y_i = \sum_{j=1}^{n} a_{ij} x_j \, .$$

The following code shows the non-vectorised MATLAB version of this code for a random $\boldsymbol{A}$ and $\boldsymbol{x}$.

Algorithm 2 worst correct code for matrix multiplication.

```
    feature accel off
    n=300
    a=rand(n,n);
    x=rand(n,1);
    tic
    for i=1:n
      y(i)=0;
      for j=1:n
         y(i)=y(i)+a(i,j)*x(j);
      end
    end
    full_interp_time=toc()
```

Exercise 2.14: Run the code in Algorithm 2 several times and observe the time.

This code cannot effectively utilise the vector mode in MATLAB. See that in the inner loop, each addition involves `y(i)` and the result has to be stored into the

same number `y(i)`; thus each addition cannot start until the previous addition has finished. The computation will be stuck doing additions serially and slowly.

One of the basic and most effective algorithmic tools is simply to interchange the order of the loops. Interchanging the loops in Algorithm 2 gives the code shown in Algorithm 3.[4]

Algorithm 3 interchange loops as a step towards better code for matrix multiplication.

```
feature accel off
n=300
a=rand(n,n);
x=rand(n,1);
tic
for i=1:n
  y(i)=0;
end
for j=1:n
  for i=1:n
     y(i)=y(i)+a(i,j)*x(j);
  end
end
loop_int=toc()
```

Exercise 2.15: Run the code in Example 3 several times and observe the time—the times will be similar to Exercise 2 because we have not yet explicitly invoked the vectorisation.

Now the inner loop is over `i` and the result of an addition involving `y(i)` is not needed for another addition until the next iteration of the outer `j`-loop. All the additions for each fixed `j` can be executed at the same time. Explicitly vectorise this innermost loop simply by crossing out its `for` and `end` (with a semi-colon to avoid printing):

```
i=1:n;
y(i)=y(i)+a(i,j)*x(j);
```

However, in this computation the vector `i` is redundant so the above is more simply just

```
y=y+a(:,j)*x(j);
```

as the colon denotes all of the jth column. Also vectorise the initialisation of `y` via `y=zeros(n,1);`. Algorithm 4 shows how to explicitly vectorise in MATLAB.

[4] Although the absolute best code is to use the expressiveness of MATLAB and simply code `y=a*x`, but that does not show the principle of loop interchange.

Algorithm 4 vectorised code for matrix multiplication.

```
feature accel off
n=300
a=rand(n,n);
x=rand(n,1);
tic
y=zeros(n,1);
for j=1:n
   y=y+a(:,j)*x(j);
end
vec_time=toc()
```

Exercise 2.16: Run the code in Example 4 several times and compare the time with earlier computations.

This vectorised code is much faster than the interpreted code given earlier. Such *simple loop interchanges to vectorise is one of our main tools* to obtain high performance from modern computer architecture.

Incidentally By far the fastest code for the matrix-vector multiply is simply `y=a*x;;` try it and see. This code uses the inbuilt BLAS and LAPACK routines that not only know how to use the vector mode to keep CPU pipelines busy, but also the routines know how to efficiently use cache (§3.4.2). BLAS and LAPACK are the public domain result of decades of work by some of the best numerical researchers—use them.

2.10 Summary

At the end of this module, you should have an understanding of:

- Computer representation of numbers;
- Taylor Series;
- Round off and truncation errors;
- Functions and scripts in MATLAB;
- Basic MATLAB structures;
- Vector and Scalar operation;
- The concepts of loop interchange;

Chapter 3 Solving equations determines system interactions

Chapter contents

3.1 Introduction

Equations represent relations between parts or components of a system. Modern science and engineering deal with complex systems that have up to millions of such interacting components. Equations of such vital complex systems are tricky to solve. Here we develop a sound foundation for solving complex system equations.

We start with linear equations as their solution empowers us to solve general non-linear equations. We also start with just one or two components, variables, as these serve to introduce the key concepts.

Linear Equations A linear equation with one variable x is an equation such as

$$2x + 3 = 0\,.$$

Its solution is $x = -3/2$. A linear equation with two variables, x and y, is of the form $ax + by + c = 0$ and, on its own, has many solutions. However, when two equations with two variables must be simultaneously satisfied, such as

$$\begin{aligned} a_1x + b_1y + c_1 &= 0\,,\\ a_2x + b_2y + c_2 &= 0\,, \end{aligned}$$

then usually only one pair of values for x and y satisfy the pair of equations. Section 3.4 discusses the solution of large systems of linear equations in many unknowns.

Non-linear Equations We also discuss the solution of systems of *non*linear equations in several variables. The simplest non-linear equation is one of the form $ax^2 + b = 0$ and has two solutions $x = \pm\sqrt{(-b/a)}$ which may be real or complex numbers.

Quadratic equations of the form $ax^2 + bx + c = 0$ are transformed to the simpler form by completing the square: $ax^2 + bx + c = (x + b/2a)^2 + c/a - (b/2a)^2 = 0$ from which we see that the equation will be satisfied if $x + b/2a = \pm\sqrt{(b/2a)^2 - c}$. The familiar quadratic formula follows. For example, such equations arise in the solution of second order linear constant coefficient differential equations. Higher order equations require the solution of higher degree polynomial equations. The eigenvalue problem also, at least theoretically, leads to the need to solve high degree polynomial equations.

Henrik Abel showed that solutions for quintic and higher degree equations cannot in general be expressed in terms of formulae using the elementary functions. For example, the equation $x^5 + 2x^3 - x + 1 = 0$ has only one real solution and it is in the range $-0.87839 < x < -0.87838$. We can improve the approximation to as many decimal places as we like but we have to do this through computation—there is no formula for the solution.

Physical processes and systems arising in scientific and engineering applications are often governed by non-linear equations in one or more variables. Polynomial equations are just one example of non-linear equations. Equations involving any of the trigonometric, logarithmic or any other of the transcendental functions will also be non-linear equations. For example, $x - \cos x = 0$ is a simple non-linear equation. There is usually no formula for the solution of such equations. In general, solutions must be found by numerical approximation techniques.

Example 3.1: In compressible fluid dynamics the flow properties behind an oblique shock wave are determined by solving the equation

$$\left[M_1^2(\gamma + \cos 2\beta) + 2\right] \tan\theta \sin\beta - 2\cos\beta(M_1^2 \sin^2\beta - 1) = 0$$

for the unknown wave angle β, where M_1 is the known upstream Mach number, θ is the known deflection angle and γ is a known gas constant. In this case every variable in the equation would be known, except the only unknown β.

Example 3.2: In chemistry, the equilibrium point of certain hydrocarbon reactions is found by solving the equation

$$\frac{e^{-1/T_s} + \kappa\rho_b e^{-\alpha/T_s}}{\rho_{t1} + e^{-1/T_s} - \rho_b e^{-\alpha/T_s}} - \frac{\beta}{\mu_i}(T_s - \theta_a) = 0$$

for the equilibrium temperature T_s, where μ_i, θ_a, β, α, ρ_b, ρ_{t1} and κ are given reaction parameters related to concentration, temperature and energy. We would need to solve for T_s.

To study and understand the behaviour of such processes and systems under a variety of operating conditions (such as pressure, temperature, stress, velocity) requires

solutions to the governing non-linear equations. The examples above involve equations that are not particularly complicated but analytic solutions, that is solutions obtainable from an explicit formula, cannot be found. Numerical methods must be employed to seek accurate (to a specified number of significant figures) *approximate* solutions. Methods for doing this will be developed here.

The task for us is to find values of x, known as zeros or roots, for which

$$f(x) = 0\,,$$

where $f(x)$ is some non-linear function. In the examples above x was a single number or scalar (β or T_s).

A further step in complexity involves *systems* of non-linear equations in which $\boldsymbol{x}$ denotes a vector of variables and $\boldsymbol{f}$ represents a vector of functions. Often a numerical approximation to the solution begins by approximating the function values by linear functions. In the case of a single variable, interpret this line as approximating the function's values by the values of the tangent line to the function at some point. For example, to solve the equation $\sin x = 1 - x$ we might guess that the solution is somewhere near $x = 0$ and approximate the values of $\sin x$ with those of its tangent line at the origin, namely $y \approx x$. The original equation is then replaced by the linear equation $x = 1 - x$ which readily gives $x = 0.5$ as an approximation. The solution to the original problem lies between 0.51097 and 0.51098, so we have made a good start. For systems of non-linear equations, the first approximation would involve the solution of an approximating system of linear equations.

Section 3.2 studies the solution of non-linear equations in one variable and, Section 3.5 explores solutions of non-linear equations in two variables, but we will not go beyond three variables with non-linear equations. However, because of their considerable importance, including their use as first approximations to non-linear systems, we will study the solution of large linear systems in some detail in Sections 3.3–3.4.

General objectives for this module

- To gain a working knowledge and understanding of some basic iterative numerical techniques for solving non-linear equations in one or more variables
- To write MATLAB programs that implement these methods on a computer
- To be aware of potential hazards that can arise when solving linear systems on a computer
- To apply iterative methods for solving linear systems of equations, and to understand when these methods can be used successfully

Prerequisite knowledge

- Elementary calculus in one variable, first derivatives, local tangent approximations

- A working knowledge of partial derivatives for functions of several variables, and the gradient function are required for the solution of several simultaneous non-linear algebraic equations. This is a small portion of the course.
- Matrix representation for systems of linear equations, and some general theory of linear algebraic equations.

3.2 Iterative solutions of non-linear equations in one variable

Specific Objectives To understand, apply and program Newton's iterative method or the Bisection method to solve a non-linear equation in one variable.

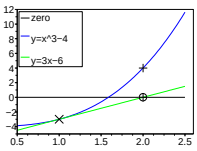
To understand what is meant by the *convergence* of an iterative technique and how the speed of convergence may vary between different methods.

3.2.1 Newton's method

Newton's method is an *iterative* method; that is one that starts with some initial approximation (perhaps a guess) to the solution and produces a sequence of improved estimates by repeated application of some procedure.

Example 3.3: one step towards a cube root Let us start by computing the cube root of four: the cube root is the solution of the equation $f(x) = x^3 - 4 = 0$. Any hand calculator will tell us the answer is $\sqrt[3]{4} = 1.5874$; but how does it know this? By Newton's method (or the Bisection method or a combination of both).

Newton's method is iterative: we start with a guess, and then find some way to improve it. Here we know that $1^3 = 1$ and $2^3 = 8$ so we are reasonably sure that the cube root of four lies somewhere between one and two. Start with a crude guess of $x = 1$, and let us see how one step of Newton's method indicates $x = 2$ is a better guess.



First Newton step from $(1, -3)$ to the next estimate $x = 2$ along the (green) line.

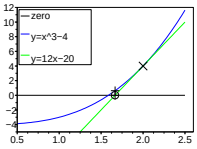
Solution: Start with the guess that $x = 1$, where the function $f(x)$ is $f(1) = 1^3 - 4 = -3$. Certainly f is not near zero. But *near* $x = 1$ the graph $y = f(x)$ will be approximately linear: use this approximate linear dependence to make a new guess in the following manner as shown to the left. The curve goes through the point $(x, y) = (1, -3)$ and has slope $f'(1) = 3 \cdot 1^2 = 3$ as the derivative $f'(x) = 3x^2$. Thus the straight line $y - (-3) = 3(x - 1)$ approximates the curve: it comes from the rule that

$$(\text{the change in } y) = (\text{slope}) \cdot (\text{change in } x).$$

Rearranging with algebra, this straight line is $y = 3x - 6$. Now although the straight line is only *locally* approximate, Newton's method is to use this straight line to derive a new guess for the solution of $f(x) = x^3 - 4 = 0$ by instead solving the approximate $f(x) \approx 3x - 6 = 0$. Algebra then tells us the new guess to the cube root of four is thus $x = 2$.

Example 3.4: more steps towards the cube root Do some more Newton steps towards solving $f(x) = x^3 - 4 = 0$ in order to find the cube root of four.

Continue Example 3.3 by following the same procedure but now step from the second guess, $x = 2$, towards the solution.



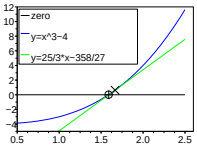
Second Newton step from $(2, 4)$ to the next estimate $x = 5/3$ along the (green) line.

Solution: The function value at $x = 2$ is $f(2) = 2^3 - 4 = 4$. That is, the curve $y = f(x)$ goes through the point $(x, y) = (2, 4)$. Let us approximate it by a straight line with the same slope $f'(2) = 3 \cdot 2^2 = 12$, namely $y - 4 = 12(x - 2)$ which you recall comes from

$$(\text{the change in } y) = (\text{slope}) \cdot (\text{change in } x).$$

Rearranging, this straight line is $y = 12x - 20$. Newton's method is to make a new guess by finding when this approximate straight line is zero: namely, solve $12x - 20 = 0$ to find the new guess is $x = 5/3$.

Do another step towards the cube root. The function value is $f(5/3) = (5/3)^3 - 4 = 17/27$ and the slope of the function is $f'(5/3) = 3(5/3)^2 = 25/3$. Thus a straight line approximating the curve $y - f(x)$ is $y - \frac{17}{27} = \frac{25}{3}(x - \frac{5}{3})$, that is $y = \frac{25}{3}x - \frac{358}{27}$. Using this straight line to suggest the new guess to the cube root we solve $\frac{25}{3}x - \frac{358}{27} = 0$ to deduce the new guess is $x = \frac{358}{225} = 1.5911$. This is only 1% in error.



Third Newton step is small.

Generally label the successive guesses, or iterates, generated by Newton's method with a subscript so we can then discuss the sequence of approximations, $x_0, x_1, x_2, x_3, \ldots$, to the solution of $f(x) = 0$.

Example 3.5: compute each step with algebraic rule In the previous example the guesses would be labelled $x_0 = 1$, $x_1 = 2$, $x_2 = \frac{5}{3}$, $x_3 = \frac{358}{225}$, and so on. These labels empower us to derive a straightforward formula for each step of Newton's iterates. This formula then empowers computation.

Given the guess $x = x_k$ for the cube root, what is the new guess $x = x_{k+1}$ by Newton's method?

Solution: The function value $f(x_k) = x_k^3 - 4$ and the slope of the curve is $f'(x_k) = 3x_k^2$. Hence an approximating straight line is $y - x_k^3 + 4 = 3x_k^2(x - x_k)$ which we rearrange to $y = (3x_k^2)x - 2x_k^3 - 4$. Newton says to determine the next guess $x = x_{k+1}$ by finding when this straight line is zero: namely solve $(3x_k^2)x - 2x_k^3 - 4 = 0$ to deduce $x_{k+1} = (2x_k^3 + 4)/(3x_k^2)$.

The formula $x_{k+1} = (2x_k^3 + 4)/(3x_k^2)$ encapsulates all the preceding arguments.

Algorithm 5 uses this formula to iteratively find the cube root of four. Execute Algorithm 5. Just five Newton steps are all it takes to get the cube root to seven decimal places.

What happens when we start from $x_0 = 0$? The immediate divide by zero causes the algorithm to fail immediately. Other starting values can also cause failure: find another.

General aspects of this example

- Newton's method can work spectacularly well: it can converge to the solution in just a few iterations (steps).

Algorithm 5 Newton's method to find the cube root of four.

```
% Find the cube root of four via Newton's method
% Tony Roberts, March 2007
% Modified: Harry Butler, Jan 2017
x(1)=1
for k=1:10
  x(k+1)=(2*x(k)^3+4)/(3*x(k)^2)
end
```

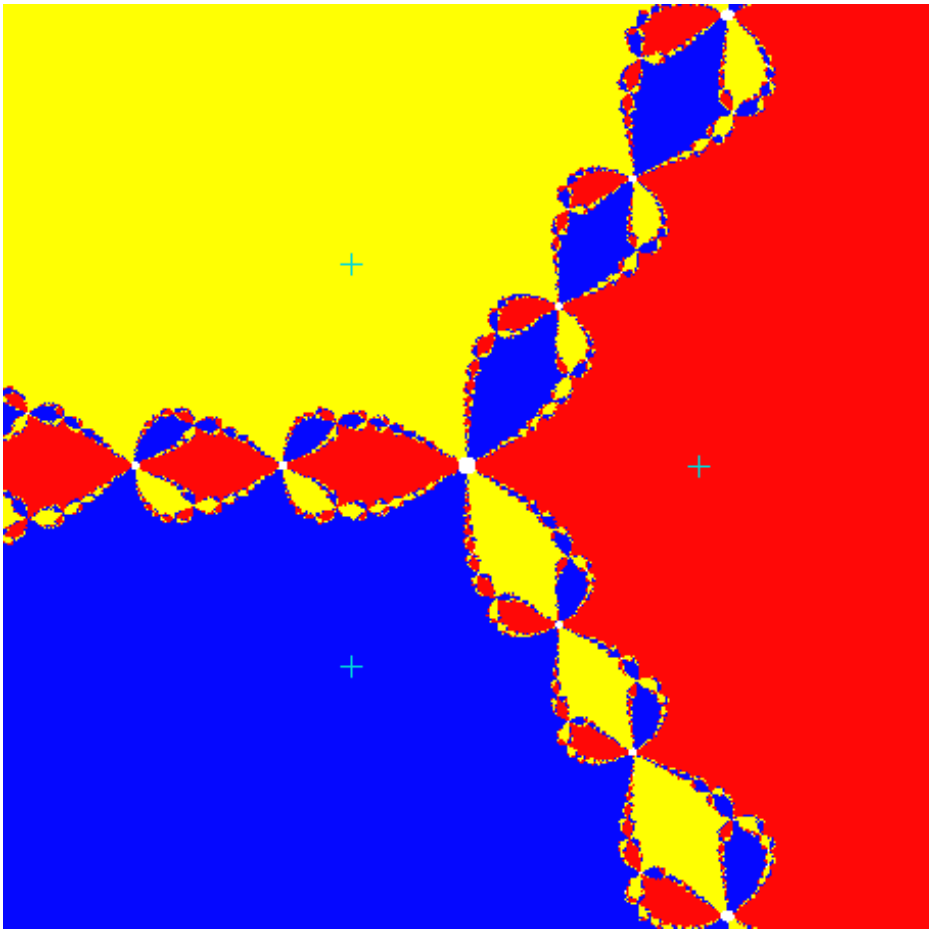


Figure 3.1: the complex number equation $z^3 = 4$ has three roots. Newton's iteration converges to a different root depending upon the starting value for the iteration. This figure colours each starting value according to the root Newton's method eventually reaches. Note the fascinating boundary between the domains of attraction of each of the three complex roots.

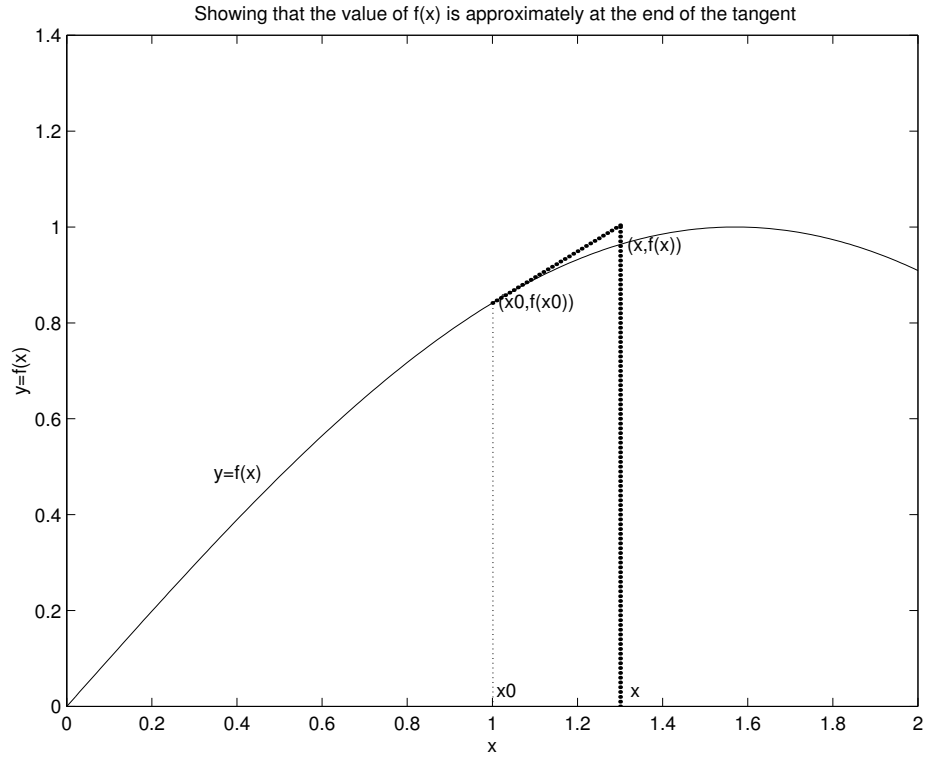


Figure 3.2: If the function is approximately linear, near x_0 the tangent line estimates function values.

- However, the sequence of iterates can converge to two or more different values depending on your starting value, see Figure 3.1 for just one well established example.
- Further, Newton's iteration occasionally fails for no apparent reason—until you investigate further.

General formula for Newton's method Consider a starting guess, x_0, of the solution to the non-linear equation $f(x) = 0$. The function values $f(x)$ are approximated by those of the tangent line to the function at x_0:

$$y - f(x_0) = f'(x_0)(x - x_0)\,.$$

We assume that the derivative f' of f can be computed at x_0; that is, that x_0 is not a corner or point of discontinuity of the function. Thus we approximate the curve $y = f(x)$ by the linear

$$y = f(x_0) + (x - x_0)f'(x_0)\,.$$

Figure 3.2 shows how the value of the function $f(x)$ is approximated by the value on the tangent line, where the tangent is drawn at x_0, provided that x is close to x_0.

Our first approximation to solving $f(x) = 0$ is to solve this linear equation for when the *approximation*, y, to $f(x)$ is zero,

$$f(x) \approx y = f(x_0) + (x - x_0)f'(x_0) = 0\,.$$

Hence we have the approximation equation

$$(x - x_0)f'(x_0) + f(x_0) = 0\,.$$

Rearranging this equation, with $y = 0$, then gives

$$x = x_0 - \frac{f(x_0)}{f'(x_0)},$$

the next *approximate* solution which, we trust, will be better than the initial x_0.

We treat this new approximation as a new estimate of the solution, x_1, and repeat the procedure; aiming to get an even better approximation. If the process does produce better approximations to the solution, then the process is said to *converge*.

Representing the sequence of approximations that we obtain by repeating the process by $x_0, x_1, x_2, \ldots, x_k, x_{k+1}, \ldots$ then entry x_{k+1} in the sequence is obtained from the previous one x_k from the computation

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}.$$

Geometrically, the updated value x_{k+1} at each step is simply where the tangent line to $f(x)$ from x_k cuts the x axis.

Taylor Series derivation of Newton's Method Newton's method can also be derived directly from the Taylor series. Starting with Equation 2.13

$$f(x + h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \cdots$$

we can assume that x is our initial guess of the root of the function f, and that h is the correction to x that will give us the root we are looking for. That is, we are using the information we know at the guess to construct the Taylor series. This means that $f(x+h) = 0$ as it is the root, and the unknown correction h to our guess x must be

$$h = -\frac{f(x)}{f'(x)} - \frac{1}{2}\frac{f''(x)}{f'(x)}h^2 - \cdots$$

As h is unknown we drop the terms in the series containing an h— that is, all terms except the first one. Therefore the new value for x becomes

$$\begin{aligned} x_{k+1} &= x_k + h, \\ x_{k+1} &= x_k - \frac{f(x_k)}{f'(x_k)} \end{aligned}$$

An important insight into Newton's method is gained by using a Taylor series derivation—the dropped terms of the series. Initially we started by saying that h is the difference between our guess and the true root—but this is only true if we retain all the terms of the series. As we only retain the first term of the series this introduces an error—of order h^2 (normally written $O(h^2)$) the coefficient of the first and hopefully largest of the dropped terms. This tells us we will be converging to the solution with an error at each iteration of $O(h^2)$—Newton's method is therefore called a "second'' order method. A second order method converges faster (in fewer iterations) than a first order (or linear) method, where the error is $O(h)$. In a second order method, as we converge to the solution, h is getting smaller but the error is getting smaller much faster.

3.2.2 Terminate iteration upon convergence

So far our Newton's iteration **continues for a fixed, preset number of steps**. In the examples, Newton's method usually converges to a solution but in different examples it takes different numbers of steps. *How can we terminate Newton's method once we have found an acceptable solution?*

The simplest way is to stop the Newton iteration once the iterates x_k are no longer changing. You *must never* test for equality of real numbers in a computer: roundoff error in any computation makes equality almost meaningless. As we have seen modern computers work with just over 15 digit accuracy in double precision, so the concept of the iterates 'not changing' is actually just that the iterates are within about 10^{-15} of each other: stop iterating when the iterates are close to each other. That is, terminate the iteration when $|x_{k+1} - x_k| < \epsilon$ where the tolerance ϵ is some small number like 10^{-13}. Never attempt to converge a solution to the supposed full accuracy of the computer—roundoff will ensure that the last couple of digits are meaningless!

How should you program the termination? A `while` loop springs to mind. But usually it is shorter, simpler and safer to use a `for` loop to provide a 'safety net' of a fixed (largish) maximum number of iterations. Now, within any loop, executing the `break` command transfers execution to the statements immediately after the loop. Then within the iterative `for` loop, use an `if` test to invoke the `break` command upon convergence.

Example 3.6: terminate when converged Apply Newton's method to find $\sqrt{5}$ by solving $x^2 - 5 = 0$. Further, under what conditions might you need as many as say 20 iterations?

Solution: In this case we have $f(x) = x^2 - 5$ and so $f'(x) = 2x$. The Newton iteration formula is thus

$$x = x_0 - \frac{x_0^2 - 5}{2x_0} = \frac{2x_0^2 - x_0^2 + 5}{2x_0}$$

which reduces to

$$x = \frac{x_0^2 + 5}{2x_0}.$$

The general iteration formula is thus

$$x_{k+1} = \frac{x_k^2 + 5}{2x_k}.$$

Starting with $x_0 = 2.5$ produces the sequence of iterates

$$x_1 = 2.25,\ x_2 = 2.23611,\ x_3 = 2.23607\ \ldots$$

which appears to be converging rapidly to the exact solution, since

$$2.2360679774 < \sqrt{5} < 2.2360679775$$

Since we know the answer for this case we can estimate the error $e_k = x_k - \sqrt{5}$ at each step, giving the error sequence 0.2639, 0.01393, 4.31×10^{-5}, 4.16×10^{-10}.

Visualise some iterations from an alternative starting point by writing and running the MATLAB Algorithm 6.

If x_0 is very far away from $\sqrt{5}$, then many iterations are necessary: for very large x_k, the iteration gives $x_{k+1} \approx \frac{1}{2}x_k$ as 5 is negligible compared to a very large x_k^2. Thus starting from $x_0 \approx 2^{20}$ will require about twenty iterations to reach $\sqrt{5}$.

Algorithm 6 Newton's method on a simple quadratic.

```
% Script:  root5
% Pre-defined variables (must exist in the workspace):
%   x0 - the starting value for the right hand side
% Created/over-written variable:
%    x - the value for sqrt(5) after iteration
% A script for solving  x^2-5 by Newton iteration.
% Written: Tony Roberts, March 2007
% Modified: Leigh Brookshaw, January 2010
% Modified: Harry Butler, January 2017
  if ~exist('x0')
    error('Please give x0 a value')
  end
  x = x0
  maxiter=100;
  for k = 1:maxiter
    xnew = (x^2+5)/(2*x)
    if abs(xnew-x)<1e-13
      break
    end
    x = xnew
  end
  if k>=maxiter
    error('Failed to converge to root');
  end
```

But suppose the problem is to find the value of some national budget which is of the order of billions of dollars? Then testing for closeness by requiring iterates be within 10^{-13} of each other is useless. For amounts of billions of dollars, and a computer with about 15 digit accuracy, roundoff error affects digits at the level of 10^{-5}. Thus roundoff will cause meaningless fluctuations at a level very much larger than the test is supposing. The answer in many cases is to make the termination test relative in magnitude.

Depending upon the application, you will need to invoke either

- an *absolute test*, $|x_{k+1} - x_k| < \epsilon$, or
- a *relative test*, $|x_{k+1} - x_k| < \epsilon|x_k|$, or
- since the aim is really to make the function small, you might terminate iteration with the *absolute test* $|f(x_k)| < \epsilon$, or

- the *relative test* $|f(x_k)| < \epsilon|f(x_0)|$.

Which? The difficulty is that in different circumstances, each of these tests can be misleading or fail. Experience will help you decide. In the interim, be careful.

3.2.3 Unknown derivative? No problem

If $f(x)$ is not available from analytic formulae, but instead the output from some computer program, then it may not be possible to find its derivative $f'(x)$. Use a *Taylor series* to approximate the derivative—

$$f(x+h) \approx f(x) + f'(x)h$$

rewriting in terms of the kth iterate gives

$$f'(x_k) \approx \frac{f(x_k+h) - f(x_k)}{h}$$

at each step. Typically choose this h in the range 10^{-6} to 10^{-8}. Also note, that an approximation for $f'(x)$ will add more "ignored" error terms to the Newton's formula which means it will in theory slow the convergence rate—in practice this is not noticeable.

Example 3.7: Apply Newton's method to find a value of x for which

$$\exp(\sin(x^2 - 3x + 2) - x) = 1\,.$$

In this case $f(x) = \exp(\sin(x^2 - 3x + 2) - x) - 1$, a composite function which Algorithm 7 calculates together with its approximate derivative.

The solution is $x \approx 0.5727291378$.

Exercise 3.8: Solve

$$\exp(\sin(x^2 - 3x + 2) + x) = 1\,,$$

with a slight modification of the script Algorithm 7.

Example 3.9: Studying the convergence of approximates to a final solution is enhanced by keeping a list of approximates, in say vector variable X, and plotting them against the iteration number, say $K = 0 : n$. Note the easy but dirty way the vector of results is built.[1]

Find all the solutions for the equation

$$\sin(2 - 3x^2 - x^4) = x\,.$$

Experiment with different values of $x0$, especially try 0.9, 1 and 1.1 for $x0$, choosing appropriate values for n. Solutions are approximately −0.9778, −0.9056 and 0.6251 .

Solution: Our computational sequence, given that $x0$ and n are given some values, will then be Algorithm 8

Algorithm 7 Newton's method on a complex function.

```
% Script:  EX2.m
% Pre-defined variables (must exist in the workspace):
%   x0 - the starting value for the right hand side
% func - the function to find the zero
% Created/over-written variable:
%    x - the approximate solution after n iterations
%    h - change to approximate derivative
%
% A script for solving  f(x)=0 by Newton's Method
% Written: Tony Roberts, March 2007
% Modified: Leigh Brookshaw, January 2010
% Modified: Harry Butler, January 2017

  if ~exist('x0')
    error('Give variable x0 a value')
  end
  h=1e-6;
  x = x0
  maxiter=100;
  for k = 1:maxiter
    fx=f(x);
    dfdx=(f(x+h)-fx)/h;
    xnew = x - fx/dfdx
    if abs(xnew-x)<1e-13*abs(x)
      break
    end
    x = xnew;
  end
  if k>=maxiter
    error('Failed to converge');
  end

% specify the function in file f.m
  function fx=f(x)
    fx = exp(sin(x^2 -3*x + 2)-x)-1;
  end
```

Algorithm 8 store intermediate results in the iteration.

```
% A script for solving  f(x)=0 by Newton's Method.
% Written: Tony Roberts, March 2007
% Modified: Leigh Brookshaw, January 2010
% Modified: Leigh Brookshaw, March 2011
% Modified: Harry Butler, Jan 2017

  if ~exist('x0')
    error('Give variable x0 a value')
  end
  h=1e-6;
  maxiter=100;
  x = x0;  X = [x];  K = [0];
  for k = 1:maxiter
    fx=f(x);
    dfdx=(f(x+h)-fx)/h;
    x = x - fx/dfdx;
    if abs(x-X(end))<1e-13*abs(x), break, end
    X=[X,x]; K=[K,k];
  end
  if k>=maxiter
    warning('Failed to converge');
  end
  plot(K,X,'*-')


% define the function in file f.m
  function fx=f(x)
    fx = sin(2-3*x^2-x^4)-x;
  end
```

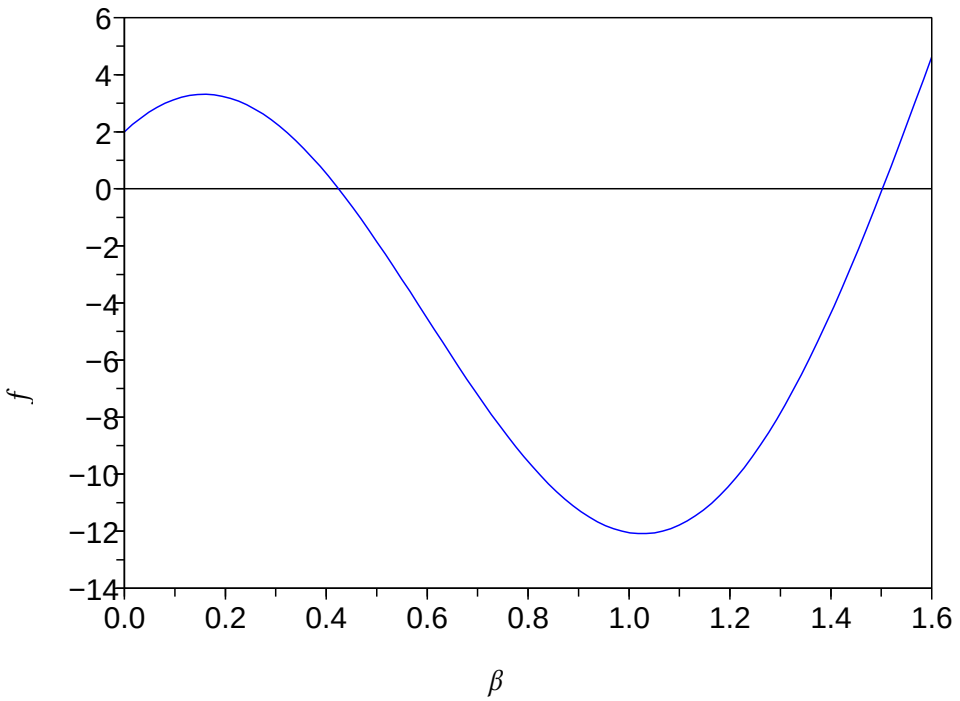


Figure 3.3: A plot of the shock function $f(\beta)$ against β, to provide a starting guess for Newton's method. Note there are two solutions, corresponding to weak and strong shock waves. The weak solution, with smaller wave angle β, will be pursued here.

You must be sure that a solution exists before accepting output from your Newton's Method computations. An easy way to do this, and to get an initial approximation for the solution, is to graph the function.

Example 3.10: Example 3.1 gives the non-linear equation governing the wave angle β for an oblique shock wave as

$$\begin{aligned} f(\beta) &= \left[M_1^2(\gamma + \cos 2\beta) + 2\right] \tan\theta \sin\beta \\ &\quad - 2\cos\beta(M_1^2 \sin^2\beta - 1) \\ &= 0\,, \end{aligned}$$

where M_1, γ and θ are parameters. Solve this equation for the parameters $M_1 = 5$, $\gamma = 1.4$ and $\theta = \pi/12$.

Physically, this represents a Mach 5 supersonic stream being turned through a sharp 15 degree corner. The task is to find the angle the shock wave makes with the original flow direction. Assume that $\theta < \beta < \pi/2$.

Perhaps find a suitable starting guess for the iterations by plotting the function.[2] Figure 3.3 gives a plot of $f(\beta)$ against β over the range of interest,

[1] Why is this way of building a vector 'dirty'? Because MATLAB has to continually recreate and reassign more storage for the vector variables as they increase in size in each iteration. Such poor practice is only usually acceptable outside the innermost loops.

[2] Such a plot requires a large number of function evaluations, more than that required to solve the equation, and is thus not practical in general. Such plots are for illustration only.

Table 3.1: Newton's iterations for the shock wave angle solution.

n	β_n	$f(\beta_n)$	$f'(\beta_n)$
0	0.5	-1.842	-25.90
1	0.428876	-9.959×10^{-2}	-22.84
2	0.424517	-5.140×10^{-4}	-22.61
3	0.424494	1.482×10^{-8}	-22.60
4	0.424494	-2×10^{-14}	-22.60

indicating two possible solutions; one near $\beta = 0.4$ and one near $\beta = 1.5$. These correspond to *weak* and *strong* shock waves respectively and we will seek the weak solution, with smaller angle β. From the plot this solution is located at $\beta \approx 0.4$ radians, which should serve as a good initial guess, but we use $\beta_0 = 0.5$ just to add an extra iteration for display.

The function is complicated When we factor in human time, it is quicker to compute the derivative using the Taylor series approximation. Modify Algorithms 7 or 8 by supplying the function

```
function fx=f(beta)
  m1=5; gamma=1.4; theta=pi/12;
  fx = (m1^2*(gamma+cos(2*beta))+2)*tan(theta)*sin(beta) ...
    -2*cos(beta)*(m1^2*sin(beta)^2-1);
end
```

Algorithm 8 returns Table 3.1 as the Newton's iterates. Newton's method achieves a function value of about 10^{-14} after four iterations. At this point convergence is attained and the iterations terminated, giving a shock wave angle of 0.4245 radians.

Exercise 3.11: Try some alternative starting points for the previous Example and see when the iterations start to diverge. Also try and find the other solution near $\beta = 1.5$.

Answer: Roughly: for $\beta_0 \leq 0.2$ iterations converge to unphysical negative β; for $0.3 \leq \beta_0 \leq 0.8$ iterations converge to $\beta = 0.4245$; for $0.9 \leq \beta_0 \leq 1.1$ iterations converge to various physical and unphysical solutions; for $1.2 \leq \beta_0 \leq 2$ iterations converge to $\beta = 1.5023$; otherwise iterations go wild.

Exercise 3.12: The frequencies of a vibrating beam come from solutions of the equation

$$\cos x \cosh x = 1 .$$

Use Newton's method to find the solution near $x = \frac{3}{2}\pi$.

Answer: $x = 4.7300$

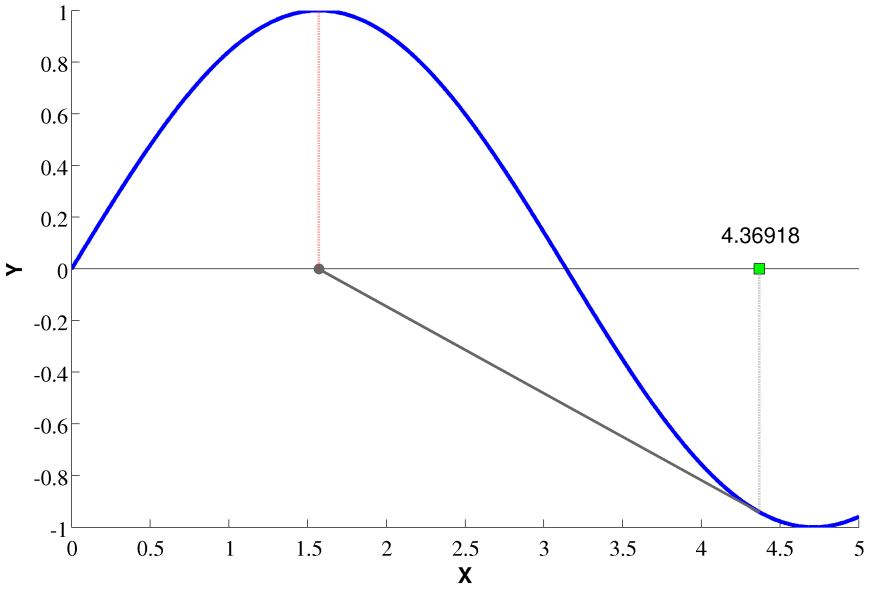


Figure 3.4: Newton's method fails with a zero derivative for the function $\sin(x) = 0$ if you choose an unlucky initial guess!

3.2.4 Newton method fails! What to do?

Newton's method works spectacularly well under the right conditions. Example 3.6 shows that the number of correct digits can be doubled in a single iteration. However, with a poor choice of starting guess it can also fail spectacularly. Also note from the iteration formula that the derivative of the function $f'(x)$ appears in the denominator, so when this is sufficiently *small* the iterations may diverge. Figures 3.4, 3.5, 3.6 and 3.7 show some of the failure states of Newton's method.

Newton's method is not the only method for finding the roots of functions—but it is the most widely used and converges the fastest. But like all numerical methods it can fail—so it is a good idea to know at least one other method to fall back on if Newton's method is proving fractious with your function. Alternative methods for finding roots are Bisection, Secant (or False Position) and others. The simplest is the Bisection method.

Bisection Method The main advantage of this method is that it *cannot fail* assuming there is a root to find. The disadvantage is that it can be slow to converge to the solution. So it is a good alternate choice when a faster converging method such as Newton's method fails.

The method is simple—over some interval $[a, b]$ the function $f(x)$ is known to contain at least one root because it changes sign, $f(a)f(b) < 0$. Evaluate the function at the interval's midpoint $c = (a + b)/2$ and examine its sign. Is $f(c)f(a) < 0$ or $f(c)f(b) < 0$—which side of the root is c to be found? Use the midpoint c to replace whichever endpoint has the same sign. Repeat until the interval is within the desired tolerance (see Figure 3.8).

Since with each iteration the interval is halved, after n iterations the root will be

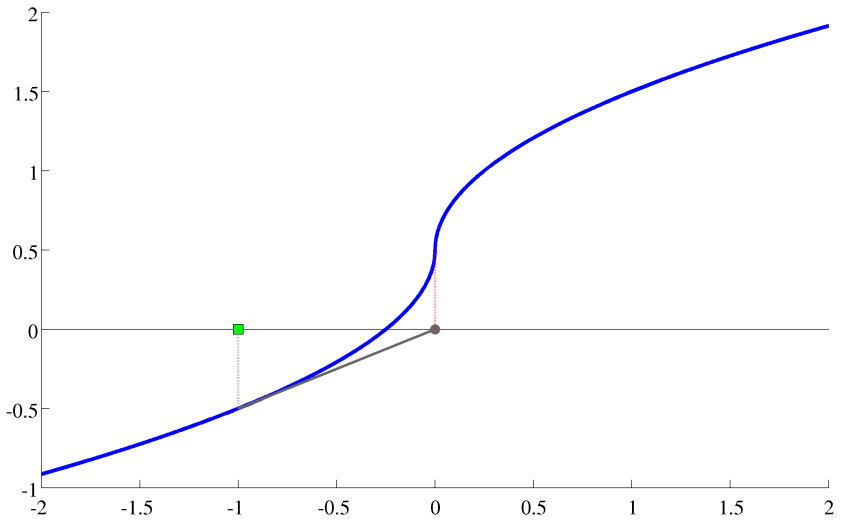


Figure 3.5: Newton's method fails with an infinite derivative for the function $\text{sign}(x) * \sqrt{|x|} + 0.5$ if the initial guess is -1.

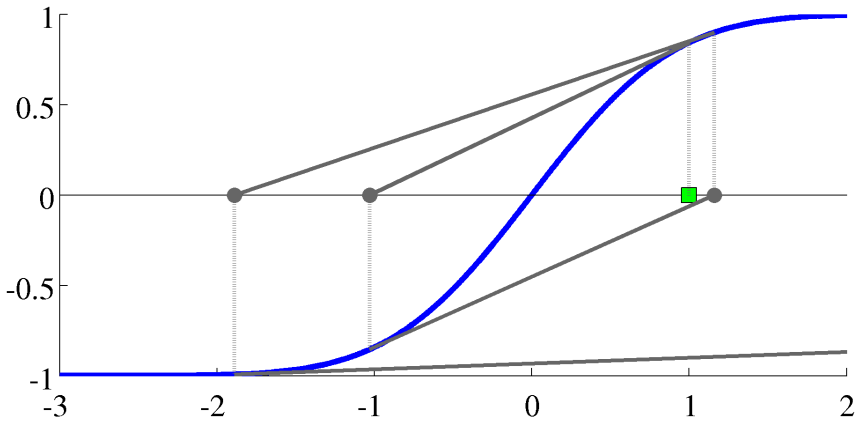


Figure 3.6: Newton's method diverges for the function $\arctan(x)$ if the initial guess is 1.4.

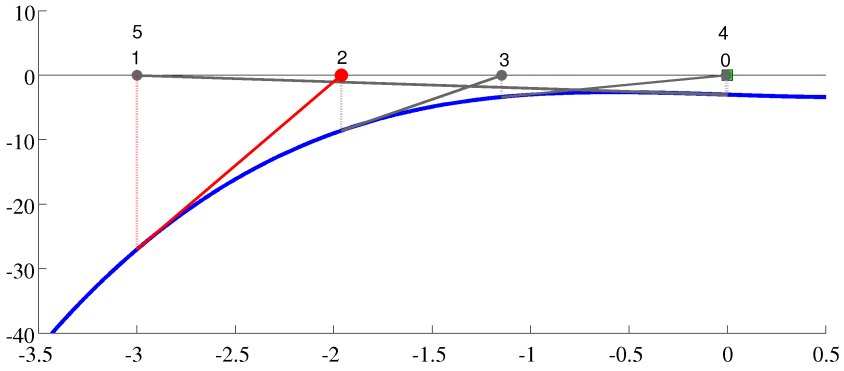


Figure 3.7: Newton's method fails to converge for the simple polynomial $x^3 - x - 3 = 0$ if the initial guess is $x = 0$. The iterations start to cycle.

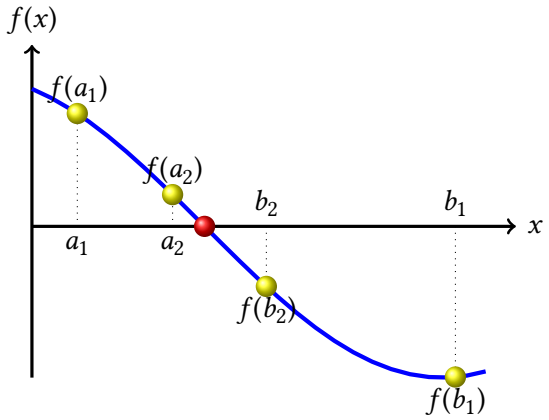


Figure 3.8: Bisection method: successive intervals are (a_1, b_1), (a_1, b_2), (a_2, b_2), converging on the root

within an interval ϵ_n of size:

$$\epsilon_n = \frac{|b - a|}{2^n}.$$

Thus, we know in advance the number of iterations required to achieve a given tolerance in the solution,

$$n = \log_2 \frac{|b - a|}{\epsilon},$$

where ϵ is the desired final tolerance.

Example 3.13: Solve the shock wave equation of Example 3.10 using Algorithm 9. With an initial guess of the interval as $(0.3, 0.5)$, before we even start the calculation we know it should take approximately 41 iterations if we set the convergence criterion to 10^{-13}. Compare that to Newton's method for the same problem (See table 3.1) which took 4 iterations.

3.2.5 Speed parametric exploration

In scientific and financial calculations we often have to explore how an output quantity of interest varies with some parameter. For example, the Reserve bank may execute simulations for many different values of the basic interest rate; they then set the Australian interest rate at a value to achieve optimum outcomes for society. Exploring outcomes over a range of parameters is called a *parametric study*.

Employing a vector mode of computation empowers fast and efficient processing. Let us return to the problem of the previous example.

Example 3.14: Reconsider the problem of Example 3.10 where we determined the shock angle β for some given angles θ. Suppose, now, we are interested in angle β for all angles θ in some range, say for a hundred values of θ over the range $0 \leq \theta \leq \pi/6$.

The simple answer is to repeat the earlier algorithm, but execute it for each angle θ. Using numerical differentiation to determine the derivative of the function, you might write Algorithm 10.

Algorithm 9 Bisection method.

```
% A script for solving  f(x)=0 by the Bisection Method.
% Written: Leigh Brookshaw, January 2010
% Written: Harry Butler, January 2017

  if ~exist('a1'),  error('Give variable a1 a value'), end
  if ~exist('b1'),  error('Give variable b1 a value'), end
  if f(a1)*f(b1)>0, error('Values do not bracket root'), end


  eps=1e-13;
  maxiter=int32(log2(abs(b1-a1)/eps))+1
  a = a1;  b = b1;
  for k = 1:maxiter
    c=(a+b)/2;
    if f(c)*f(a)<0
      b=c;
    else
      a=c;
    end
    if abs(a-b)<2*eps, break, end
  end
  if k>=maxiter
    error('Failed to converge');
  end
  k
  root = (a+b)/2

% define the function in file f.m
  function fx=f(beta)
    m1=5; gamma=1.4; theta=pi/12;
    fx = (m1^2*(gamma+cos(2*beta))+2)*tan(theta)*sin(beta) ...
    -2*cos(beta)*(m1^2*sin(beta)^2-1);
  end
```

Algorithm 10 simple scalar code for a parametric study employing Newton's method.

```
% find zeros of this function of beta
% parametrised by theta, but individually
% is slow.
% Created: Tony Roberts, March 2007
% Modified: Harry Butler, Jan. 2017

h=1e-6; % numerical differentiation step
theta=linspace(0,pi/6,100);
tic
for i=1:length(theta)
  beta(i)=0.5;
  for newton=1:9
    fn=f(beta(i),theta(i));
    dfdb=(f(beta(i)+h,theta(i))-fn)/h;
    beta(i)=beta(i)-fn/dfdb;
  end
end
tsca=toc()

function [y]=f(beta,theta)
  M1=5; gamma=1.4;
  y=(M1^2*(gamma+cos(2*beta))+2)*tan(theta)*sin(beta) ...
    -2*cos(beta)*(M1^2*sin(beta)^2-1);
end
```

- the function `f` computes the function we wish to zero;
- `theta` contain the range of angles θ;
- the innermost loop is Newton's method, here assuming just nine iterations are enough for convergence;
- the outer loop finds β for each given θ;
- the program executes in about `tsca` = 0.163 seconds on my computer.

Loop interchange speeds up Now speed up this code by changing the order of the loops; recall Section 2.9. Here each step of Newton's iteration works *simultaneously across all* angles θ to simultaneously improve *all* the different approximations β to the zeros of the function. Revise the earlier Algorithm 10 to become Algorithm 11 which is about forty times faster.

Algorithm 11 fast vector code for a parametric study employing Newton's method.

```
% find zeros of this function of beta
% parametrised by theta.  After loop
% interchange, simultaneous vector
% computation is fast.

h=1e-6; % numerical differentiation step
theta=linspace(0,pi/6,100);
tic
beta=0.5*ones(theta);
for newton=1:9
  fn=f(beta,theta);
  dfdb=(f(beta+h,theta)-fn)/h;
  beta=beta-fn./dfdb;
end
tvec=toc()
speedup=tsca/tvec

function [y]=f(beta,theta)
  M1=5; gamma=1.4;
  y=(M1^2*(gamma+cos(2*beta))+2).*tan(theta).*sin(beta) ...
    -2*cos(beta).*(M1^2*sin(beta).^2-1);
end
```

- the introduction of the `.*` and `.^` operators in the function makes these multiplications and exponentiations component by component in the vectors, hence evaluating the function for every component of the supplied vectors `beta` and `theta` (β and θ);
- initialise `beta` to a vector of initial approximations
  (**note**: For MATLAB the initialisation line must be modified to `ones(size(theta))`);
- apply nine iterations of Newton's method for each θ simultaneously;

- the only change in the loop is the use of the `./` operator to do component wise division of the function values by the corresponding derivatives;
- the program executes in about `tvec` = 0.004 seconds on my computer;
- *the vector speedup is by a factor of about 40.*

Conclusion Loop interchange and using component wise multiplication and division empowers us to employ vector computations and hence to execute code an order of magnitude faster.

Termination? The simplest way to terminate the iterations for each of the parameters is to break out of the loop once *all* of the iterates are not changing. That is, all the differences from one iteration to the next are very small. In MATLAB, a logical comparison is true overall only when all components are true. Thus the comparison `abs(bnew-beta)<1e-13*abs(beta)` compares the vector quantities on both sides and overall is true only when each and every component on the left is smaller than the corresponding component on the right. Just modify the code by replacing the Newton formula with

```
bnew=beta-fn./dfdb;
if abs(bnew-beta)<1e-13*abs(beta)
  break
end
beta=bnew;
```

We find more efficient possibilities later.

3.3 Direct solution of linear systems of equations

Solving linear systems of equations not only appear in the context of Newton's method, but are fundamental to many scientific tasks in almost all applications. There is a vast array of numerical methods for solving these systems and the choice of method depends on the properties of the coefficient matrix and the number of unknown variables. The focus here is not expressly concerned with the intimate details of methods, but more on their application and associated pitfalls.

Specific Objectives

- To understand the relevance and importance of matrix condition numbers in the context of computed solutions to linear systems of equations

3.3.1 Errors in computed solutions, norms and condition numbers

To start this section, consider the following introductory problem.

Example 3.15: Solve the linear system of equations

$$x + 2y = 5,$$

$$x + 3y = 4.$$

Find the solution by solving the matrix equation $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, where the matrix of coefficients

$$\boldsymbol{A} = \begin{bmatrix} 1 & 2 \\ 1 & 3 \end{bmatrix},$$

$\boldsymbol{x} = (x, y)$ denotes the vector of unknowns, and $\boldsymbol{b} = (5, 4)$ is the right hand side vector. Using MATLAB, find the solution using

```
A=[1 2 ; 1 3]; b=[5 ; 4]; x=A\b
```

The solution is just $x = 7$, $y = -1$. If due to some computer error (like round off) or observational error (due to instrument limitations), one of the values gets changed a little, for example the last coefficient in $\boldsymbol{A}$ becomes 3.001 instead, the solution of the system (as you should check) is now $x = 6.998$, $y = -0.999$. The small change in x and y is good as it means the solution is not sensitive to errors in $\boldsymbol{A}$—remember that small errors are inevitable.

This example is a well-conditioned system, so that any small change in the parameters (or the constants of the equations) does not change the solution much.

Example 3.16: In contrast, consider now the linear system

$$\begin{aligned} x + 2y &= 5, \\ 2x + 4.01y &= 10. \end{aligned}$$

The solution is $x = 5$, $y = 0$. Now suppose that the element 10 is modified slightly to 10.01 (for example, by round off), the solution now is $x = 3$, $y = 1$.

A very small change in one of the parameters causes a great change in the solution. This is an example of the bad behaviour of an ill-conditioned linear system.

To track down the cause of the problem, we look at the coefficient matrix and its inverse. For the first of the above examples we have

$$\boldsymbol{A} = \begin{bmatrix} 1 & 2 \\ 1 & 3 \end{bmatrix}$$

and

$$\boldsymbol{A}^{-1} = \begin{bmatrix} 3 & -2 \\ -1 & 1 \end{bmatrix}.$$

The elements of each of these two matrices are all of reasonable size. But for the second problem we have

$$\boldsymbol{A} = \begin{bmatrix} 1 & 2 \\ 2 & 4.01 \end{bmatrix},$$

and

$$\boldsymbol{A}^{-1} = \begin{bmatrix} 401 & -200 \\ -200 & 100 \end{bmatrix}.$$

These very large numbers in the inverse are an indicator that all is not well with the system. The second system is close to singular —you can easily check that the

determinant of the coefficient matrix is 0.01 (a small value for a determinant). Such a small determinant is also a strong indicator of difficulties.

However, the *condition number* is a better indication of bad behaviour. In MATLAB compute a condition number of a matrix via the function `cond()`: in the first example `cond(A)` = 14.93 which is small enough to be acceptable; whereas in the second example `cond([1,2;2,4.01])` = 2508 which is large and signals bad behaviour. Let us proceed to find out how this condition number arises, and what it means.

Exercise 3.17: We seek to solve the system of equations $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ where

$$\boldsymbol{A} = \begin{bmatrix} 3.021 & 2.714 & 6.913 \\ 1.031 & -4.273 & 1.121 \\ 5.084 & -5.832 & 9.155 \end{bmatrix} \quad \text{and} \quad \boldsymbol{b} = \begin{bmatrix} 12.648 \\ -2.121 \\ 8.407 \end{bmatrix}.$$

Verify that $\boldsymbol{x} = (1, 1, 1)$ is its solution. Now modify the matrix $\boldsymbol{A}$ by a tiny amount, change its a_{22} element by just 0.002 with the MATLAB statement `a(2,2)=-4.275`. Solve the new system and observe that the new solution is alarmingly different to the original solution $(1, 1, 1)$. Verify that the condition number of the matrix $\boldsymbol{A}$ is very large (in the tens of thousands) which flags the awfully exquisite sensitivity of the solution $\boldsymbol{x}$.

Computers are approximate When a solution to the system of linear equations $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ is calculated on a computer the answer always *must* be an approximate solution due to roundoff errors. There is an error in the computed solution and we need to quantify this somehow; a difficult task since the exact solution $\boldsymbol{x}$ is of course unknown. Let $\boldsymbol{y}$ be the computed solution so that

$$\boldsymbol{A}\boldsymbol{y} \approx \boldsymbol{b}$$

and the error $\boldsymbol{e}$ (also unknown) is

$$\boldsymbol{e} = \boldsymbol{y} - \boldsymbol{x}\,.$$

Ideally we want to make the error zero. One quantity that *is* known is the *residual* [3]

$$\boldsymbol{r} = \boldsymbol{A}\boldsymbol{y} - \boldsymbol{b} = \boldsymbol{A}\boldsymbol{y} - \boldsymbol{A}\boldsymbol{x} = \boldsymbol{A}(\boldsymbol{y} - \boldsymbol{x}) = \boldsymbol{A}\boldsymbol{e}\,,$$

which measures how well the computed solution $\boldsymbol{y}$ satisfies the equations. For an accurately computed solution this should be small in some sense, and if $\boldsymbol{y} = \boldsymbol{x}$ it will be zero. Unfortunately, the 'size' of the residual and the corresponding 'size' of the error are often markedly dissimilar, so that the former cannot be used as a reliable indicator of the latter. The relation between these two quantities depends on the 'size' of the coefficient matrix $\boldsymbol{A}$, and its inverse; the 'size' of a matrix or vector is measured using *norms*.

A norm A norm is a single real number for measuring the overall size (length) of a particular matrix or vector.

[3] The residual is directly equivalent to the function solved in Newton's method: the analogy is that $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}$.

For vectors The *usual* norm for vectors is the usual length of a vector; that is, the norm of a vector is the square root of the sum of squares of the vector elements. In symbols, the norm, denoted by $\|\cdot\|$, of a vector $\boldsymbol{v}$ is the number

$$\|\boldsymbol{v}\| = \sqrt{\sum_{i=1}^{n} v_i^2}\,.$$

For matrices In order to illustrate a measure for an ill-conditioned system, the norm we choose to use the *maximum absolute row sum.*[4] For this norm you simply add up the absolute values of the elements in each row and then take the maximum of these row sums. So the norm $\|\boldsymbol{A}\|$ of the matrix

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

is computed as $\max[|a_{11}| + |a_{12}|, |a_{21}| + |a_{22}|]$. For the first of the above examples we have $\|\boldsymbol{A}\| = \max[(1+2), (1+3)] = 4$, and $\|\boldsymbol{A}^{-1}\| = \max[(3+|-2|), (|-1|+1)] = 5$. Note that in this case the product of the norms $\|\boldsymbol{A}\|\,\|\boldsymbol{A}^{-1}\| = 20$ which is acceptable. But for the second of the above examples we have $\|\boldsymbol{A}\| = \max[(1+2), (2+4.01)] = 6.01$, and $\|\boldsymbol{A}^{-1}\| = \max[(401+|-200|), (|-200|+100)] = 601$. Note that in this case where the solution is sensitive the product of the norms $\|\boldsymbol{A}\|\,\|\boldsymbol{A}^{-1}\| = 3612$ is rather large. The product of norms $\|\boldsymbol{A}\|\,\|\boldsymbol{A}^{-1}\|$ is large for ill-conditioned systems.

The condition number An all important quantity just emerged, namely the *condition number* of the matrix $\boldsymbol{A}$,

$$\text{cond}(\boldsymbol{A}) = \|\boldsymbol{A}\|\,\|\boldsymbol{A}^{-1}\|\,,$$

which is always at least 1. Although the particular value depends on the particular norm used the following general statements are true for all norms. If the condition number is less than roughly 100, then the errors caused by roundoff will be very small. Start to get concerned about the accuracy of solutions if $\text{cond}(\boldsymbol{A})$ gets over 100. If the condition number is less than roughly 100, the solution will be resilient to roundoff inaccuracies. Between 100 and 1000 you should start to get concerned. If significantly above this, the solution will be very susceptible to inaccuracies. Further investigations and further methods would then need to be applied to the problem of finding the solution. MATLAB calculates condition numbers with the function `cond()`. It uses a different norm to the one used above, but you will find that the condition numbers calculated will be the same order of magnitude as the result using the maximum absolute row sum. For the ill-conditioned matrix above MATLAB gives condition number 2508 which is large enough to ring the same alarm bells as 3612.

How does this condition number arise? The condition number appears in a relation between the *relative error* in the solution and the size of the *residual* relative to the right-hand side vector $\boldsymbol{b}$. Let us see this. Also recall that for the exact solution $\boldsymbol{x}$, $\boldsymbol{b} = \boldsymbol{A}\boldsymbol{x}$ and taking norms indicates

$$\|\boldsymbol{b}\| = \|\boldsymbol{A}\boldsymbol{x}\| \le \|\boldsymbol{A}\| \cdot \|\boldsymbol{x}\|\,,$$

[4] Other choices are possible and similarly useful, but this one is convenient for us.

where the inequality arises from a basic property of norms (that we do not justify here) that the norm of a product is always less than or equal the product of the norms. Since the left-hand side is less than the right-hand sides, the *reciprocal* of the left must be greater than the right:

$$\frac{1}{\|\boldsymbol{b}\|} \geq \frac{1}{\|\boldsymbol{A}\| \cdot \|\boldsymbol{x}\|};$$

rewrite this as

$$\frac{1}{\|\boldsymbol{A}\| \cdot \|\boldsymbol{x}\|} \leq \frac{1}{\|\boldsymbol{b}\|}.$$

Also recall that $\boldsymbol{A}\boldsymbol{e} = \boldsymbol{r}$. Consequently, $\boldsymbol{e} = \boldsymbol{A}^{-1}\boldsymbol{r}$ and taking norms gives

$$\|\boldsymbol{e}\| = \|\boldsymbol{A}^{-1}\boldsymbol{r}\| \leq \|\boldsymbol{A}^{-1}\| \cdot \|\boldsymbol{r}\|,$$

using the same inequality property of norms. In these last two equations, the two left-hand sides are respectively smaller than the right-hand sides; hence the product of the two left-hand sides must be smaller than the product of the two right-hand sides:

$$\frac{1}{\|\boldsymbol{A}\| \cdot \|\boldsymbol{x}\|}\|\boldsymbol{e}\| \leq \frac{1}{\|\boldsymbol{b}\|}\|\boldsymbol{A}^{-1}\| \cdot \|\boldsymbol{r}\|.$$

Rearrange to

$$\frac{\|\boldsymbol{e}\|}{\|\boldsymbol{x}\|} \leq \|\boldsymbol{A}\| \cdot \|\boldsymbol{A}^{-1}\|\frac{\|\boldsymbol{r}\|}{\|\boldsymbol{b}\|};$$

see the condition number $\|\boldsymbol{A}\| \cdot \|\boldsymbol{A}^{-1}\|$ appear on the right-hand side. This inequality says that the *relative error* in a solution, $\|\boldsymbol{e}\|/\|\boldsymbol{x}\|$ is bounded by the the condition number, $\|\boldsymbol{A}\| \cdot \|\boldsymbol{A}^{-1}\|$, times the size of the residual relative to the the right-hand side $\|\boldsymbol{r}\|/\|\boldsymbol{b}\|$,. A large condition number indicates ill-conditioning because it indicates that a small residual does *not* correspond to a small error!

Summary An important manifestation of ill-conditioning is a high sensitivity of solutions to small changes in the coefficient matrix $\boldsymbol{A}$ and/or right hand side vector $\boldsymbol{b}$. That is, small changes in $\boldsymbol{A}$ or $\boldsymbol{b}$ can induce large changes in the solution vector. These small changes could well arise from rounding or error in measured or calculated data values. The condition number gives us a measure of how well to trust that our solution is accurate.

3.4 Iterative methods for linear systems of equations

In many applications the coefficient matrix $\boldsymbol{A}$ contains a very high proportion of zero elements[5], so storing the entire matrix and applying a direct solution method is extremely wasteful. In other applications involving huge numbers of unknowns, storage and efficiency considerations effectively rule out direct methods altogether. The answer in both these cases is to use iterative methods.

[5] A sparse matrix

Specific Objectives

- To understand when and why iteration methods are sometime superior for solving linear systems of equations
- To apply Jacobi iteration for solving diagonally dominant linear systems

3.4.1 Jacobi iteration solves large systems

Jacobi iteration proceeds by rewriting each equation with the diagonal element on the left hand side. A guess is updated by simply substituting it into the right hand side of each equation, returning a new set of values.

Example 3.18: Use Jacobi iteration to solve

$$\begin{aligned} 10x_1 - x_2 + 2x_3 &= 6\,, \\ -x_1 + 11x_2 - x_3 + 3x_4 &= 25\,, \\ 2x_1 - x_2 + 10x_3 - x_4 &= -11\,, \\ 3x_2 - x_3 + 8x_4 &= 15\,, \end{aligned}$$

that is, $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ where

$$\boldsymbol{A} = \begin{bmatrix} 10 & -1 & 2 & 0 \\ -1 & 11 & -1 & 3 \\ 2 & -1 & 10 & -1 \\ 0 & 3 & -1 & 8 \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} 6 \\ 25 \\ -11 \\ 15 \end{bmatrix}.$$

Note that the first equation is $10x_1 - x_2 + 2x_3 = 6$, which we rearrange in terms of x_1. The second equation is $-x_1 + 11x_2 - x_3 + 3x_4 = 25$ which we rearrange in terms of x_2 and so on. Thus the equations are rewritten as

$$\begin{aligned} x_1 &= \frac{1}{10}\left[6 + x_2 - 2x_3\right] \\ x_2 &= \frac{1}{11}\left[25 + x_1 + x_3 - 3x_4\right] \\ x_3 &= \frac{1}{10}\left[-11 - 2x_1 + x_2 + x_4\right] \\ x_4 &= \frac{1}{8}\left[15 - 3x_2 + x_3\right] \end{aligned}$$

With a starting guess

$$\boldsymbol{x}^{(0)} = (0, 0, 0, 0, 0),$$

the first Jacobi iteration on this system yields

$$\begin{aligned} x_1^{(1)} &= \frac{1}{10}[6] = 0.6, \\ x_2^{(1)} &= \frac{1}{11}[25] = 2.2727, \\ x_3^{(1)} &= \frac{1}{10}[-11] = -1.1, \\ x_4^{(1)} &= \frac{1}{8}[15] = 1.875 \end{aligned}$$

and for the second iteration these values are simply substituted into the right hand side again, giving

$$\begin{aligned} x_1^{(2)} &= \frac{1}{10}\left[6 + x_2^{(1)} - 2x_3^{(1)}\right] = 1.0473 \\ x_2^{(2)} &= \frac{1}{11}\left[25 + x_1^{(1)} + x_3^{(1)} - 3x_4^{(1)}\right] = 1.7159 \\ x_3^{(2)} &= \frac{1}{10}\left[-11 - 2x_1^{(1)} + x_2^{(1)} + x_4^{(1)}\right] = -0.8052 \\ x_4^{(2)} &= \frac{1}{8}\left[15 - 3x_2^{(1)} + x_3^{(1)}\right] = 0.8852. \end{aligned}$$

The little MATLAB script in Algorithm 12 computes more Jacobi iterations. After six iterations the approximate solution is

$$\boldsymbol{x}^{(6)} = (1.0032, 1.9922, -0.9945, 0.9944)\,.$$

Further iterations converge to the exact solution $\boldsymbol{x} = (1, 2, -1, 1)$.

Algorithm 12 simple Jacobi iteration

```
C=[0 -1  2  0
  -1  0 -1  3
   2 -1  0 -1
0  3 -1  0]
b=[6;25;-11;15]
d=[10;11;10;8]
x=zeros(4,1)
for k=1:20, x=(b-C*x)./d, end
```

Exercise 3.19: In the previous example, other starting guesses could have been used and the process will still converge. Try the Jacobi iteration starting with $\boldsymbol{x}$ values $(2, -1, 2, -1)$. You will see that the process still converges to the same solution $\boldsymbol{x} = (1, 2, -1, 1)$.

In matrix form the coefficient matrix $\boldsymbol{A}$ is split into two parts, $\boldsymbol{D}$ and $\boldsymbol{C}$, the diagonal and the off-diagonal entries respectively: then $\boldsymbol{A} = \boldsymbol{D} + \boldsymbol{C}$. For example, as in Algorithm 12,

$$\boldsymbol{C} = \begin{bmatrix} 0 & -1 & 2 & 0 \\ -1 & 0 & -1 & 3 \\ 2 & -1 & 0 & -1 \\ 0 & 3 & -1 & 0 \end{bmatrix} \quad \text{and} \quad \boldsymbol{D} = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 11 & 0 & 0 \\ 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 8 \end{bmatrix}.$$

Then the equation $\boldsymbol{Ax} = \boldsymbol{b}$ becomes $(\boldsymbol{D}+\boldsymbol{C})\boldsymbol{x} = \boldsymbol{b}$, which rearranged is $\boldsymbol{Dx} = \boldsymbol{b} - \boldsymbol{Cx}$, that is $\boldsymbol{x} = \boldsymbol{D}^{-1}(\boldsymbol{b} - \boldsymbol{Cx})$. This suggests the iteration scheme $\boldsymbol{x}^{(k+1)} = \boldsymbol{D}^{-1}(\boldsymbol{b} - \boldsymbol{Cx}^{(k)})$. But we prefer to compute $\boldsymbol{Ax}$ rather than $\boldsymbol{Cx}$ so substitute $\boldsymbol{C} = \boldsymbol{A} - \boldsymbol{D}$ to find the right-hand side $\boldsymbol{D}^{-1}(\boldsymbol{b} - \boldsymbol{Cx}^{(k)}) = \boldsymbol{D}^{-1}(\boldsymbol{b} - \boldsymbol{Ax}^{(k)} + \boldsymbol{Dx}^{(k)}) = \boldsymbol{x}^{(k)} - \boldsymbol{D}^{-1}(\boldsymbol{Ax}^{(k)} - \boldsymbol{b})$. Thus we write the iteration scheme as

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \boldsymbol{D}^{-1}(\boldsymbol{Ax}^{(k)} - \boldsymbol{b})\,;$$

that is, corrections are guided by the residuals $\boldsymbol{r}^{(k)} = \boldsymbol{A}\boldsymbol{x}^{(k)} - \boldsymbol{b}$. Note: the inverse of $\boldsymbol{D}$ is simply the diagonal matrix matrix with the reciprocals of $\boldsymbol{D}$'s diagonal elements down the main diagonal; thus the mathematical operation of multiplying by the inverse $\boldsymbol{D}^{-1}$ is computed simply by dividing component-wise by the vector $\boldsymbol{d} = \operatorname{diag} \boldsymbol{A}$ as in Algorithm 12 where we divide component-wise by $\boldsymbol{d} = (10, 11, 10, 8)$ with the operation `./d`. Thus at every step of the Jacobi technique the approximate solution is modified by subtracting from it the residual vector $\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}$ —the difference between what $\boldsymbol{A}\boldsymbol{x}$ is and what it should be—with each entry divided by the corresponding diagonal entry of $\boldsymbol{A}$. A suitable rule for stopping the iteration might be ensure that every entry of the residual is smaller than some given value (the *tolerance*).

Algorithm 13 specifies a function to execute Jacobi iteration in general, and uses native matrix-vector operations in MATLAB. Being built upon the BLAS and LAPACK libraries discussed briefly in Section 3.4.2, these computations are very fast and efficient.

In using Jacobi iteration we might like the function to also deliver the number of iterations that it took to come to its final output, see Algorithm 13. This algorithm implements a `for` loop to provide a 'safety net' in case the iterations do not converge.

Algorithm 13 The inputs to a function that will execute the Jacobi iteration and output the final approximate solution to a particular tolerance level will be the coefficient matrix $\boldsymbol{A}$, the right-hand side vector $\boldsymbol{b}$, an initial approximate solution $\boldsymbol{x}$ and the tolerance T. Count the number of iterations.

```
%  Function: Jacobi
%  Syntax:  [x,n] = Jacobi(A,b,x0)
%  Inputs :
%   A, matrix of coefficients is assumed to be square
%   b, right-hand side vector
%   x0, initial approximation to the solution
%  Output:
%   x, the approximate solution vector
%   n, the number of iterations
%  Algorithm: performs Jacobi iteration to find
%      an approximate solution for the equations Ax=b
%      for which the largest absolute value of any
%      residual in b-Ax is less that 10^(-12)
%
    function [x,n] = Jacobi(A,b,x0)
    x=x0;
    for n=1:999
      r = A*x - b ;
      x = x - r./diag(A) ;
      if norm(r)<1e-12, break, end;
    end
```

In use note that

- if the output is not assigned to a variable then only the $\boldsymbol{x}$ value is returned as in

```
Jacobi(A,b,zeros(4,1))
```

- if the output is assigned to a single variable then it is $\boldsymbol{x}$ that is returned as in

```
x=Jacobi(A,b,zeros(4,1));
```

- and n is only returned when the output is assigned to two variables as in

```
[x,n]=Jacobi(A,b,zeros(4,1));
```

Convergence? To be assured of convergence, the matrix $\boldsymbol{A}$ must satisfy the condition called *diagonal dominance*. In the previous examples the diagonal elements in each row of $\boldsymbol{A}$ were larger in magnitude than the sum of the absolute off diagonal elements, a property known as diagonal dominance

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|, \quad \text{for all } i = 1, \ldots, n\,.$$

When the matrix is diagonally dominant, then convergence is guaranteed from any starting guess. If the matrix is not diagonally dominant, you can sometimes be lucky and get convergence, but success is not assured.

3.4.2 Memory management

Vectorising over parametric studies greatly increases speed. Using simple matrix-vector operations in Jacobi iterations is also fast. However, memory is not simply homogeneous 'random access' as its name implies. In high performance workstations and supercomputers, memory is structured, and we have to work with its structure for maximum speed. For example, memory is not just the main RAM but also is the cache (usually several layers of cache).

Memory allocation Memory within a computer can be viewed as one long 'street' of addresses. At each address one number is stored. By manipulating which numbers we store and access in this 'street', it is possible to make more effective use of memory. To illustrate, how this works consider the following example.

Consider the process of accessing elements of the following matrix:

$$\boldsymbol{a} = \begin{bmatrix} 1 & 7 & 13 & 19 \\ 2 & 8 & 14 & 20 \\ 3 & 9 & 15 & 21 \\ 4 & 10 & 16 & 22 \\ 5 & 11 & 17 & 23 \\ 6 & 12 & 18 & 24 \end{bmatrix}$$

As computer memory is stored in consecutive addresses, as a vector rather than a matrix, these elements are stored as follows.

1	2	3	4	5	6	7	8	···	24

Now we request the first element ($a_{1,1}$) from memory. The CPU finds that this element is not in the (fast) cache, so it needs to get it from (slow) RAM. When the CPU makes a request to RAM it does not just grab a single element, rather it grabs a whole chunk of elements, sequentially from the one requested. The actual number of elements grabbed depends on the memory page size. If the matrix is small enough, then such a grab may bring the whole matrix into cache. However, in most applications only part of matrix will be brought into cache. To illustrate the effect that this has on performance we suppose that when we request the first element of column one, only the whole column is brought into cache. Thus, when the CPU requested the first element, the following numbers are now stored in cache (they form one 'line' of cache).

Number stored in cache

1	2	3	4	5	6

If the next element required is $a_{2,1}$ the CPU gets it from the fast cache. However, if the next element required is $a_{1,2}$ (from the second column), the CPU has to get it from slow RAM. This implies that another column of data has to be loaded into a line of cache and there is a major additional overhead in getting the data from memory. While this overhead may seem small, it becomes extremely import when processing large arrays. Thus, structure memory access in such a way to make use of cache.

BLAS and LAPACK empower efficient computations Fortunately, efficient matrix-vector computation is so important that excellent software exists to make efficient use of cache and the vector/superscalar nature of modern computers. The software is called the BLAS and LAPACK libraries. MATLAB use these efficient libraries when you code clean matrix-vector operations.

I quote some extracts from the web.

> The BLAS (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations. The Level 1 BLAS perform scalar, vector and vector-vector operations, the Level 2 BLAS perform matrix-vector operations, and the Level 3 BLAS perform matrix-matrix operations. Because the BLAS are efficient, portable, and widely available, they are commonly used in the development of high quality linear algebra software,
> http://www.netlib.org/blas/faq.html

> LAPACK ... provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. The associated matrix factorisations (LU, Cholesky, QR, SVD, Schur, generalised Schur) are

> also provided, as are related computations such as reordering of the Schur factorisations and estimating condition numbers. ... The original goal of the LAPACK project was to ... run efficiently on shared-memory vector and parallel processors. On these machines, LINPACK and EISPACK are inefficient because their memory access patterns disregard the multi-layered memory hierarchies of the machines, thereby spending too much time moving data instead of doing useful floating-point operations. LAPACK addresses this problem by reorganising the algorithms to use block matrix operations, such as matrix multiplication, in the innermost loops. These block operations can be optimised for each architecture to account for the memory hierarchy, and so provide a transportable way to achieve high efficiency on diverse modern machines. http://www.netlib.org/lapack/

Use this software to make good use of modern computer architecture without needing intricate details of the computer.

3.5 Iterative methods for systems of non-linear equations

Specific Objectives

- To apply the two and three dimensional form of Newton's method for solving systems of non-linear equations

3.5.1 Newton's method for non-linear systems of equations

The simplest type of non-linear *system* consists of a pair of equations

$$f(x, y) = 0\,, \quad g(x, y) = 0\,,$$

in two unknowns x and y. For example, suppose you have been asked to solve the pair of non-linear equations

$$x^2 - xy^2 - xy = 1\,, \quad y^2 + x^3 + xy = 3\,.$$

In this case $f(x, y) = x^2 - xy^2 - xy - 1$ and $g(x, y) = y^2 + x^3 + xy - 3$. These are non-linear because the involve products of the unknowns. Each equation represents a curve in the xy-plane and one interpretation of the task is to find intersection point(s) for the curves. There are actually four real solutions to this system of two equations, one in each quadrant of the xy-plane.

Example 3.20: a Newton's step with two variables Suppose we wish to solve

$$f(x, y) = x^2 - xy^2 - xy - 1 = 0\,, \quad g(x, y) = y^2 + x^3 + xy - 3 = 0\,.$$

We have no idea where solutions might be. Let us guess there might be one near $(x, y) = (1, 0)$ for which $f(1, 0) = 0$ and $g(1, 0) = -2$. (Although $f = 0$ may be viewed as being a partial solution, $f = 0$ is not significant as you must treat the problem as a system. Since $g \neq 0$ means we have not satisfied the equations as a pair.) How can we take a step from $(x, y) = (1, 0)$ to improve this guess?

Answer: Newton tells us. Suppose we take a supposedly *small* step of (h, v) away from $(1, 0)$ to a nearby location $(x, y) = (1 + h, v)$. Then

$$\begin{aligned} f(x, y) &= (1 + h)^2 - (1 + h)v^2 - (1 + h)v - 1 \\ &= 1 + 2h + h^2 - v^2 - hv^2 - v - hv - 1 \\ &= 0 + 2h - v + \cdots , \\ g(x, y) &= v^2 + (1 + h)^3 + (1 + h)v - 3 \\ &= v^2 + 1 + 3h + 3h^2 + h^3 + v + hv - 3 \\ &= -2 + 3h + v + \cdots , \end{aligned}$$

where the ellipses, "$\cdots$", denote powers and products of h and v that are ignored in Newton's linear approximation. That is, in matrix-vector notation

$$\begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} 0 \\ -2 \end{bmatrix} + \begin{bmatrix} 2 & -1 \\ 3 & 1 \end{bmatrix} \begin{bmatrix} h \\ v \end{bmatrix} + \cdots .$$

Assuming this linear approximation holds good, we estimate where $(f, g) = \mathbf{0}$ by solving for when the linear parts of the right-hand side are zero. That is, $(h, v) = (2/5, 4/5)$. Thus a new guess for the solution of $(f, g) = \mathbf{0}$ is $(x_1, y_1) = (7/5, 4/5)$.

This previous example shows how we may take a step from one guess of a solution to another guess, (x_1, y_1). Similar arguments empower us to take more steps.

Example 3.21: more Newton's steps The aim is to solve

$$f(x, y) = x^2 - xy^2 - xy - 1 = 0 , \quad g(x, y) = y^2 + x^3 + xy - 3 = 0 .$$

Suppose we estimate a solution to $(f, g) = (0, 0)$ is the point $(x, y) = (x_1, y_1)$. What step from (x_1, y_1) to a nearby point $(x_1 + h, y_1 + v)$ will improve the estimate of the solution?

Solution: At the point (x_1, y_1) the function

$$f = (x_1^2 - x_1 y_1^2 - x_1 y_1 - 1) + (2x_1 - y_1^2 - y_1)h + (-2x_1 y_1 - x_1)v + \cdots ,$$

where the $\cdots$ denote small terms in products of the small h and v. Similarly, the function

$$g = (x_1^3 - x_1 y_1 + y_1^2 - 3) + (3x_1^2 + y_1)h + (x_1 + 2y_1)v + \cdots .$$

For example, at $(x_1, y_1) = (7/5, 4/5)$

$$\begin{aligned} f &= -1.056 + 1.36\, h - 3.64\, v + \cdots , \\ g &= 1.504 + 6.68\, h + 3\, v + \cdots . \end{aligned}$$

That is, in matrix-vector notation

$$\begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} -1.056 \\ 1.504 \end{bmatrix} + \begin{bmatrix} 1.36 & -3.64 \\ 6.68 & 3 \end{bmatrix} \begin{bmatrix} h \\ v \end{bmatrix} + \cdots .$$

Similarly rewrite the general expressions in matrix-vector form:

$$\begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} f_1 \\ g_1 \end{bmatrix} + \underbrace{\begin{bmatrix} 2x_1 - y_1^2 - y_1 & -2x_1y_1 - x_1 \\ 3x_1^2 + y_1 & x_1 + 2y_1 \end{bmatrix}}_{J} \begin{bmatrix} h \\ v \end{bmatrix} + \cdots ,$$

where $f_1 = f(x_1, y_1)$ and $g_1 = g(x_1, y_1)$. The 2×2 matrix $\boldsymbol{J}$ is called the *Jacobian*. Again, *assuming* the displayed linear parts are good enough, a better estimate of the solution $(f, g) = \mathbf{0}$ is obtained by putting $(f, g) = \mathbf{0}$ into this equation and solving for $(h, v) = -\boldsymbol{J}^{-1}(f_1, g_1)$. For example, from $(x_1, y_1) = (7/5, 4/5)$ we compute $(h, v) = (-.0812, -.3205)$ to make the next estimate of the solution $(x_2, y_2) = (1.3187, 0.4795)$.

Algorithm 14 shows that successive Newton's steps converges quickly to the solution $(x, y) = (1.3212, 0.4024)$. Use this algorithm to find the other four real solutions by starting from different initial guesses.

Algorithm 14 script to solve a pair of non-linear equations.

```
% Newton solves a pair of coupled non-linear equations
% Tony Roberts, March 2007
% Modified Harry Butler, Jan 2017
    xy=[1;0]
    for k=1:6
      x=xy(1); y=xy(2);
      fg=[x^2-x*y^2-x*y-1
      y^2+x^3+x*y-3]
      J=[2*x-y^2-y -2*x*y-x
        3*x^2+y     2*y+x];
        xy=xy-J\fg
    end
```

Given any approximation to a solution, (x_k, y_k), the procedure is analogous to the single variable case.

Generalise the one variable Newton's method

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)},$$

as it can be written as

$$x_{k+1} = x_k - \big[f'(x_k)\big]^{-1} f(x_k).$$

Also recall Jacobi iteration has the similar form

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \boldsymbol{D}^{-1}\boldsymbol{r}^{(k)}$$

where we seek to zero the residual $\boldsymbol{r}^{(k)} = (\boldsymbol{A}\boldsymbol{x}^{(k)} - \boldsymbol{b})$. The two dimensional form of Newton's method is an extension of these above forms:

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ y_k \end{bmatrix} - \begin{bmatrix} \frac{\partial f}{\partial x}(x_k, y_k) & \frac{\partial f}{\partial y}(x_k, y_k) \\ \frac{\partial g}{\partial x}(x_k, y_k) & \frac{\partial g}{\partial y}(x_k, y_k) \end{bmatrix}^{-1} \begin{bmatrix} f(x_k, y_k) \\ g(x_k, y_k) \end{bmatrix},$$

where $\frac{\partial f}{\partial x}(x_k, y_k)$ is the *partial* derivative of $f(x, y)$ with respect to x evaluated at (x_k, y_k).

For example, if $f(x, y) = x^2y - y^2x - 1$, then $\frac{\partial f}{\partial x}(x, y) = 2xy - y^2$ (that is, we hold y constant), and $\frac{\partial f}{\partial y}(x, y) = x^2 - 2yx$ (that is we hold x constant).

The matrix of partial derivatives appearing on the right hand side is called the Jacobian matrix:

$$\boldsymbol{J} = \begin{bmatrix} \frac{\partial f}{\partial x}(x_k, y_k) & \frac{\partial f}{\partial y}(x_k, y_k) \\ \frac{\partial g}{\partial x}(x_k, y_k) & \frac{\partial g}{\partial y}(x_k, y_k) \end{bmatrix}.$$

Notionally we refer to its *inverse* which is the generalisation of the *reciprocal* used in the one dimensional case. Proof of this two dimensional iteration formula awaits your further mathematics study.

Example 3.22: Compute two steps of Newton's method to solve the pair of equations

$$f(x, y) = x^2 + y^2 - 2 = 0\,, \quad g(x, y) = y - \cos x = 0\,,$$

given they have a solution near $(x_0, y_0) = (1, 0)$.

Solution: The Jacobian matrix is

$$\boldsymbol{J} = \begin{bmatrix} \frac{\partial f}{\partial x}(x, y) & \frac{\partial f}{\partial y}(x, y) \\ \frac{\partial g}{\partial x}(x, y) & \frac{\partial g}{\partial y}(x, y) \end{bmatrix} = \begin{bmatrix} 2x & 2y \\ \sin x & 1 \end{bmatrix}$$

and so the first step of Newton's method becomes

$$\begin{aligned} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} &= \begin{bmatrix} 1 \\ 0 \end{bmatrix} - \begin{bmatrix} 2 & 0 \\ 0.8415 & 1 \end{bmatrix}^{-1} \begin{bmatrix} -1 \\ -0.5403 \end{bmatrix} \\ &= \begin{bmatrix} 1 \\ 0 \end{bmatrix} - \begin{bmatrix} 0.5 & 0 \\ -0.4208 & 1 \end{bmatrix} \begin{bmatrix} -1 \\ -0.5403 \end{bmatrix}, \end{aligned}$$

since $f(1, 0) = -1$ and $g(1, 0) = -0.5403$. MATLAB then gives $(x_1, y_1) = (1.5, 0.1196)$ as a better guess to the solution of the system of equations.

Now the function values $f(x_1, y_1) = 0.2643$, $g(x_1, y_1) = 0.0488$. The second Newton step becomes

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1.5 \\ 0.1196 \end{bmatrix} - \begin{bmatrix} 3 & 0.2392 \\ 0.9975 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0.2643 \\ 0.0488 \end{bmatrix},$$

MATLAB then gives $(x_2, y_2) = (1.4085, 0.1621)$.

Continuing the process for five iterations, an accurate solution is $(1.4045, 0.1655)$.

The iteration scheme to use for systems of pairs of equations in two unknowns is thus

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ y_k \end{bmatrix} - \begin{bmatrix} \frac{\partial f}{\partial x}(x_k, y_k) & \frac{\partial f}{\partial y}(x_k, y_k) \\ \frac{\partial g}{\partial x}(x_k, y_k) & \frac{\partial g}{\partial y}(x_k, y_k) \end{bmatrix}^{-1} \begin{bmatrix} f(x_k, y_k) \\ g(x_k, y_k) \end{bmatrix},$$

For a general system of many non-linear equations, $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0}$, in many unknowns, the iteration is

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \boldsymbol{J}_k^{-1} \boldsymbol{f}(\boldsymbol{x}_k),$$

where $\boldsymbol{J}_k$ is the $m \times m$ Jacobian matrix evaluated at $\boldsymbol{x}_k$. Note that although the MATLAB function `inv()` could be used to find `inv(J)`, it is far, far better to use the \ operator to solve equations.

Start with some initial guesses for the values of x_0 and y_0, (that is, the initial vector (x_0, y_0)). Iterations continue until at some iteration either

- the displacement is sufficiently small,

$$\max[|x_{k+1} - x_k|, |y_{k+1} - y_k|] < \epsilon,$$

- or the function values are sufficiently small,

$$\max[|f(x_k, y_k)|, |g(x_k, y_k)|] < \epsilon,$$

- or the sizes are small in *relative* magnitude instead of the absolute magnitude shown above,

where ϵ is chosen to be small (for example, $\epsilon = 10^{-6}$).

Exercise 3.23: Check the answer for the above example using MATLAB. Note that when J is the matrix to be inverted, avoid inversion with the MATLAB command

```
x=[1;0]-J\[-1;-0.5403]
```

Large systems As the number m of equations and variables increases, the number of derivatives required for the Jacobian, m^2, rapidly becomes unmanageable for hand evaluation and programming. As in the single variable case finite difference approximations can be used to give adequate approximations. For example,

$$\frac{\partial f}{\partial x}(x_k, y_k) \approx \frac{f(x_k + h, y_k) - f(x_k, y_k)}{h}$$

in the two variable case. To illustrate further, if we had $f(x, y) = x^2 y + y^2 x + 4$ and $h = 0.0001$, then

$$\frac{\partial f}{\partial x}(2, 1) \approx \frac{f(2 + h, 1) - f(2, 1)}{h}$$

$$= \frac{(2.0001^2 \times 1 + 1^2 \times 2 + 4) - (2^2 \times 1 + 1^2 \times 2 + 4)}{0.0001}$$
$$= 5.0001.$$

However, you will only be required to do the method for a maximum of three non-linear equations, and equations that are straightforward to differentiate. Thus for now, just compute the Jacobians by calculus, not numerical differentiation.

Very large systems For systems with more than thousands of equations and variables, the direct solution `J\f` is impractical. Instead use Krylov subspace methods that are a sophisticated generalisation of the Jacobi iteration of Section 3.4.1.

3.6 Summary and key formulae

This module introduced iterative solutions to linear and non-linear equations.

- Iterative methods solve non-linear equations of the form $f(x) = 0$ by producing a sequence of estimates x_k which *converges* to the solution under the right conditions. One powerful method is *Newton's Method*, in which iterates are generated by the formula

  $$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}.$$

  Another important consideration for convergence in iterative methods is the choice of *starting guess*.

- For non-linear *systems* of equations, Newton's method takes the form

  $$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \boldsymbol{J}_k^{-1} \boldsymbol{f}(\boldsymbol{x}_k),$$

  where $\boldsymbol{J}$ is the *Jacobian* matrix, or matrix of first partial derivatives for the vector function $\boldsymbol{f}(\boldsymbol{x})$.

- The *condition number* of the matrix $\boldsymbol{A}$ indicates how well the *residual* of a computed solution to $\boldsymbol{Ax} = \boldsymbol{b}$ measures its accuracy, and how sensitive the solution will be to changes in matrix $\boldsymbol{A}$ and right hand side vector $\boldsymbol{b}$.

- For strictly *diagonally dominant* matrices $\boldsymbol{A}$, iterative methods such as the *Jacobi* method

  $$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \boldsymbol{D}^{-1} \left[\boldsymbol{A}\boldsymbol{x}^{(k)} - \boldsymbol{b}\right]$$

  converge to the solution from any initial guess.

- BLAS and LAPACK empower us to make effective use of cache and memory structure when we code computations in matrix-vector form in MATLAB.

Chapter 4 Interpolation fills in the unknown

Chapter contents

4.1 Introduction

Experimental measurements produce limited data which are insufficient to allow further analysis. For example, a wind tunnel test for a model wing section produces measured pressure coefficients at a limited set of points on the wing surface. This data is typically insufficient for accurate calculation of the lift of the wing. We require ways to estimate data values we do not directly know: either by fitting trends or filling in between data.

The simplest approximation is to say that in a sufficiently small neighbourhood of any particular data point the value of the dependent variables will not differ significantly from the value that it has at that particular point. Computer visualisation of data works on this simple principle, as does some data mining such as case based reasoning.

The computer screen consists of a given number of pixels in the horizontal direction and a given number of pixels in the vertical direction. We may see a smooth curve on the screen, but if we were to magnify it down to pixel level we would see that this curve is made up of small rectangular blocks. Each block is expressing the fact that over a small domain (the width of a pixel) we are approximating the values of the underlying function by some constant value (or, more precisely, by *any* of the values that the vertical breadth of the pixel represents).

Although computer plotting approximates functions with constant values, we obtain much better approximations by approximating function values in an interval with straight lines of appropriate slope. This will be possible if the slope of the function is approximately constant over the interval, or, in other words, that the function has a continuous slope over the domain of the plot. Again we will have

to be wary of functions that have corners, or cusps or have points of discontinuity and draw graphs to deal with these points correctly.

When trying to fit data we have two choices: to find a function or functions that go through the data points or to find a function that "fits" the data without requiring it to go through each data point. The first is called "Interpolation" and is normally used with data that is assumed to be accurate (data for computer graphics, geolocation data, data from computer models &c.). The second is "Curve Fitting" and is normally used when the data is known to contains errors (experimental data) and the best we can hope for is to identify trends in the data.

When interpolating the data is normally split into sections or "pieces" and a different function is found for each piece. This is called "piecewise" interpolation. Normally the form of the functions will be the same for each piece—polynomials will be the same order with different coefficients, or the same trigonometric function will be used for each piece, &c.

When curve fitting one method is *Least Squares* which finds a single curve that gives the *best* fit (in the sense of minimising differences between the data and the curve) to the given data.

In general polynomials of any degree can approximate function values over certain intervals. Once we have developed formulae for approximating polynomials we can use them to develop formulas for approximate integrals and derivatives so that the analysis of the relationship can call on all the tools of the calculus.

Polynomial approximation is not the only sort of function approximation that can be considered. If a function is periodic—and polynomials are not periodic—then using trigonometric functions, sin and cos, will produce better results. These sort of approximations lead to *Fourier Series*, see MAT3105 *Harmony of Partial Differential Equations*, whereas approximation by polynomials leads to *Power Series*. More generally, any known set of functions that are continuous over the range of the fit can be used to interpolate data. We will not pursue these questions here.

General objectives for this module

- to gain a knowledge and understanding of piecewise interpolation for data and functions;
- to apply these methods on small data sets;
- to use piecewise approximations to functions to find the values of integrals and derivatives;
- to be able to fit the 'best' lines or curves through approximate data.

Prerequisite knowledge

- Low order polynomial functions, derivatives and integrals;
- solving systems of linear algebraic equations.

4.2 Polynomial interpolation

Interpolation can be used when some property/function is known at a given set of data points and we wish a procedure for computing property/function values at intermediate points. We restrict ourselves to functions of a single variable and represent the given points, at which the function values are known, as a list vector $\boldsymbol{x}$. We will often refer to this vector as the set of *tabulation points*. We assume that this list is given in increasing order and drawn from a single interval. The values may, or may not be uniformly spaced.

Example 4.1:

- $$\texttt{xu} = 0.0,\ 0.2,\ 0.4,\ 0.6,\ 0.8,\ 1.0$$

  is a uniformly spaced list from the interval $[0, 1]$.
- Non-uniformly spaced lists also occur. For example, simply square the previous list entries:

  $$\texttt{xn} = 0.00,\ 0.04,\ 0.16,\ 0.36,\ 0.64,\ 1.0\,;$$

  that is, `xn = xu.^2`. In MATLAB the dot on `xu.^2` is essential otherwise MATLAB will attempt matrix multiplication and find that `xu` is not a matrix of the right dimensions for squaring matrices.
- Sorting a list of random numbers also produces a nonuniform list `x=sort(rand(10,1))`.

For many of our examples we will draw a sample of values from well known functions rather than present experimental data or function values from some complex process. In these cases our tabulation points will be a sample of values from the domain of a well known function, but do not infer that this is the normal situation (except for computer graphics). Genuine applications use real data.

Specific Objectives:

- to compute and plot approximate values for piecewise approximations interpolating between given tabulation points;
- to compute and plot the derivatives and integrals of these approximating functions;

4.2.1 Piecewise constant approximations

Given a set of points x and corresponding functions values $f(x)$, a piecewise constant approximation will assume that the value of $f(x)$ at every element of x is used as the function value throughout some interval around each point of x. Several choices might be made for the these intervals of constancy (Figure 4.1 shows two): perhaps from one element to the next, or from one element to the previous, or between the midpoints between the elements.

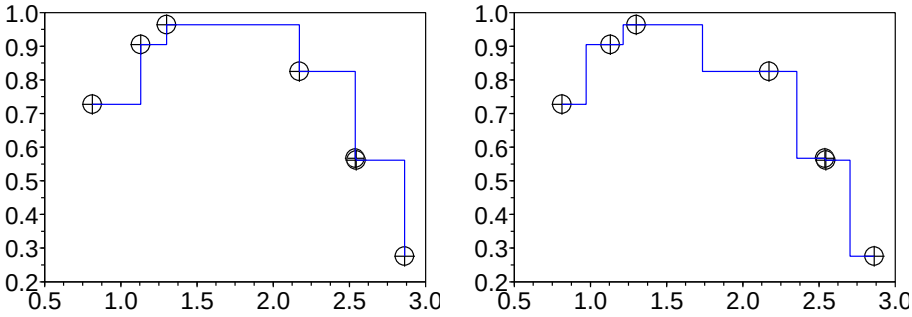


Figure 4.1: Local constant approximations: left is piecewise constant adopting the left value; right is constant between the midpoints.

This last choice is known by the grand names of "nearest neighbour fitting" or "case based reasoning" in the two fields of Data Mining and Artificial Intelligence. Data mining is a huge and growing industry in business.

We get different pictures from the graphs of these approximations.

If the function is continuous and if we have a sufficiently large number of function values available to represent it moderately faithfully, then various approximations begin to look, at least to the eye, very similar to the true function. See Figure 4.2 for example. The horizontal lines of the approximation 'disappear' as the eye follows the points of the changing function values.

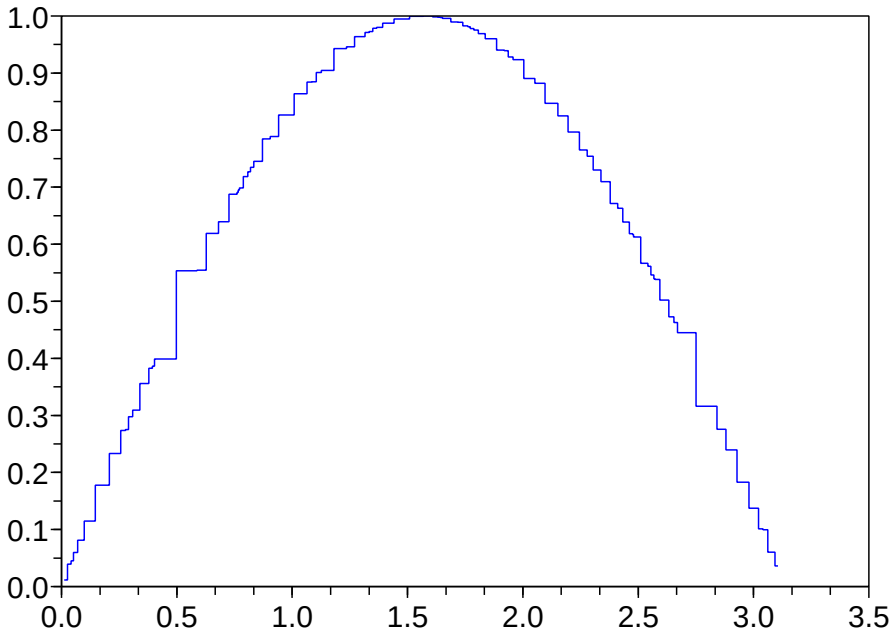


Figure 4.2: Local, moderately dense, constant approximation to $y = \sin x$.

Example 4.2: Left value constant Suppose you have six data points as in Figure 4.1 and wish to draw the piecewise constant, left value, approximation, as shown in the left graph of Figure 4.1. The six points have coordinates (x_1, y_1) to

(x_6, y_6). The left value, piecewise constant, approximation starts from (x_1, y_1), then is a horizontal straight line to (x_2, y_1), then a vertical line to (x_2, y_2), then a horizontal line to (x_3, y_2), then a vertical to (x_3, y_3), then a horizontal to (x_4, y_3), and so on to (x_6, y_6). That is, we connect eleven corner points with x coordinates whose subscripts are 1,2,2,3,3,4,4,5,5,6,6, and with y coordinates whose subscripts are correspondingly 1,1,2,2,3,3,4,4,5,5,6. In MATLAB we easily generate these subscripts.

Algorithm 15 lists MATLAB executable code for the piecewise constant approximation with left values. This code assumes that MATLAB plotting functions join the given points with straight lines. See how the index vectors `i` and `j` duplicate coordinate values so that the points with coordinates `(x(i),y(j))` enumerate the corner points of the desired plot.

Algorithm 15 piecewise constant approximation with left values: to the sine function on $0 < x < \pi$.

```
% Draw piecewise constant approximation to sine
% with randomly chosen arguments.
% Tony Roberts, 2007
% Modified Harry Butler, Jan 2017
n=10
x=sort(rand(1,n)*pi)  % generate sorted random x
y=sin(x); % compute corresponding y values
i=ceil(1:.5:n)
j=floor(1:.5:n)
plot(x(i),y(j))
```

Similar algorithms do other piecewise constant approximations. Algorithm 16 provides a function to plot midpoint values. Note how the index vectors duplicate data values to get the appropriate vertices of the plot:

- $x_1 = (x_1 + x_1)/2$, $(x_1 + x_2)/2$ twice, $(x_2 + x_3)/2$ twice, ..., $(x_{n-2} + x_{n-1})/2$ twice, $(x_{n-1} + x_n)/2$ twice, $x_n = (x_n + x_n)/2$;
- and y_1, y_1, y_2, y_2, y_3, ..., y_{n-2}, y_{n-1}, y_{n-1}, y_n, y_n, correspondingly.

Other variations will plot other piecewise constant approximations.

To be able to plot the approximations may be convenient but in practice we may need to have the actual function values at given points. Algorithm 17 lists a function to return these values. But before looking at Algorithm 17, let us explore the simpler problem of computing a value at any *one* given X.

- Given a vector of x values, stored in `x` and of length n, the vector `dd=abs(x-X)` computes the distance from X to each of the x values. For example, for $X = 1.71$ and vector

$$\begin{aligned} & \texttt{x} = (0.21, 0.62, 1.76, 2.08, 2.28) \\ \Rightarrow \quad & \texttt{dd} = (1.50, 1.09, 0.05, 0.37, 0.57)\,. \end{aligned}$$

- Then the minimum of `dd` gives the distance from X to the closest point in `x`: for example, `min(dd)` $= 0.05$.

Algorithm 16 function to plot a piecewise constant approximation to data: for any given x the value is the y-value of the nearest supplied x-point.

```
% Function :  PP0plot
% Syntax   :  z = PP0plot(x,y,c)
% Input    :  Vectors x and y  where (x,y) are data points,
%             c is the usual Matlab colour specifier
% Output   :  Plot of piecewise constant approximation
%             using centred values.
% Source   :  Tony Roberts, 2007
% Modified Harry Butler, Jan 2017

function  [z] = PP0plot(x,y,c)
  n=length(x);
  i=[1,floor(1:.5:n)];
  j=[ceil(1:.5:n),n];
  xmid=(x(i)+x(j))/2;
  k=ceil(.5:.5:n);
  plot(xmid,y(k),c)
end
```

- However, we want to know which point is closest: is it the first, second, third, or which? Use

  `[d,i]=min(dd)`, for example, returns `d = 0.05  i = 3`,

  to set `i` to the index of the point at which the minimum distance occurs; in this case the third is the closest.
- Then `y(i)` will be the required value.

Algorithm 17 does the same computation 'simultaneously' using arrays. The key is the computation of the array `dd` which contains the distance from each given x coordinate to each of the given X coordinate at which values are required: `dd(i,j)` contains the distance from the ith data x coordinate to the jth X coordinate at which a value is desired; the `min` function then informs us which y value to use for each given X.

The function in Algorithm 17 gives the simplest version of what some people call '*nearest neighbour interpolation*' or '*case based reasoning*' in the fields of data mining, machine learning and artificial intelligence.

The derivative of the approximating function

An approximating function that remains constant within each sub-interval cannot show anything about the rate of change of the function it is approximating. The slope of the approximating function is zero within each interval and undefined at each of the tabulation points (except that in the case of the centred values approximation the slopes are undefined at the midpoints of the intervals).

Algorithm 17 function to compute values of a piecewise constant approximation to data: for the given X the value is the y-value of the nearest supplied x-point.

```
% Function :  PP0pts
% Syntax   :  Y = PP0pts(x,y,X)
% Input    :  Vectors x and y,  (x(i),y(i)) are data points,
%             X = coords to find approximate values
% Output   :  Y = approximate values at points X
% Source   :  Tony Roberts, 2007
% Modified Harry Butler, Jan 2017
function  [Y] = PP0pts(x,y,X)
  n=length(x); % number of given points
  m=length(X); % number of desired values
  dd=abs(x(ones(1,m),:)'-X(ones(n,1),:)) % array of distances
  [d,i]=min(dd) % find the closest in each column
  Y=y(i)
end
```

The integral of the approximating function

In scientific computing we almost always have to prepare summary information about some highly detailed computation. Such summary information is typically a sum, average or integral of quantities. Estimating integrals is a core computational task.

The definite integral of a function f,

$$\int_a^b f(x)\,dx\,,$$

can be approximated by calculating the integral of the piecewise continuous approximation may be

$$\sum_{r=1}^{n} y_r dx_r \quad \text{or in MATLAB} \quad \texttt{sum(y.*diff(x))}$$

where y represents the vector of approximating constant values and the y_r represent the elements of this vector, that is, y= $\{y_r\}_{r=1}^{n} = \{y_1, y_2, \ldots, y_n\}$.

Using MATLAB instructions[1] two alternative estimates of the integral are

```
left=sum(y(1:end-1).*diff(x))
midpt=sum(0.5*y(1:end-1).*diff(x)+0.5*y(2:end).*diff(x))
```

we get the integral of the left-approximation and of the midpoint approximation. Why does this second formula work? Because, with n data points, it divides the area under the approximation into $2(n-1)$ rectangles: the first is from x_1 to the midpoint $(x_1 + x_2)/2$ and of height y_1; the second is from the midpoint $(x_1 + x_2)/2$ to x_2 and of height y_2; the third is from x_2 to the midpoint $(x_2+x_3)/2$ and of height y_2; the fourth is from the midpoint $(x_2 + x_3)/2$ to x_3 and of height y_3; and so on.

[1] Recall that end in MATLAB refers to the length of the vector in which it appears.

4.2.2 Piecewise linear approximations

By default, MATLAB plots linear approximations for function values between tabulation or sample points. (On the screen of course these can only be shown approximately with constant approximations, but with sufficiently many pixels we will not normally be irritated by this.) So to graph a piecewise linear approximation to a function we need only call upon MATLAB's `plot` function, whose algorithm will compute the intermediate points. However, as in the case of the constant approximation, it is important to be able to compute the approximate function values for a particular set of sample points.

Most importantly, we have to vectorise such a computation. For example, suppose the x and y coordinates of five data points are

$$\begin{aligned} \mathrm{x} &= (0.21, 0.62, 1.76, 2.08, 2.28) \\ \mathrm{y} &= (0.20, 0.58, 0.98, 0.87, 0.75). \end{aligned}$$

We aim to find the value of the linear interpolant at $X = 1$ say. The first task is to find in which interval the required X lies, the second task is to then compute the linear interpolant.

1. To humans it is immediately apparent that $0.62 < 1 < 1.76$, that is, $x_2 < X < x_3$ and so the desired point $X = 1$ lies on the straight line between (x_2, y_2) and (x_3, y_3). But how does the computer determine this (using vector computations)? Testing each x_i value one after the other is not vectorisable. Instead, in MATLAB we test them all simultaneously by the comparison

   $$\mathrm{x{<}X} = (1, 1, 0, 0, 0)$$

   as the comparison returns a 'one' or 'true' for each x_i value less than the desired X. Then compute the sum of this vector via `sum(x<X)` which here returns two—this indicates that there are precisely two data points to the left of X and so here we know $x_2 < X = 1 < x_3$ (provided the x_i values are indeed sorted).

2. To compute the linear interpolation between (x_2, y_2) and (x_3, y_3) note that the desired point lies a fraction

   $$\frac{X - x_2}{x_3 - x_2} = \frac{1 - 0.62}{1.76 - 0.62} = 0.33$$

   from the left point x_2 to the right point x_3. Thus the linear functions value is the same fraction from y_2 to y_3, that is,

   $$Y = f(X) = y_2 + 0.33 * (y_3 - y_2) = 0.71\,.$$

   In summary, find the function value Y at X via

   ```
   i=sum(x<X)
   Y=y(i)+(X-x(i))/(x(i+1)-x(i))*(y(i+1)-y(i))
   ```

3. Use one further refinement: what if the desired X is outside the domain of the data x_i? Realise that *using data outside its domain is risky* as it is prone to

large errors. However, if you must, then here simply use either of the extreme line segments via `i=min(max(1,i),n-1)` to ensure the value `i` points to the end line segment nearest X whenever the required X lies outside the domain of the data.

Algorithm 18 provides a function to compute linear interpolants for many required X values simultaneously via more vectorisation.

Algorithm 18 computes points on a linear interpolation through given data points: the indices `i` find out how many data points there are to the left of each requested X coordinate; with adjustments for any X value outside the range of x; then the last line computes the linear interpolation.

```
% Function :  PP1pts
% Syntax   :  Y = PP1pts(x,y,X)
% Input    :  Vectors x and y  where  y = f(x) are data points,
%             X points at which approximate values are desired.
% Output   :  Y the approximate values at points X
% Source   :  Tony Roberts, 2007
% Modified Harry Butler, Jan 2017
function  [Y] = PP1pts(x,y,X)
  n=length(x); % number of known data points
  m=length(X); % number of requested values
  i=sum(x(ones(1,m),:)'<X(ones(n,1),:));
  i=min(max(1,i),n-1) % use extremes if outside
  Y=y(i)+(X-x(i)).*(y(i+1)-y(i))./(x(i+1)-x(i));
end
```

The derivative of the approximating function

The derivative of the approximating piecewise linear function will be constant between each of the tabulating points and undefined at the tabulation points. The slopes are given by `diff(y)./diff(x)`.

Higher order derivatives are zero at all points except the tabulation points where they are undefined.

The integral of the approximating function

Since the area under a line joining two points is the same as the area under a horizontal line at the average of the two end points, see Figure 4.3, the areas to the tabulation points are given by the same formula as for centred piecewise constant interpolation:

```
area=sum(0.5*(y(1:end-1)+y(2:end)).*diff(x))
```

Note: In the numerical integration of data sets the integration of the piecewise linear approximating function is called the Trapezoidal Rule.

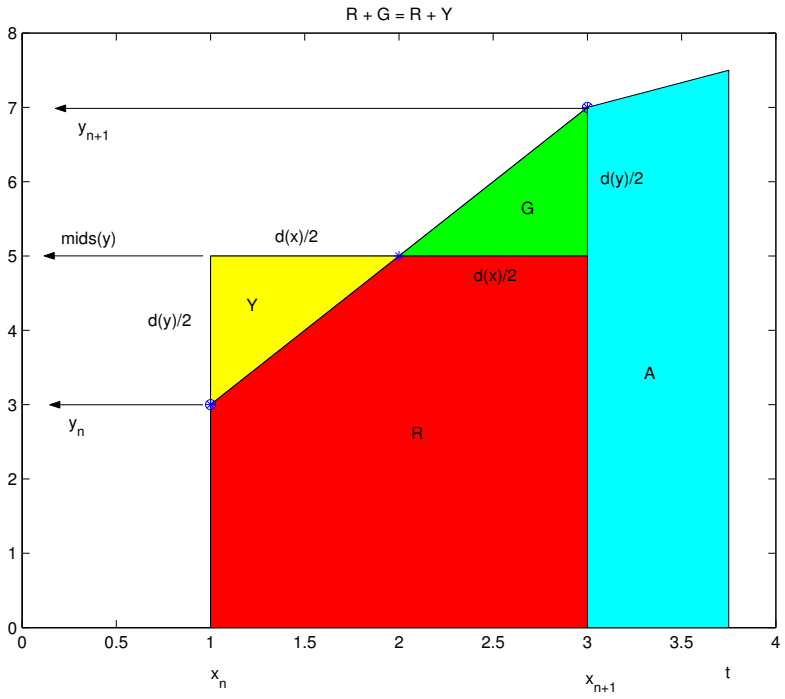


Figure 4.3: Trapezium area = area under averaged value

4.2.3 Piecewise quadratic approximations

When interpolating data we are approximating the underlying unknown function that the data represents. In most situations we expect the underlying function to be smooth and continuous—and we would like our interpolated approximation to also be smooth and continuous. A "smooth" function means that the function's derivatives are also continuous—how "smooth" a function is depends on how many derivatives of the function are continuous. Even if interpolation functions are not continuous (piecewise constant) or or are not smooth (piecewise linear) they are still useful for representing smooth data—but to compensate for their deficiencies many more data points must be used.

In many instances dictating how many data points are required for accurate interpolation is not an option—the interpolation must be adjusted to the data set. We have seen we can fit a constant, or a straight line to data—the ideas above can be extended to fitting data with any degree of polynomial we require.

To find an unknown polynomial of degree n we need $n+1$ equations so that we can find the unknown constants of the polynomial. To calculate the piecewise quadratic approximations to a data set we can use three data points for each piece. Algorithm 19 demonstrates how to move through the data—a quadratic section at a time and calculate the interpolated values (and the interpolated derivative).

Example 4.3: Consider the following data set

$$\begin{aligned} x &= (1.0, 2.0, 3.5, 6.0, 7.0) \\ y &= (6.83, 8.67, 6.42, 6.00, 12.83). \end{aligned}$$

we can fit a quadratic to the data points (1.0, 2.0, 3.5) and to the points (3.5, 6.0, 7.0). Figure 4.4 shows the piecewise linear and piecewise quadratic fits to the data.

Exercise 4.4: Algorithm 19 assumes the data set is sorted in ascending order by x and the interpolation points X are also sorted in ascending order. What would happen if they are not?

Exercise 4.5: If the restriction on sorted input data in Algorithm 19 was relaxed what MATLAB function would have to be used to pre-sort the data before being used in the `while` loop? How would it be used so that sorting on x could also sort on y?

How would the result vectors Y and dY be reordered to match the original input order of X?

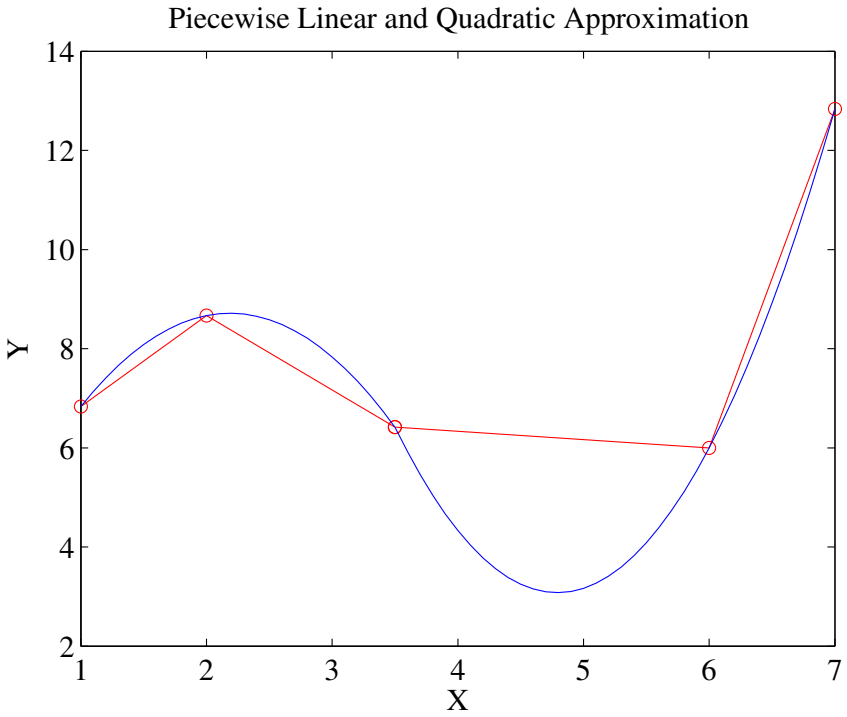


Figure 4.4: Comparison of piecewise linear and quadratic interpolation approximations using the data set of Example 4.3

The derivative of the approximating function

The derivative of the approximating piecewise quadratic functions will be a straight line for each function. Though the data point (point $x = 3.5$ in Figure 4.4) where the two quadratics meat appears to be a smooth join, closer inspection shows that the derivative is discontinuous at this point (see Figure 4.5). With quadratic approximation we can now calculate first and second derivatives but they will in general, be discontinuous as we have not attempted to specify continuous derivatives where the approximating functions meet.

Algorithm 19 computes points on quadratic interpolation functions through the given data points: the indices contained in `i` are the points found in the given quadratic section. The code loops through each quadratic section.

```
% Function : quadint
% Syntax   : [Y, dY] = quadint(x,y,X)
% Input    : Vectors x and y where y = f(x) are
%            data points, X points at which
%            approximate values are desired
% Output   : Y the approximate values at points X
%            dY the approximate derivative at points X;
% Source   : Leigh Brookshaw, 2011
% Modified Harry Butler, Jan 2017
function [Y, dY] = quadint(x,y,X)

  % Need column vectors for the data points
  x = x(:);
  y = y(:);
  n = 1;

  while n < length(x),
    dx = x(n:n+2);
    i = find( X>=dx(1) & X<=dx(3) );
    % Only calculate quadratic if it is required
    if length(i) > 0,
      dy = y(n:n+2);
      c = [ dx.*dx dx ones(3,1)];
      s = c\dy;
      Y(i) = s(1)*X(i).*X(i)+s(2)*X(i)+s(3);
      dY(i) = 2*s(1)*X(i)+s(2);
    end
    n = n + 2;
  end
end
```

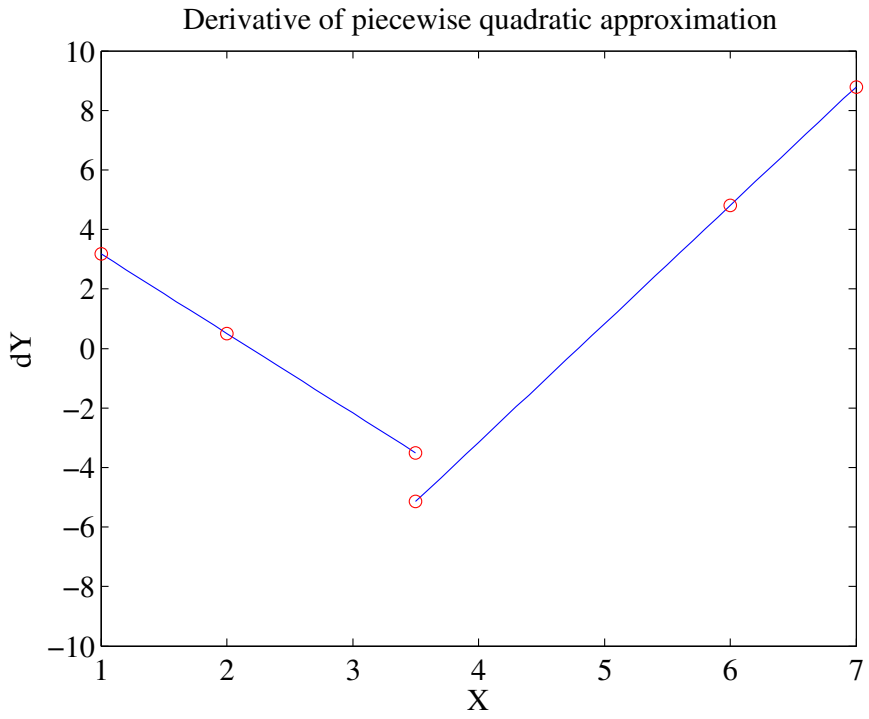


Figure 4.5: The derivative of the piecewise quadratic functions calculated from the data set of Example 4.3

The integral of the approximating function

Each quadratic approximating function can be integrated and the area under the entire domain can be found by adding the areas of each section.

Note: In the numerical integration of data sets the integration of the piecewise quadratic approximating function is called Simpson's Rule.

4.2.4 Splines

As we have seen it is possible to interpolate data by using higher and higher order polynomials. The higher the order of the polynomial the more data points that have to be used to find the polynomial (this form of interpolation is called Lagrangian interpolation). In fact we could use all the data points we have and fit one polynomial to them all. With one polynomial fitted to the entire data set we would have a very smooth interpolation function—continuous and smooth higher order derivatives could be found. Unfortunately this seldom works as the higher the order of the polynomial used to fit a data set, the more unwanted oscillations in the approximating function (unless the data describes a polynomial then the fit is very good!).

A method that can find piecewise approximating functions, ensure continuous derivatives and avoid unwanted oscillations is spline interpolation[2]. The idea be-

[2] The term is derived from the elastic rods, called *splines*, which engineers used for a long time to fit curves through given points

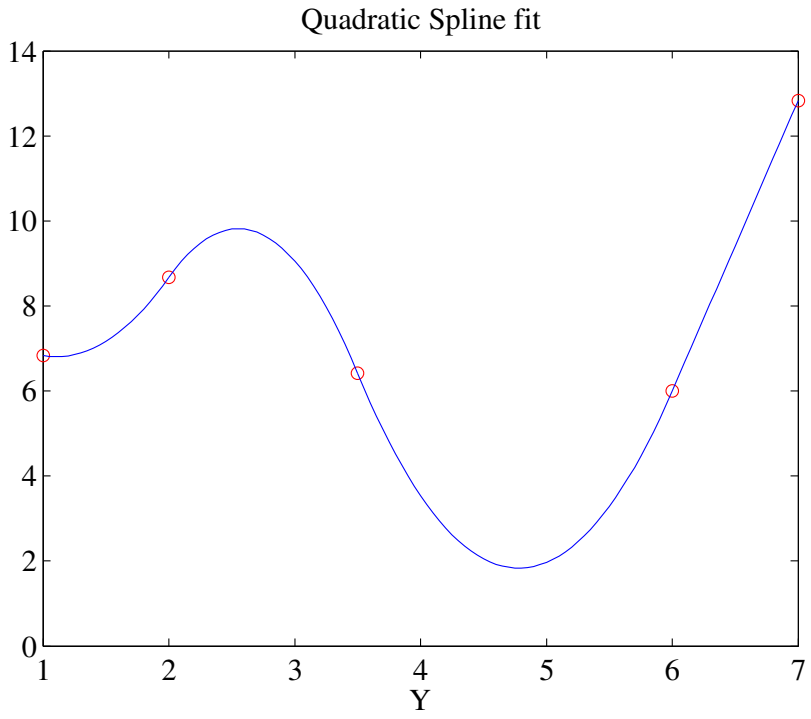


Figure 4.6: The quadratic spline fit to the data set of Example 4.3

hind spline interpolation is to fit a different interpolation function in the gaps between the data points. This is exactly what we did for piecewise linear interpolation! With two data-points we can only find the equation of a straight line—unless we specify that not only the functions are to be continuous the functions' derivatives must also be continuous where they join! The data points where splines join and the derivatives are continuous are called *knots*.

Quadratic Splines

To fit the data of Example 4.3 with splines—we will need to solve for four quadratic functions. Four quadratic functions contain twelve unknowns—so we need twelve equations in the unknowns. Using the data points of Example 4.3 we can construct eight equations—two equations for each function:

$$\begin{aligned}
6.83 &= A_0 + A_1 + A_2 \\
8.67 &= A_0 + 2A_1 + 4A_2 \\
8.67 &= B_0 + 2B_1 + 4B_2 \\
6.42 &= B_0 + 3.5B_1 + 12.25B_2 \\
6.42 &= C_0 + 3.5C_1 + 12.25C_2 \\
6.00 &= C_0 + 6C_1 + 36C_2 \\
6.00 &= D_0 + 6D_1 + 36D_2 \\
12.83 &= D_0 + 7D_1 + 49D_2
\end{aligned}$$

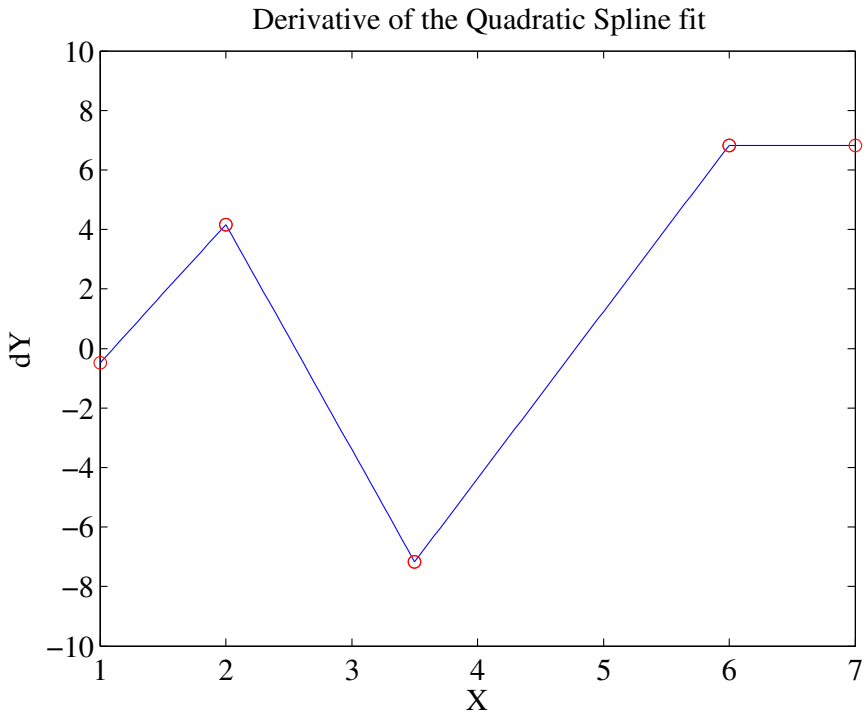


Figure 4.7: The derivative of the quadratic spline fit to the data set of Example 4.3

We need to find four more equations. If we require that the first derivative of the quadratic spline functions must be continuous at the knots (interior data points where splines meet) then we have three more equations:

$$\begin{aligned} A_1 + 4A_2 &= B_1 + 4B_2 \\ B_1 + 7B_2 &= C_1 + 7C_2 \\ C_1 + 12C_2 &= D_1 + 12D_2 \end{aligned}$$

giving eleven in total—we are short one equation. At this point we have to make a common decision when fitting splines—we have to reduce the number of unknowns by deciding on end conditions! To a certain extend this is arbitrary, but there are standard choices that are known to give reasonable results.

For quadratic splines it is common to set one of the end function to be a straight line. For Example 4.3 the best choice would be the right-hand end of the data set. This would mean that D_2 is set to zero and we now have eleven equations in eleven unknowns.

Figure 4.6 shows the quadratic spline fit to the data of Example 4.3. Figure 4.7 shows the derivative of the quadratic spline fit to the data of Example 4.3. As can be seen, the derivative is continuous across the data set but it is not smooth. The plot also shows the right-hand side interpolation function is not a quadratic but a straight line.

Cubic Splines

The most common splines used is the cubic spline. In this case we fit a cubic polynomial to each section. To fit the data of Example 4.3 with cubic splines—we will need to solve for four cubic functions. Four cubic functions contain sixteen unknowns—so we need sixteen equations. Using the data points of Example 4.3 we can construct eight equations—two equations for each function. Specifying that the first and second derivatives of the cubic spline functions are continuous at the knots gives us six more equations—for a total of fourteen equations in sixteen unknowns—two short!

For cubic splines this is expected so we need to choose end conditions that will eliminate two unknowns. The most common end conditions used are:

- setting the second derivative of the end spline to zero. This end condition when used at both ends produces what are called *natural* splines[3].
- fixing the first derivative of the end spline. This end condition when used at both ends produces what are called *clamped* splines.

  The first derivatives at the end point or points are normally known from the physical conditions that produced the data set or assumptions about the data set.
- The "not-a-knot" condition specifies that the third derivative be continuous at two *interior* points—the first and the last. The advantage of this is no assumptions are made about the end points.

With two end conditions chosen we now have sixteen equations in sixteen unknowns—and we can solve for the cubic splines.

Figure 4.8 shows the cubic spline fit to the data of Example 4.3. The "not-a-knot" end conditions was used in the fit. Figure 4.9 shows the derivative of the cubic spline fit to the data of Example 4.3

Exercise 4.6: To fit cubic splines to a data set in MATLAB use the functions `spline` and `ppval` (Just type *help spline*).

Calculate the cubic spline fit to the following data set—the Vapour pressure-temperature data for H_2O:

T ($^\circ C$)	5.0	10.0	12.0	14.0	16.0
P (mm Hg)	6.54	9.21	10.52	11.99	13.63

Compare the results when using "natural" splines and the "not-a-knot" end condition.

4.3 Least squares curve fitting

In the previous sections on polynomial interpolation, we assumed that the curve to be fitted must pass through the data points exactly. These methods are used when you know the data points are accurate.

[3] "Natural" splines most closely model the original physical spline used by engineers.

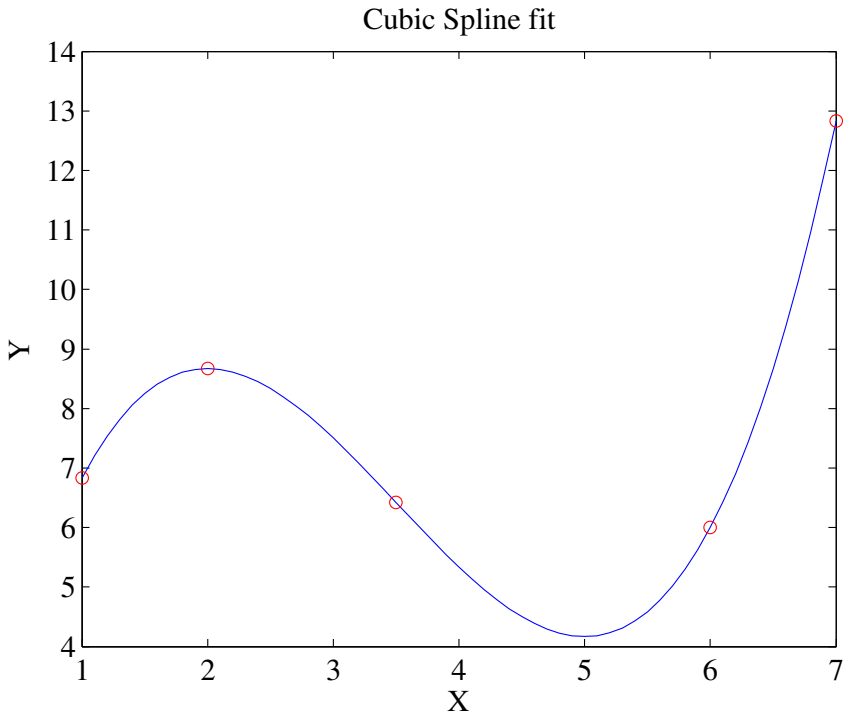


Figure 4.8: The cubic spline fit to the data set of Example 4.3

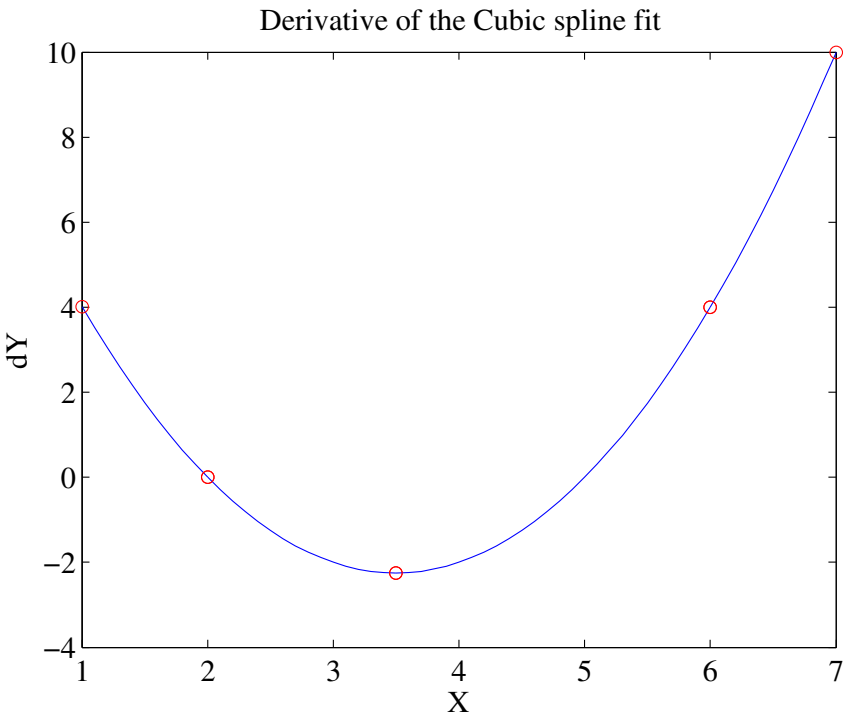


Figure 4.9: The derivative of the cubic spline fit to the data set of Example 4.3

However, almost always experimental data are collected which should lie on a straight line or on a parabola or on an exponential growth or decay curve, but do not. It is then a question of just where the straight line or parabola should go if we cannot be sure which points are in error and by how much. The line or curve need not necessarily pass exactly through any of the data points, but it should somehow be the best fit possible. Being the 'best curve', according to *least squares*, is one method.

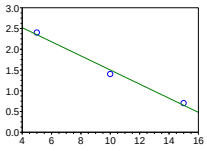
Specific objectives

- To understand the way in which it is possible to fit a straight line, or a quadratic, or a cubic polynomial, which represents a good fit to given approximate data using the method of least squares
- To be able to find the least squares fits to example data sets
- To understand how to use residuals to test the appropriateness of the choice of polynomial to be used in the least squares fit

4.3.1 Curve fitting

We begin this section by learning how to find the straight line $y = a + bx$ which *best* fits a set of approximate data points. What does *best* fit mean? If a straight line is to fit the data points, then you will want the points to be close to the line, in some sense. In this case *close* is taken to mean that the sum of the squares of all the vertical distances from the points to the line should be a minimum.

Example 4.7: power consumption Suppose you know the power consumption of a room heater at three temperatures: at $x_1 = 5^\circ\text{C}$ the power $y_1 = 2.4\,\text{kW}$; at $x_2 = 10^\circ\text{C}$ the power $y_2 = 1.4\,\text{kW}$; at $x_3 = 15^\circ\text{C}$ the power $y_3 = 0.7\,\text{kW}$. Pose the power consumption depends linearly upon the temperature, say $y = ax + b$, as plotted to the left. What are the 'best' values to choose for the coefficients $\boldsymbol{c} = (a, b)$?



Power consumption versus temperature: dots are the three data points; the line is a best fit.

Solution: The magic algorithm is to first create

$$\boldsymbol{M} = \begin{bmatrix} 5 & 1 \\ 10 & 1 \\ 15 & 1 \end{bmatrix} \quad \text{and} \quad \boldsymbol{y} = \begin{bmatrix} 2.4 \\ 1.4 \\ 0.7 \end{bmatrix}.$$

Second, find the coefficients $\boldsymbol{c}$ by solving the linear *normal equation* $(\boldsymbol{M}^{\mathsf{T}}\boldsymbol{M})\boldsymbol{c} = (\boldsymbol{M}^{\mathsf{T}}\boldsymbol{y})$ which here is

$$\begin{bmatrix} 350 & 30 \\ 30 & 3 \end{bmatrix} \boldsymbol{c} = \begin{bmatrix} 36.5 \\ 4.5 \end{bmatrix},$$

via the MATLAB command[4] `c=(M'*M)\(M'*y)` which here gives the coefficients $\boldsymbol{c} = (a, b) = (-0.17, 3.2)$. That is, this magic recipe gives the power consumption

$$y = -0.17\,x + 3.2\,\text{kW}$$

depending upon the temperature x.

[4] Remember, never compute the inverse of a matrix.

As usual the line is good within the domain of the data (interpolation), but the line is silly too far outside the domain of the data (extrapolation) as, for example, it predicts a negative power consumption, $y = -0.2\,\text{kW}$ at a temperature $x = 20^\circ\text{C}$.

The magical normal equation Suppose we have n data points, $\{(x_k, y_k)\}_{k=1}^n$, all of which should fit on some straight line, then we would like to solve the equations

$$\begin{aligned} ax_1 + b &= y_1 \\ ax_2 + b &= y_2 \\ &\vdots \quad \vdots \quad \vdots \\ ax_n + b &= y_n \end{aligned}$$

In vector form we have

$$a\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + b\begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

or

$$a\boldsymbol{x} + b\boldsymbol{u} = \boldsymbol{y} \quad \text{where} \quad \boldsymbol{u} = (1, 1, \ldots, 1)\,.$$

We cannot expect to satisfy all these n equations with only two variables available. However, we do ask: what linear combination of $\boldsymbol{x}$ and $\boldsymbol{u}$ lies closest to $\boldsymbol{y}$?

Linear combinations of $\boldsymbol{x}$ and $\boldsymbol{u}$ determine a two dimensional plane in space and with $\boldsymbol{y}$ not in this plane the closest point to $\boldsymbol{y}$ that is in the plane will be the perpendicular projection of $\boldsymbol{y}$ onto the plane. The residual vector $\boldsymbol{r} = a\boldsymbol{x} + b\boldsymbol{u} - \boldsymbol{y}$ will have the least length (the square root of the sum of the squares of the vectors components) when it is perpendicular to the plane of $\boldsymbol{x}$ and $\boldsymbol{u}$. Thus the required conditions are

$$\begin{aligned} (a\boldsymbol{x} + b\boldsymbol{u} - \boldsymbol{y})^T\boldsymbol{x} &= 0\,, \\ (a\boldsymbol{x} + b\boldsymbol{u} - \boldsymbol{y})^T\boldsymbol{u} &= 0\,. \end{aligned}$$

Multiplying these out gives

$$\begin{aligned} a\boldsymbol{x}^T\boldsymbol{x} + b\boldsymbol{u}^T\boldsymbol{x} &= \boldsymbol{y}^T\boldsymbol{x}\,, \\ a\boldsymbol{x}^T\boldsymbol{u} + b\boldsymbol{u}^T\boldsymbol{u} &= \boldsymbol{y}^T\boldsymbol{u}\,. \end{aligned}$$

In matrix form, as $\boldsymbol{y}^\mathsf{T}\boldsymbol{x} = \boldsymbol{x}^\mathsf{T}\boldsymbol{y}$ and $\boldsymbol{y}^\mathsf{T}\boldsymbol{u} = \boldsymbol{u}^\mathsf{T}\boldsymbol{y}$, we have

$$\begin{bmatrix} \boldsymbol{x}^T\boldsymbol{x} & \boldsymbol{u}^T\boldsymbol{x} \\ \boldsymbol{x}^T\boldsymbol{u} & \boldsymbol{u}^T\boldsymbol{u} \end{bmatrix}\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}^\mathsf{T}\boldsymbol{y} \\ \boldsymbol{u}^\mathsf{T}\boldsymbol{y} \end{bmatrix}$$

Define the matrix $\boldsymbol{M} = [\boldsymbol{x}, \boldsymbol{u}]$—the matrix with vectors $\boldsymbol{x}$ and $\boldsymbol{u}$ in its two columns—the equations to solve are

$$(\boldsymbol{M}^T\boldsymbol{M})\boldsymbol{c} = \boldsymbol{M}^T\boldsymbol{y}\,.$$

These equations are called the *normal equations* for linear regression.[5]

[5] Achieve the step from the equations $\boldsymbol{M}\boldsymbol{c} = \boldsymbol{y}$ to $\boldsymbol{M}^T\boldsymbol{M}\boldsymbol{c} = \boldsymbol{M}^T\boldsymbol{y}$ by simply multiplying both sides of the first equation by $\boldsymbol{M}^T$.

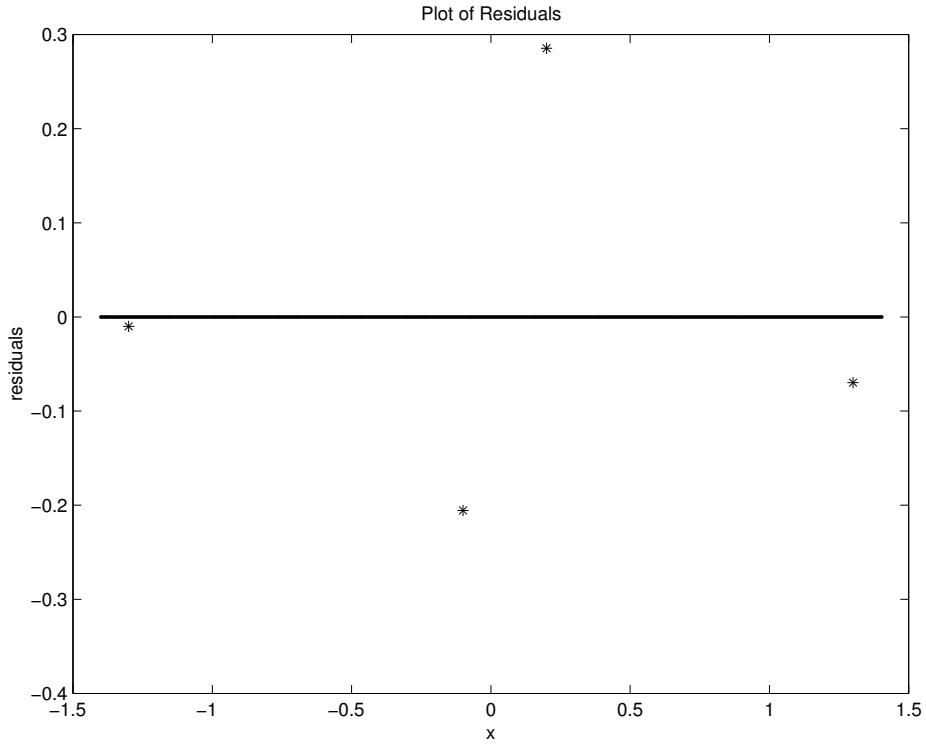


Figure 4.10: schematic plot of residuals: the difference between the calculated values of y (from the least squares line) and the data values of y_j.

Solution Now while $\boldsymbol{M}$ is certainly not a square matrix, $\boldsymbol{M}^T\boldsymbol{M}$ is so that the solution of the normal equation is easily expressed as

$$\boldsymbol{c} = (\boldsymbol{M}^T\boldsymbol{M})^{-1}\boldsymbol{M}^T\boldsymbol{y}\,.$$

However, remember the rule that one must never compute the inverse of a matrix. Instead, solve using the backslash operator. In MATLAB given column vectors `X` and `Y`, construct the matrix `M=[X,ones(length(X),1)]` and get the coefficients of the least squares line via

```
c=(M'*M)\(M'*Y)
```

4.3.2 Residuals

Inspect the *residuals* once you have determined the line of best fit. The residuals are the set of differences between the least squares values of y and the set of data points y_j. That is the set of values $\{y(x_j) - y_j \mid j = 1, 2, 3, \ldots, n\}$. In one case we get the residuals to be $-0.0100, -0.2056, 0.2855, -0.0698$ as plotted in Figure 4.10.

Note that the points are scattered reasonably randomly about the x-axis although there are really too few to tell there is any pattern to be concerned about.

Example 4.8: a poor line Find a least squares straight line fit to the data

$$\begin{aligned} \mathrm{x} &= (0, 0.6, 1.0, 1.2, 1.5, 2.1, 2.6, 3.1)\,, \\ \mathrm{y} &= (0.0025, 0.44, 1.06, 1.63, 2.36, 4.2, 6.86, 9.5)\,. \end{aligned}$$

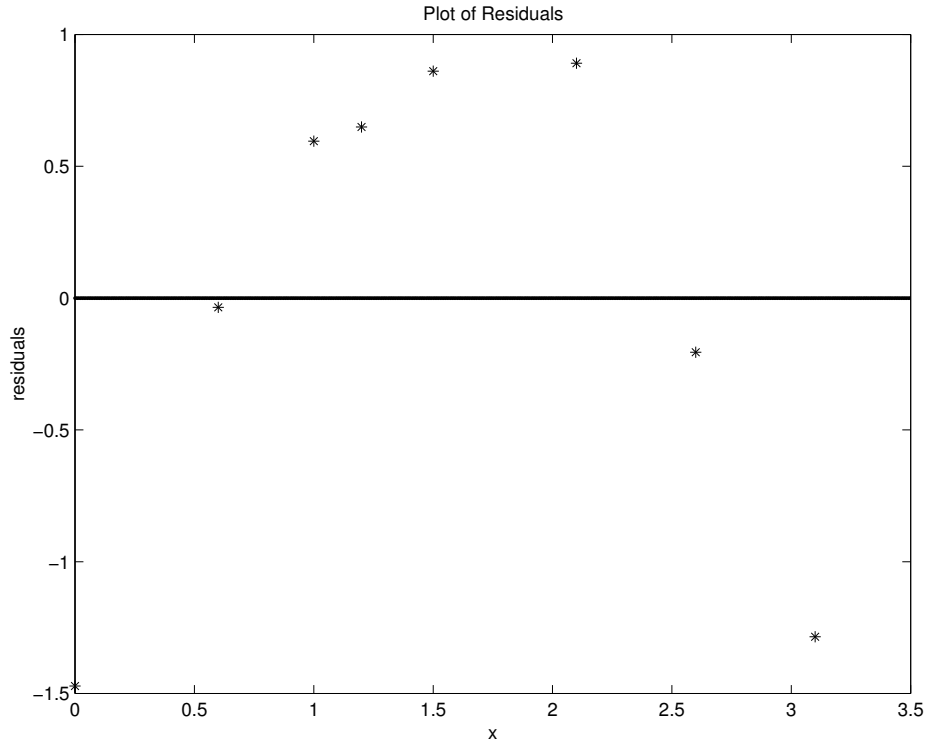


Figure 4.11: Residuals show a well-defined structure: they are not scattered evenly; thus a straight line fit is inappropriate.

Using the above method, check that the least squares line for this example is

$$y = -1.47 + 3.12x\,.$$

When we plot the residuals, see Figure 4.11, see that the they are not scattered randomly, but have some structure to them.

If we plot the original data, we see why the linear fit is not such a good idea. Figure 4.12 shows that the data is clearly closer to a quadratic curve rather than to a straight line.

Least squares polynomial fitting

Our goal is to fit a polynomial of a specific degree, say m, to n data points where $n > m + 1$ (usually many more). For example, we aim to be able to fit a quadratic curve to the data points in Example 4.8.

$$\begin{aligned}
c_m x_1^m + c_{(m-1)} x_1^{m-1} + \cdots + c_2 x_1^2 + c_1 x_1 + c_0 &= y_1 \\
c_m x_2^m + c_{(m-1)} x_2^{m-1} + \cdots + c_2 x_2^2 + c_1 x_2 + c_0 &= y_2 \\
&\vdots \quad \vdots \quad \vdots \quad \vdots \\
c_m x_n^m + c_{(m-1)} x_n^{m-1} + \cdots + c_2 x_n^2 + c_1 x_n + c_0 &= y_n
\end{aligned}$$

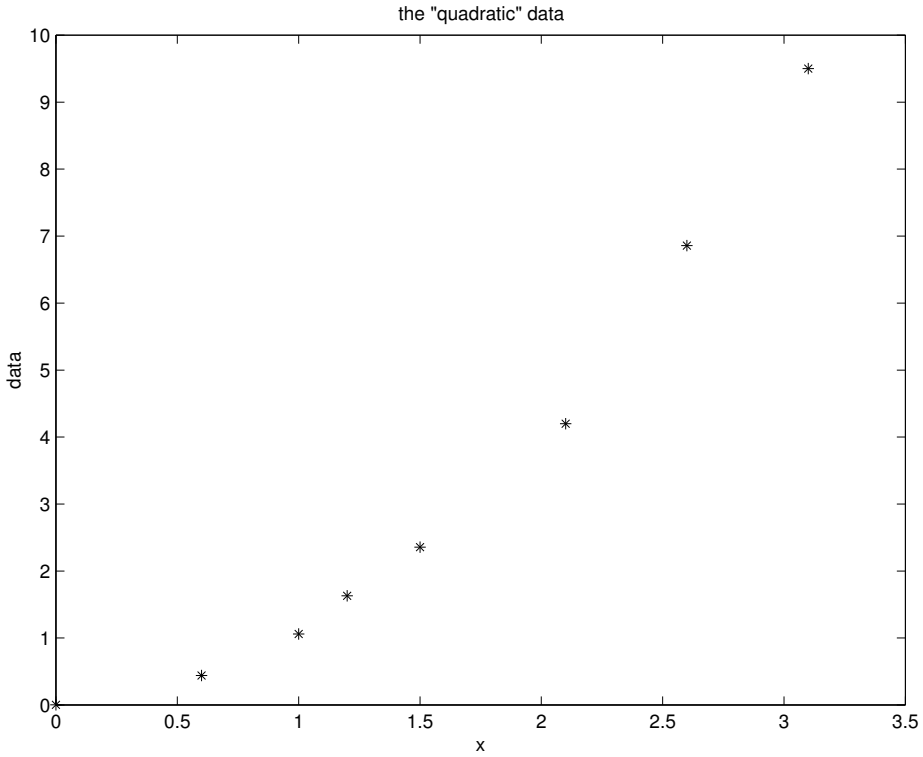


Figure 4.12: The data—also suggests the data is better fitted with a quadratic.

In vector form these equations are

$$c_m \begin{bmatrix} x_1^m \\ x_2^m \\ \vdots \\ x_n^m \end{bmatrix} + \cdots + c_2 \begin{bmatrix} x_1^2 \\ x_2^2 \\ \vdots \\ x_n^2 \end{bmatrix} + c_1 \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + c_0 \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix},$$

or

$$c_m \boldsymbol{x}^m + c_{(m-1)} \boldsymbol{x}^{m-1} + \cdots + c_2 \boldsymbol{x}^2 + c_1 \boldsymbol{x} + c_0 \boldsymbol{u} = \boldsymbol{y},$$

where $\boldsymbol{u} = (1, 1, \ldots, 1)$ and we use $\boldsymbol{x}^p = (x_1^p, x_2^p, \ldots, x_n^p)$. Again, generally we cannot satisfy all the many n equations using the relatively few $m + 1$ constants.

Again, we ask the question of what linear combination of the vectors

$$\boldsymbol{x}^m, \boldsymbol{x}^{(m-1)}, \ldots, \boldsymbol{x}^2, \boldsymbol{x} \quad \text{and} \quad \boldsymbol{u}$$

lies closest to $\boldsymbol{y}$? These linear combinations determine a hyperplane in n-dimensional space and with $\boldsymbol{y}$ not in this plane the closest point to $\boldsymbol{y}$ that *is* in the plane will be the perpendicular projection of $\boldsymbol{y}$ onto the plane. The residual vector $\boldsymbol{r} = c_m \boldsymbol{x}^m + \cdots + c_2 \boldsymbol{x}^2 + c_1 \boldsymbol{x} + c_0 \boldsymbol{u} - \boldsymbol{y}$ will have the least length (the square root of the sum of the squares of the vectors components) when it is perpendicular to the plane.[6] Thus the required conditions are

$$(c_m \boldsymbol{x}^m + \cdots + c_2 \boldsymbol{x}^2 + c_1 \boldsymbol{x} + c_0 \boldsymbol{u} - \boldsymbol{y})^T \boldsymbol{x}^m = 0$$

[6] For those who have studied *Algebra & Calculus II* you will recognise that is perpendicular to all the vectors that span that subspace, $\boldsymbol{x}^m, \boldsymbol{x}^{(m-1)}, \ldots, \boldsymbol{x}^2, \boldsymbol{x}$ and $\boldsymbol{u}$.

$$\begin{aligned} &\vdots && \vdots \\ (c_m\boldsymbol{x}^m + \cdots + c_2\boldsymbol{x}^2 + c_1\boldsymbol{x} + c_0\boldsymbol{u} - \boldsymbol{y})^T\boldsymbol{x} &= 0 \\ (c_m\boldsymbol{x}^m + \cdots + c_2\boldsymbol{x}^2 + c_1\boldsymbol{x} + c_0\boldsymbol{u} - \boldsymbol{y})^T\boldsymbol{u} &= 0 \end{aligned}$$

Multiplying these out gives

$$\begin{aligned} c_m(\boldsymbol{x}^m)^T\boldsymbol{x}^m + \cdots + c_2(\boldsymbol{x}^2)^T\boldsymbol{x}^m + c_1\boldsymbol{x}^T\boldsymbol{x}^m + c_0\boldsymbol{u}^T\boldsymbol{x}^m &= \boldsymbol{y}^T\boldsymbol{x}^m \\ &\vdots && \vdots \\ c_m(\boldsymbol{x}^m)^T\boldsymbol{x}^2 + \cdots + c_2(\boldsymbol{x}^2)^T\boldsymbol{x}^2 + c_1\boldsymbol{x}^T\boldsymbol{x}^2 + c_0\boldsymbol{u}^T\boldsymbol{x}^2 &= \boldsymbol{y}^T\boldsymbol{x}^2 \\ c_m(\boldsymbol{x}^m)^T\boldsymbol{x} + \cdots + c_2(\boldsymbol{x}^2)^T\boldsymbol{x} + c_1\boldsymbol{x}^T\boldsymbol{x} + c_0\boldsymbol{u}^T\boldsymbol{x} &= \boldsymbol{y}^T\boldsymbol{x} \\ c_m(\boldsymbol{x}^m)^T\boldsymbol{u} + \cdots + c_2(\boldsymbol{x}^2)^T\boldsymbol{u} + c_1\boldsymbol{x}^T\boldsymbol{u} + c_0\boldsymbol{u}^T\boldsymbol{u} &= \boldsymbol{y}^T\boldsymbol{u} \end{aligned}$$

In matrix form we have

$$\begin{bmatrix} (\boldsymbol{x}^m)^T\boldsymbol{x}^m & \cdots & \boldsymbol{x}^T\boldsymbol{x}^m & \boldsymbol{u}^T\boldsymbol{x}^m \\ \vdots & & \vdots & \vdots \\ (\boldsymbol{x}^m)^T\boldsymbol{x} & \cdots & \boldsymbol{x}^T\boldsymbol{x} & \boldsymbol{u}^T\boldsymbol{x} \\ (\boldsymbol{x}^m)^T\boldsymbol{u} & \cdots & \boldsymbol{x}^T\boldsymbol{u} & \boldsymbol{u}^T\boldsymbol{u} \end{bmatrix} \begin{bmatrix} c_m \\ \vdots \\ c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} (\boldsymbol{x}^m)^T\boldsymbol{y} \\ \vdots \\ \boldsymbol{x}^T\boldsymbol{y} \\ \boldsymbol{u}^T\boldsymbol{y} \end{bmatrix}.$$

The normal equation Gather the powers of $\boldsymbol{x}$ into the columns of the matrix

$$\boldsymbol{M} = \left[\boldsymbol{x}^m, \ldots, \boldsymbol{x}^2, \boldsymbol{x}, \boldsymbol{u}\right],$$

then the previous equations are

$$(\boldsymbol{M}^T\boldsymbol{M})\boldsymbol{c} = \boldsymbol{M}^T\boldsymbol{y},$$

just as in the linear case. These are the *normal equations*.

Again $\boldsymbol{M}^T\boldsymbol{M}$ is a square matrix: it is an $m \times m$ matrix and so compute the solution of the normal equation using the backslash operator.

There is nothing special about the source of the coefficients for these equations and the above theory applies to producing *least squares* solutions to any over-determined system of equations.

Example 4.9: good quadratic fit Returning to the Example 4.8 of fitting a curve to the data where we saw that a quadratic polynomial might be a better fit. Using MATLAB we would implement the following computational sequence for a solution.

```
x=[0,0.6,1.0,1.2,1.5,2.1,2.6,3.1]';
y=[0.0025,0.44,1.06,1.63,2.36,4.2,6.86,9.5]';
M=[x.^2 x ones(length(x),1)];
c=(M'*M)\(M'*y)
```

Returns:

```
c =

    0.9575
    0.0873
    0.0330
```

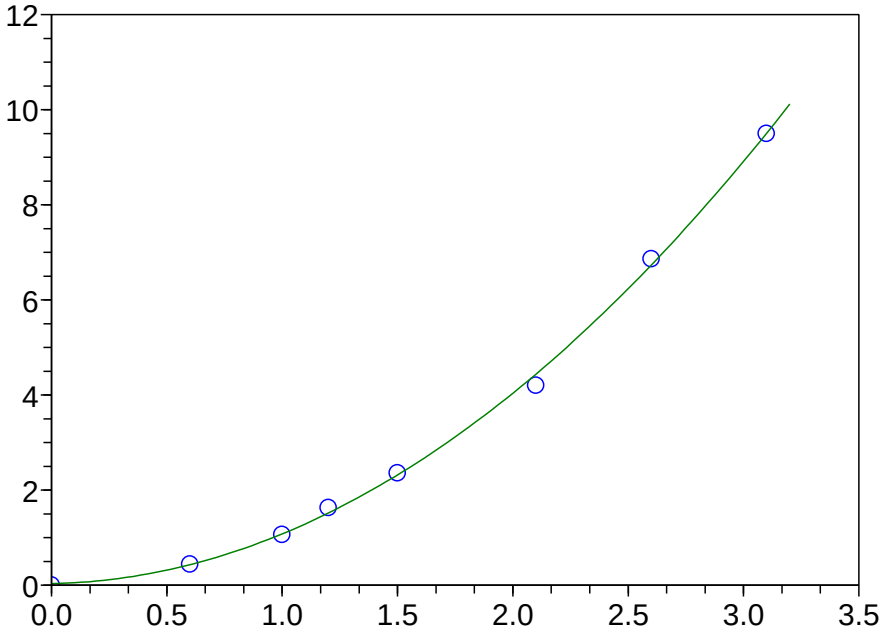


Figure 4.13: The quadratic fit to the data.

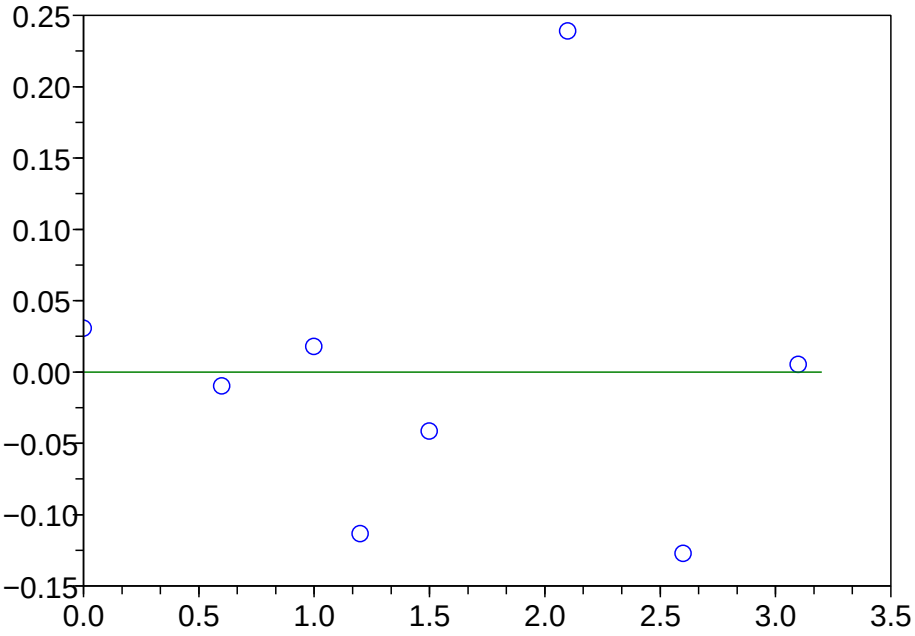


Figure 4.14: The scatter of residues for the quadratic fit to the data.

The least squares quadratic fit is then

$$p(x) = 0.0330 + 0.0873\,x + 0.9575\,x^2$$

There is no apparent structure to the residuals and so we surmise that the fit is good.

Note: the MATLAB function `polyfit` does this least square fitting for us. MATLAB's `polyfit` function with the syntax `polyfit(X,Y,n)`, given tabulation points `X`, corresponding function values `Y` and a specified degree, `n`, for the polynomial. When `n = length(X)-1` the polynomial will pass through all the data points. For smaller `n` the function returns the least squares solution and for larger `n` there are an infinite number of solutions. MATLAB warns of this but does return one answer.

4.3.3 Condition numbers favour low order polynomials

The condition number informs us not to fit high order polynomials.

Example 4.10: Avoid high order polynomials In the previous Example 4.9 we fitted a quadratic to the data. Why not fit a cubic? or higher order polynomial? surely they would be better. Maybe fit a cubic, but never any higher order polynomial.

Consider the condition numbers of the matrix $\boldsymbol{M}^{\mathsf{T}}\boldsymbol{M}$ in the normal equations. For the data of Example 4.9 and for fitting a quadratic the condition number `cond(M'*M) = 465`. Such a condition number is acceptable, although smaller would be better. It says we can reasonably accurately find a quadratic fit.

However, if we were to fit a cubic to this data, when the matrix $\boldsymbol{M}$ has an extra column of $\boldsymbol{x}^3$ values, then the condition number `cond(M'*M) = 18052`. This is a large condition number and indicates the solution, the best cubic, is very sensitive to errors in the data. Fit such a cubic only with much caution.

Attempting to fit a quartic polynomial to this data leads to condition number `cond(M'*M)` $= 10^6$ and attempting to fit a quintic leads to condition number `cond(M'*M)` $= 10^8$. These condition numbers are far too high, they indicate exquisite sensitiveness. Do not fit such high order polynomials.

4.4 Summary and key formulae

This module has covered interpolation of data and function values through piecewise continuous polynomial approximations working with function values. It also discussed the least squares best fit polynomials to data.

- Curve fitting can be exact with piecewise polynomials, or approximate with low order polynomials. Never fit a polynomial of higher order than cubic.
- Definite integrals of the form

$$\int_a^b f(x)dx$$

can be estimated by *numerical integration* rules when the integrand $f(x)$ is too complicated to integrate *analytically*. In addition, if $f(x)$ is only known as discrete values, from measurements for example, numerical rules must be used to calculate the integral. A simple, effective numerical integration rule is the *trapezium rule*

$$\int_a^b f(x)dx \approx \frac{h}{2}\left[f(x_0) + 2\left(f(x_1) + f(x_2) + \cdots + f(x_{n-1})\right) + f(x_n)\right]$$

which is *exact* for *linear* functions.

- Least Square polynomial fits to approximate data are achieved by approximately solving a linear system $\boldsymbol{Mc} = \boldsymbol{y}$ where $\boldsymbol{M}$ is the matrix of coefficients of the normal equations and $\boldsymbol{c}$ are the unknowns parameters. The vector $\boldsymbol{y}$ is the right hand side.

  For numerical work, these equations are conveniently written

  $$(\boldsymbol{M}^T\boldsymbol{M})\boldsymbol{c} = (\boldsymbol{M}^T\boldsymbol{y})$$

  where the columns of matrix $\boldsymbol{M}$ hold the powers the tabulation points up to the degree of the polynomial being fitted.

Chapter 5 Monte Carlo methods simulate complex systems

Chapter contents

5.1 Introduction

Monte Carlo methods are used to approximately simulate processes where *deterministic* mathematical models are inadequate, unavailable or too expensive; instead of this the models take on a *stochastic* nature, relying heavily on probabilities of various events taking place. Thus the techniques introduced here are very different to those studied so far in that they involve randomness. Monte Carlo methods apply to problems as simple as: the tossing of a loaded die; to the highly complex simulation of flow-fields around re-entering spacecraft, where probabilities are concerned with molecular velocities, energies and collisions; and to financial valuations of options and insurance (see MAT3104 *Random Processes to Financial Mathematics*).

At the heart of a Monte Carlo procedure is a random number generator, which produces random numbers according to a certain probability distribution. In most of the cases considered here this distribution is *uniform*, which means that the probability of generated numbers falling in any given subinterval is proportional to the size of the subinterval.

Numerical integration serves as a useful introduction to the basic principles of Monte Carlo methods. While numerical integration is a mathematical process that involves no element of chance, the Monte Carlo procedure creates a model of the process and estimates the result probabilistically. We then move on to process simulation which involves random influences from many different factors.

General objectives for this module

- To understand issues relating to generating random numbers on a computer and the problems associated with the process.
- To gain a working knowledge and understanding of Monte Carlo methods applied to numerical integration and process simulation.

Prerequisite knowledge

- Basic numerical integration rules.

5.2 Random number generation on a computer

Specific Objectives

- To understand issues relating to generating random number on a computer and the problems associated with the process.

The suggestion that a computer might produce *random numbers* on demand might seem paradoxical. Every computer function returns computed outputs based on the algorithm (the computer programme) in the function definition. The results computed given particular inputs are totally predictable. No computer generated list of random numbers can be truly random, but functions whose outputs do not seem to follow any particular pattern and seem to be unpredictable (without access to the algorithm generating the values) can be produced with some ingenuity. The popularity of running large simulations of systems that are otherwise too difficult to describe mathematically and the use of these simulations to guide decision-making means that producing a *good* random number generating function is a very important task. Computer-based random number generators are sometimes referred to as *pseudo-random* (or *quasi-random*) generators.

If the results of simulations are to be used to make important decisions, it is important to use *good* random number generators. One working definition of randomness is that if we use two different random number generators in our application then we should get statistically similar results. If we do not, then at least one of the generators is not producing a good sequence of random numbers. This definition implies that randomness is to some extent dependent on the application; that is, what is sufficiently random for one application may not be random enough for another different application.

All *good* random number generators, such as those found in MATLAB (see Figures 5.1 and 5.2), have passed many statistical tests designed to unearth any of the well known problems with random generators.[1] How much one needs to know about these tests and the random number generator used by any particular software, depends upon how important the decisions are that will be made on the basis of the results from some probabilistic model. With any random number generator it is a good idea to check that the algorithm being used to generates acceptable output (see Figures 5.5 and 5.6 for an illustration of the problems that a faulty algorithm can produce).

[1] The random number generator in Microsoft's Excel has failed a significant number of tests.

Any *pseudo-random* number generator starts with some initial input to produce its numbers. This input is called the random number *seed*. The seed determines the starting point of the random sequence. Thus if the software allows the user to reset the seed to a given value, then the same sequence of outputs will be generated. This is sometimes useful because if a simulation produces some surprising result, then one would like to be able to re-produce and study that situation. At most other times, simulations based on different random seeds are the norm.

In MATLAB the seed is reset to the same value each time a new MATLAB session is started. So if one thought to run several new parallel MATLAB sessions and collect the results, then one would find that they were identical. The random number seed for MATLAB can be changed—see `help rand` and `help randn` for details. Often the clock is used to generate a suitable seed—say by computing the number of milliseconds since 1 January, 1900, at the time of starting the session.

Exercise 5.1: Start a new MATLAB session. Generate an array of say 150 random numbers using command `rand(15,10)`. Record these numbers, say by cutting and pasting the output into a text editor like Notepad. Quit and restart MATLAB repeating the above command. Cut and paste the output to the text editor and compare. Now reissue the command in the current session. This will produce a totally different sequence of numbers.

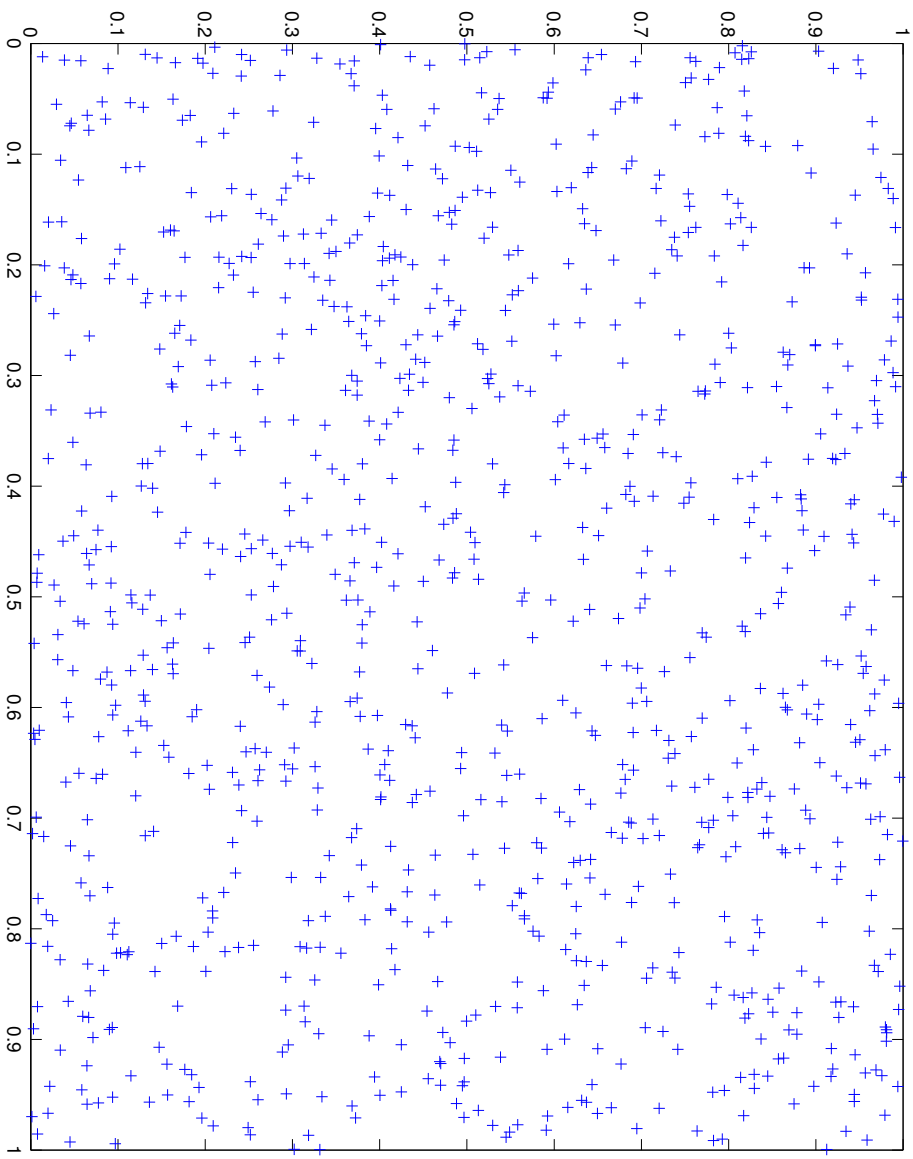


Figure 5.1: Using the MATLAB `rand` function to generate a random set of x, y coordinates between 0 and 1.

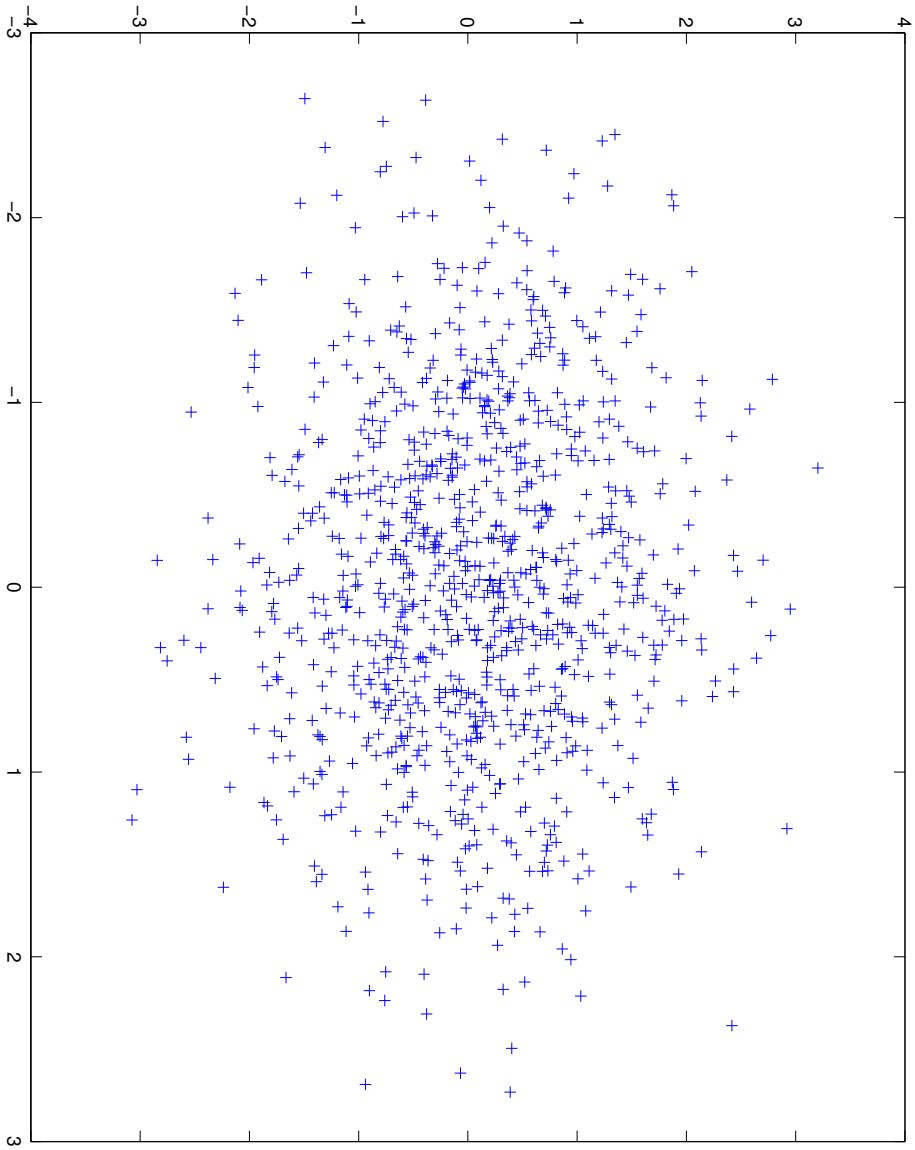


Figure 5.2: Using the MATLAB `randn` function to generate a standard-normal random set of x, y coordinates with mean 0, standard deviation 1.

In MATLAB the function `rand` generates uniformly distributed random numbers in the interval $[0, 1)$, $0 \leq x < 1$, and these are transformed to $[a, b)$, $a \leq x < b$, using $y = a+x(b-a)$, where x is the computed random number: `y=a+rand(100,1)*(b-a);`.

5.3 Monte Carlo estimates numerical integrals

Specific objectives

- To apply Monte Carlo methods for the estimation of definite integrals in one and more dimensions.
- To appreciate the convergence and accuracy behaviour of Monte Carlo integration.

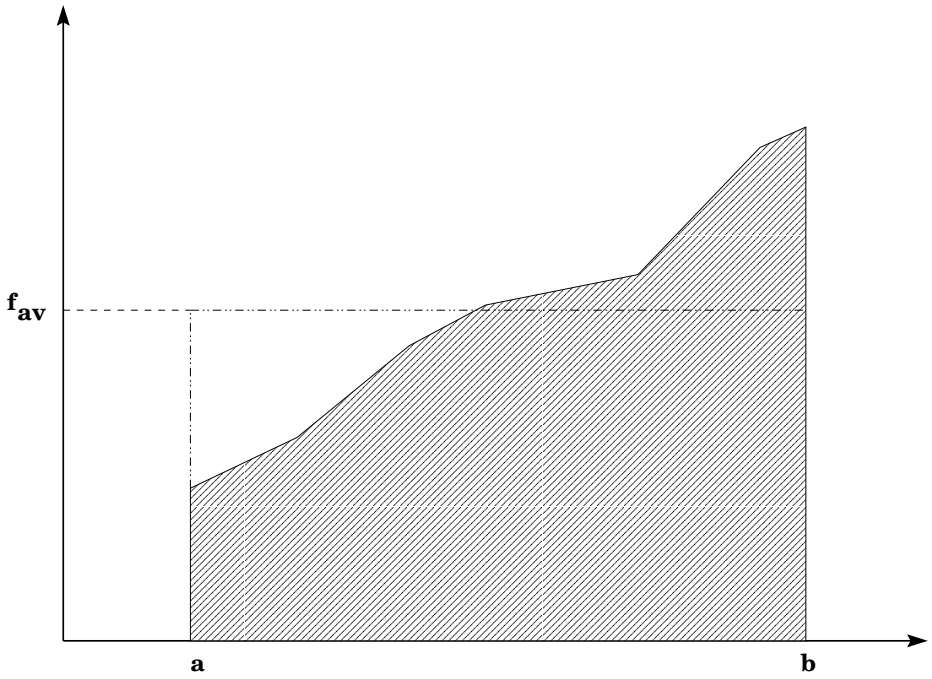


Figure 5.3: Graphical illustration of the idea behind Monte Carlo integration, namely that the $\int_a^b f(x)\,dx = (b-a)\bar{f}$. That is, the area under the curve $f(x)$ between a and b is equal to the area of the rectangle bounded by a, b and f_{av}.

Monte Carlo integration This is derived by considering the average value of a function $f(x)$ on some interval $[a, b]$ (see Figure 5.3), defined by the integral

$$\bar{f} = \frac{1}{b-a}\int_a^b f(x)\,dx\,.$$

Rearranging this gives an expression for the definite integral in terms of the average function value

$$\int_a^b f(x)\,dx = (b-a)\bar{f},$$

and it is the approximation of $\bar{f}$ with randomly generated points in $[a, b]$ that characterises the Monte Carlo approach.

Estimate an average value of the function by generating random points x_i in $[a, b]$, evaluating the function at each point, and simply averaging these values. If N points are generated in the interval, then the average of their function values is

$$\bar{f} = \frac{1}{N}\sum_{i=1}^{N} f(x_i),$$

so using the previous formula gives a Monte Carlo integral estimate

$$\int_a^b f(x)dx \approx \frac{b-a}{N}\sum_{i=1}^{N} f(x_i). \tag{5.1}$$

Example 5.2: To estimate the integral

$$I = \int_0^{\pi/2} \sin x \, dx$$

using this approach with $N = 8$ points (a totally inadequate number of points) for four different runs (a totally inadequate number of runs) gave estimates of 0.8320, 0.8507, 1.0697 and 0.9165 and, of course displayed a unacceptable degree of variability with mean 0.9172 and standard deviation 0.1079. The exact value of this integral is 1. In MATLAB these four estimates are computed with the statements.

```
N = 8; M = 4;
x = pi/2*rand(N,M);
I = pi/2*sum(sin(x))/N
[mean(I) stdev(I)]
```

where each column of results represents one particular run. The means and standard deviations are computed using `mean` and `stdev`. For larger numbers of points better results are expected, for example with $N = 32$ points another four runs gives the estimates 1.0775, 0.9890, 1.0807 and 0.9951, with mean 1.0356 and standard deviation 0.0503. Whereas $N = 10,000$ points gives 0.9960, 1.0017, 0.9987 and 1.0053, with mean 1.0004 and standard deviation 0.004.

Increasing the number of points slowly increases the accuracy. With errors of 0.0828, 0.0356 and 4.25×10^{-4}, convergence to the exact value of 1 is indeed taking place as the number of points is increased, while the standard deviation is also diminishing.

The convergence speed is very poor compared to a standard method like trapezoid or Simpson's integration. Monte Carlo is not a good technique to replace the standard ones for the numerical integration of a function of a single variable. However, in high-dimensional integrations (often occurring in financial mathematics) its advantages of simplicity and minimal storage requirements outweighs the speed advantage of more precise techniques that require careful analysis and the storing of large grids and their associated data.

Typically errors decrease like $1/\sqrt{N}$ Repeating the previous calculation with more values of N offers greater insight into the error behaviour of Monte Carlo integration. Calculating errors from the mean integral estimate at each N and taking a log plot of these against N gives typical results shown in Figure 5.4. The error is far from monotonically decreasing, but clearly shows an affinity for the theoretical behaviour $E = cN^{-1/2}$ (we will not prove this result here), shown as a dashed line. Recall trapezoid error is like N^{-2} and Simpson's is like N^{-4}, which amounts to considerably steeper and better error lines than that shown in the plot.

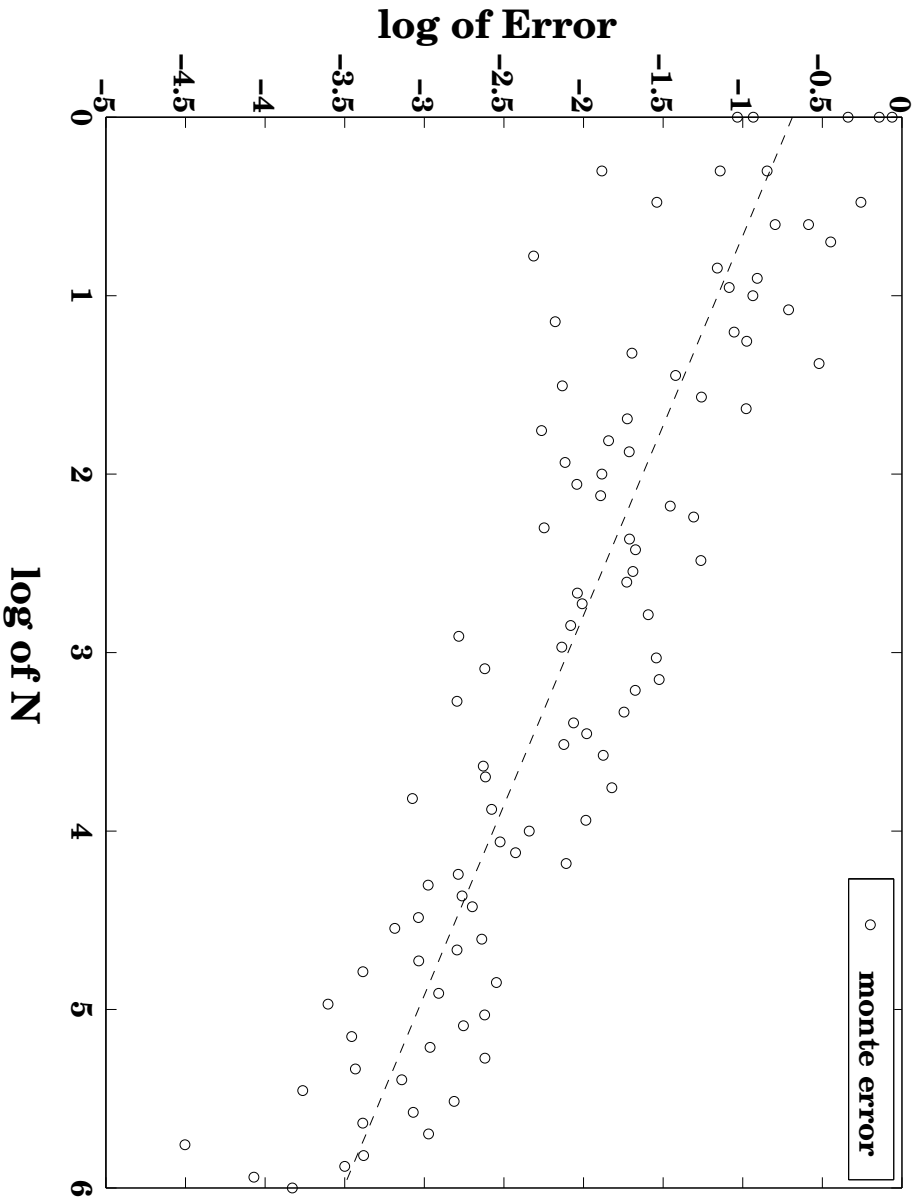


Figure 5.4: Error behaviour for Monte Carlo integration of the definite integral $\int_0^1 e^x \, dx$. The dashed line represents the theoretical error estimate $E = cN^{-1/2}$.

Exercise 5.3: Calculate some Monte Carlo estimates for the integral $\int_0^1 e^x \, dx$ and observe the error behaviour shown in Figure 5.4.

Although not normally competitive for one dimensional integration, Monte Carlo methods often become the only option for multiple integrals, where complex integration regions, large storage requirements for multi-dimensional grids and expensive function evaluations combine to make traditional approximation formulae awkward and inefficient. For example, some financial computations require a dimension for each day covered by the financial process: thus a one year analysis requires integrals in 360 dimensions, that is, a 360-fold multiple integral. Monte Carlo methods are the only remotely practical ways to estimate such high dimen-

sional integrals.

Example 5.4: The double integral

$$I = \int_{-1}^{1}\int_{-1}^{1} (x^2 + y^2)\, dx\, dy$$

involves a simple square region in the (x, y) plane. Let us first of all apply the trapezium rule. Integrating in y first and applying the trapezium rule on n intervals with $h = \frac{2}{n}$ gives

$$\begin{aligned} I &= \int_{-1}^{1}\int_{-1}^{1} f(x,y)\, dx\, dy \\ &\approx \frac{h}{2}\int_{-1}^{1}\left[f(x,y_0) + 2\sum_{j=1}^{n-1} f(x,y_j) + f(x,y_n)\right] dx \\ &= \frac{h}{2}\int_{-1}^{1} f(x,y_0)\, dx + h\sum_{j=1}^{n-1}\int_{-1}^{1} f(x,y_j)\, dx + \frac{h}{2}\int_{-1}^{1} f(x,y_n)\, dx. \end{aligned}$$

Applying the trapezium rule again with the same spacing in x then gives

$$\begin{aligned} I &\approx \frac{h^2}{4}\left[f(x_0,y_0) + 2\sum_{i=1}^{n-1} f(x_i,y_0) + f(x_n,y_0)\right] \\ &+ h\sum_{j=1}^{n-1}\frac{h}{2}\left[f(x_0,y_j) + 2\sum_{i=1}^{n-1} f(x_i,y_j) + f(x_n,y_j)\right] \\ &+ \frac{h^2}{4}\left[f(x_0,y_n) + 2\sum_{i=1}^{n-1} f(x_i,y_n) + f(x_n,y_n)\right] \\ &= h^2\sum_{i=1}^{n-1}\sum_{j=1}^{n-1} f(x_i,y_j) \\ &+ \frac{h^2}{4}\left[f(x_0,y_0) + f(x_0,y_n) + f(x_n,y_0) + f(x_n,y_n)\right] \\ &+ \frac{h^2}{2}\left[\sum_{i=1}^{n-1}\left(f(x_i,y_0) + f(x_i,y_n)\right) + \sum_{j=1}^{n-1}\left(f(x_0,y_j) + f(x_n,y_j)\right)\right] \end{aligned}$$

which is starting to look tedious. In MATLAB, with vectorisation, it is much simpler.

Using $\mathscr{A}$ as the area of the region of integration, a Monte Carlo estimate on the same number of points, $N = n^2$, takes the simple form

$$I \approx \frac{\mathscr{A}}{N}\sum_{i=1}^{N} f(x_i, y_i),$$

where (x_i, y_j) are randomly generated points in the square domain of integration with area $\mathscr{A} = 4$. Comparisons of the two computed estimates are given in Table 5.1, in which the Monte Carlo gives rough estimates at low N and slowly improves with larger N.

Algorithm 20 Monte Carlo and trapezoidal integration.

```
n = 100;  % change as required
% trapezoidal integration
x = linspace(-1,1,n); y = x;
dx = diff(x); dy =diff(y); g=zeros(1,n);
for k=1:n
 f = x.^2+y(k)^2;
 g(k) = sum(0.5*(f(1:end-1)+f(2:end)).*dx);
end
trap=sum(0.5*(g(1:end-1)+g(2:end)).*dy)
% Monte Carlo integration
x=rand(n,1); y=rand(n,1);
f=x.^2+y.^2;
monte=4*mean(f);
```

Table 5.1: $\int_{-1}^{1}\int_{-1}^{1}(x^2+y^2)\,dx\,dy$ using Monte Carlo and Trapezoids.

N	Monte Carlo	Trapezoid
4	2.7685	3.2593
9	2.6247	2.7500
25	2.5992	2.6759
81	2.5826	2.6675
289	2.6947	2.6667
1089	2.6735	2.6667
4225	2.6727	2.6667

Exercise 5.5: Use the Monte Carlo method to compute estimates of the integral

$$I = \int_{-1}^{1}\int_{-1}^{1} (x^2 + y^2)\sin(\omega x)\, dx\, dy,$$

for various values of ω ("omega").

Cut out complex domains The previous example involved a double integral on a very simple region of the (x, y) plane. For more complex regions the integration limits are functions, giving the form

$$I = \int_a^b dx \int_{h(x)}^{g(x)} f(x, y) dy,$$

and random points must be generated in this region to calculate Monte Carlo estimates.

Example 5.6: For example the integral

$$I = \int_0^1 dx \int_x^1 f(x, y)\, dy$$

is calculated on a triangular region. The region is $0 \le x \le 1$, $x \le y \le 1$ with area $\int_0^1 dx \int_x^1 dy = 1/2$. A Monte Carlo estimate is then

$$I \approx \frac{1}{2N}\sum_{i=1}^{N} f(x_i, y_i),$$

where (x_i, y_i) are uniformly distributed in the triangle. To achieve this one may be tempted to generate uniform random x values in $[0, 1)$ and for each of these to take a random y in $(x, 1)$. This approach will not produce a uniform distribution of points in the triangle (see Figure 5.5) because any two equal width strips in the triangle parallel to the y axis will contain approximately the same number of points. The two strips have different areas and therefore should not contain the same number of points if the distribution is uniform. One way of generating uniformly distributed points in the triangle is to generate uniform points in the square $(0 \le x \le 1, 0 \le y \le 1)$ and simply discard those that lie outside the triangle (see Figure 5.6).

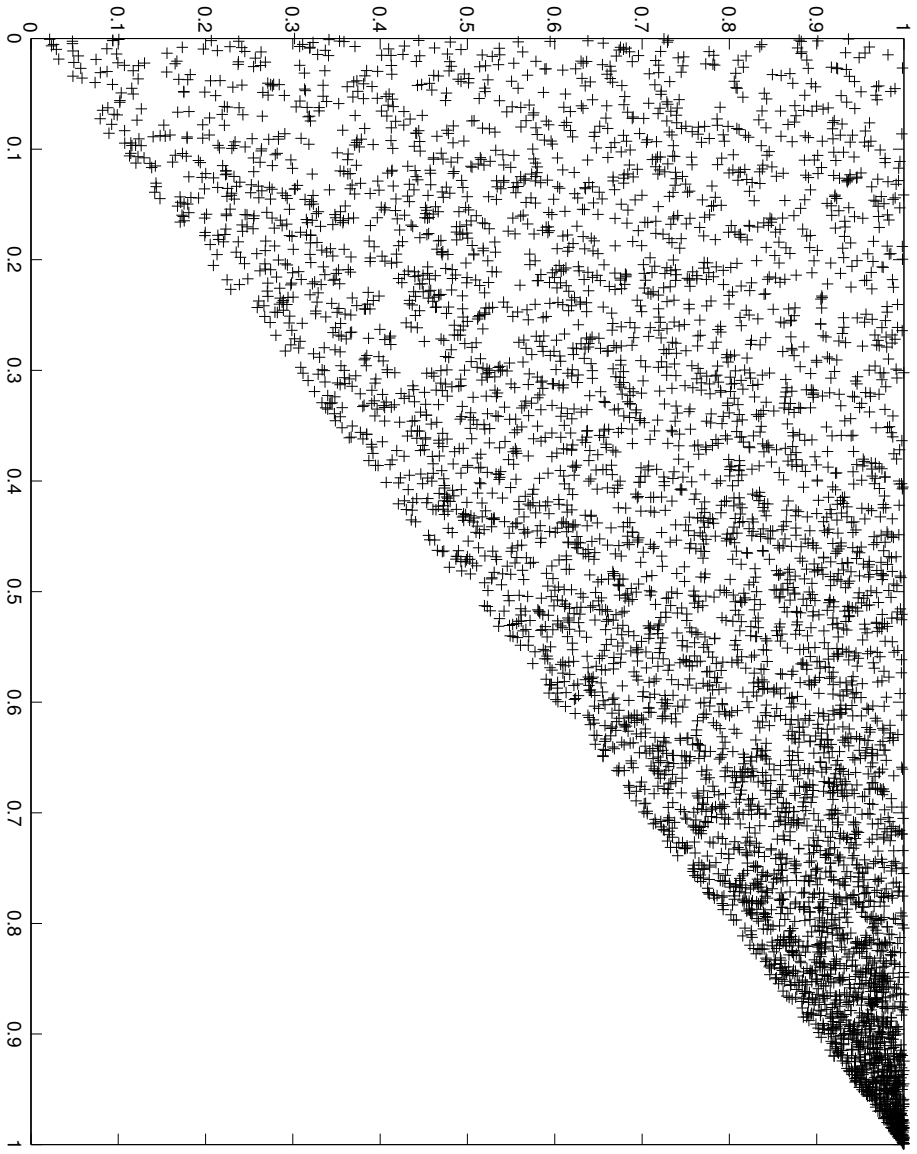


Figure 5.5: Using the MATLAB `rand` function to generate a random set of x, y coordinates such that $0 \leq x \leq 1$ and $x \leq y \leq 1$. Notice generating the random numbers by first generating the x and than generating y such that it is a random number between x and 1 produces a non-uniform distribution of coordinates.

For example, for $f(x, y) = x^2 + y$ over this triangular domain the `find` command empowers us to simply compute the integral as

```
N=1000
x=rand(N,1); y=rand(N,1);
i=find(x<y);
f=x(i).^2+y(i);
I=mean(f)/2
```

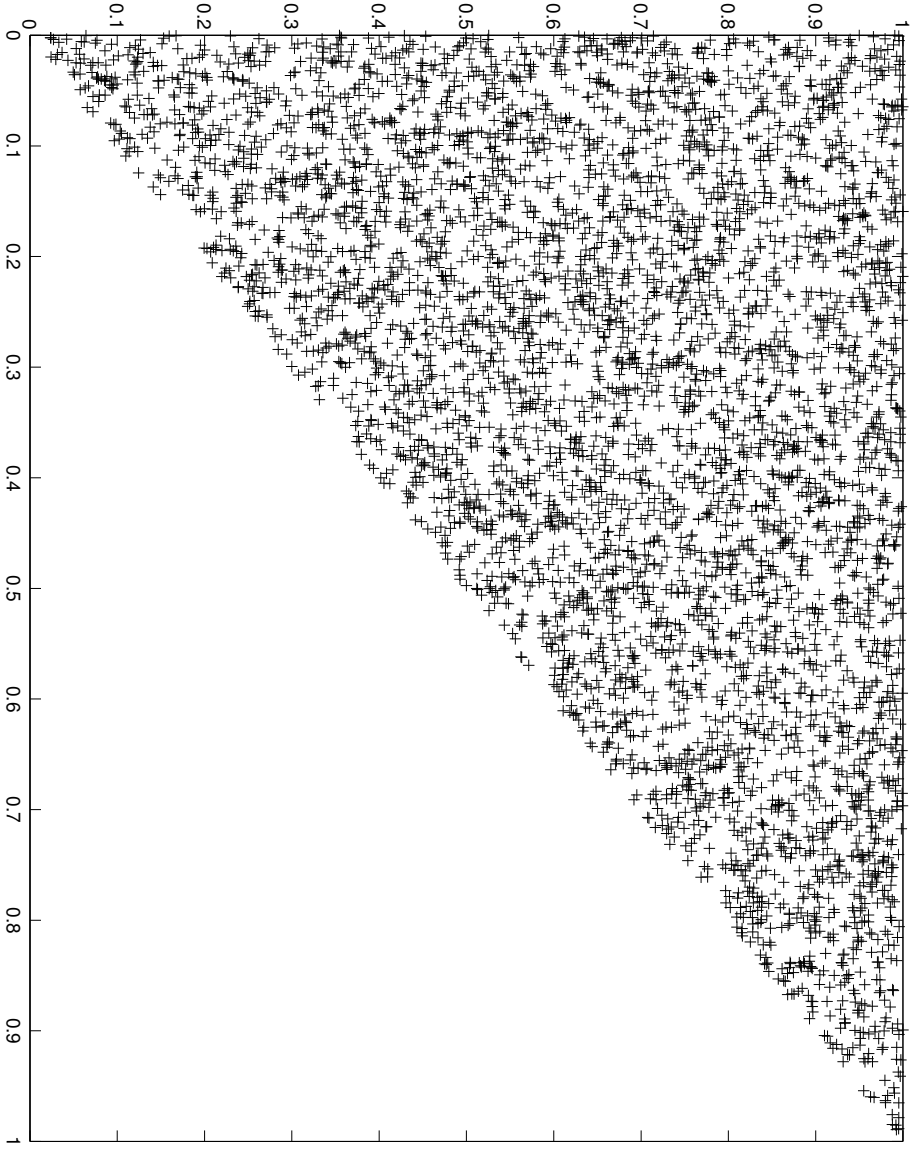


Figure 5.6: A much better approach to generating random numbers in the region $0 \leq x \leq 1$ and $x \leq y \leq 1$ is to generate random coordinates in the unit square and discard those outside the required region. Notice that when compared to Figure 5.5 the distribution is much more uniform.

Example 5.7: The double integral

$$I = \int_0^1 \int_0^{\sqrt{1-x^2}} e^x \cos(xy)\, dx\, dy$$

is calculated on the first quadrant of the unit circle with area $\pi/4$. A Monte Carlo estimate

$$I \approx \frac{\pi}{4N} \sum_{i=1}^{N} f(x_i, y_i)$$

is calculated by generating random points in the square ($0 \le x \le 1, 0 \le y \le 1$) and discarding those that lie outside the unit circle $x^2 + y^2 = 1$. Let M uniformly distributed points be generated in the square and N be the number of these lying in the unit circle. The following MATLAB script loops through the M values $M = 2, 4, 8, \ldots$, generates random points for each M, discards those not in the circle, counts those in the circle and calculates the corresponding Monte Carlo integral estimates.

Algorithm 21 Monte Carlo integral on quadrant domain

```
% Function: MCex2
% Syntax: I =  MCex2(M)
% Inputs: M = number of random (x,y) values from domain D,
%             the first quadrant of unit circle
% Output: I = value of the integral over D of exp(x)*cos(xy)
% Algorithm: Monte Carlo Integration with M points
function I = MCex2(M)
x = rand(M,1); y = rand(M,1);
s = x.^2 + y.^2;          % distance of points from origin
i = find(s < 1);          % indices of points in domain
f = exp(x(i)).*cos(x(i).*y(i)); % function values at the points
I = (pi/4)*mean(f);       % area of domain * mean
```

To collect results for various numbers M of sample points execute a sequence like, see Table 5.2,

```
M = 2 .^[1:9];  I=[];
for m=M
  I = [I,MCex2(m)];
end
[M;I]'
```

Note that if you produce a table using the same MATLAB instructions, you will not get the same results, as each run uses random points.

5.3.1 Exercises

Ex. 5.8: The double integral

$$\int_0^1 \int_0^1 \frac{dx\, dy}{1 - xy}$$

has the exact value $\pi^2/6$. Compute Monte Carlo estimates of the integral and observe the error behaviour.

Table 5.2: Monte Carlo Estimates of the double integral $I = \int_0^1 \int_0^{\sqrt{1-x^2}} e^x \cos(xy)\, dx\, dy$ computed by generating M uniform random points in the square $(0 \le x \le 1, 0 \le y \le 1)$ and discarding those that lie outside the unit circle.

M	N	N/M	I
2	2	1.0	1.3970
4	4	1.0	1.4674
8	5	0.625	1.1928
16	10	0.625	1.2018
32	25	0.7812	1.2499
64	53	0.8281	1.2380
128	105	0.8203	1.2412
256	192	0.75	1.2016
512	402	0.7852	1.2115

Ex. 5.9: Compute Monte Carlo estimates of the triple integral

$$\int_0^{\pi}\int_0^{\pi}\int_0^{\pi} \frac{dx\, dy\, dz}{3 - \cos x - \cos y - \cos z}$$

which has exact value 0.50546 . Observe the error behaviour.

Ex. 5.10: Compute Monte Carlo estimates for the integral

$$\int_2^3 \int_{1+y}^{\sqrt{y}} x^2 \cos(y)\, dx\, dy\,.$$

Ex. 5.11: Use Monte Carlo integration to find the value of the integral

$$\iint_{\mathscr{R}} \sin\sqrt{\ln(x + y + 1)}\, dx\, dy$$

where $\mathscr{R}$ is the region given in Figure 5.7, or in classical mathematical notation:

$$\mathscr{R} = \left\{(x, y) \mid \left(x - \frac{1}{2}\right)^2 + \left(y - \frac{1}{2}\right)^2 \le \frac{1}{4}\right\}.$$

5.4 Simulate processes with Monte Carlo

Specific objectives

- To apply Monte Carlo methods for the simulation of simple processes involving an element of chance

Many of the complex processes occurring around us all the time are very difficult or even impossible to model let alone solve via deterministic means. Examples include the traffic flow through a road intersection, where the predictions of the flow density are required to design and maintain an appropriate regulation system such as traffic lights or up a long incline where the information will influence the route

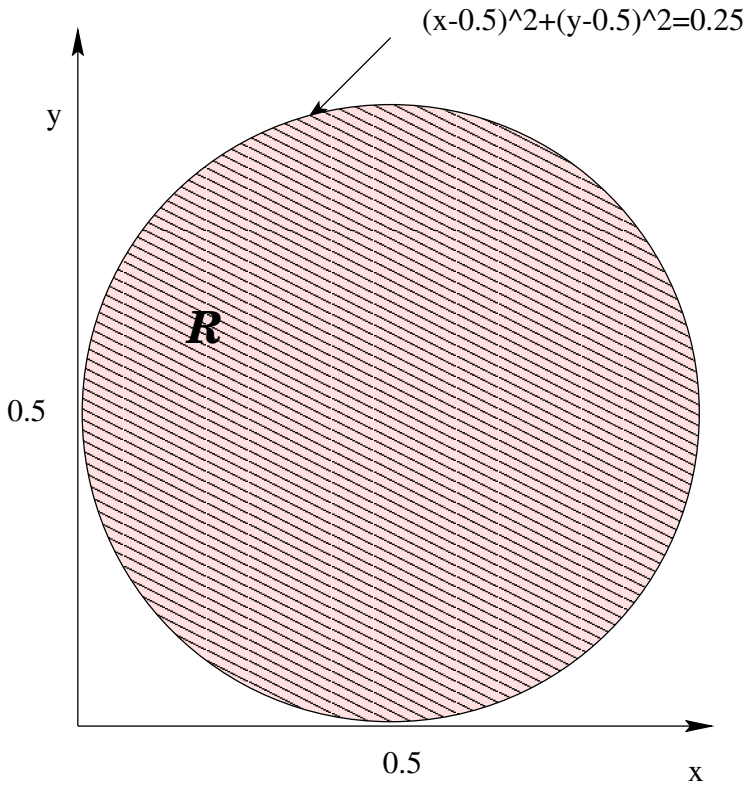


Figure 5.7

Table 5.3: Example Probability Distribution of a loaded die

Number	1	2	3	4	5	6
Probability	.2	.14	.22	.16	.17	.11
Cumulative Probability	.2	.34	.56	.72	.89	1

and design of the road; in airline flight scheduling, where predictions of passenger demand at various times are required and obviously in the insurance industry. In these examples human behaviour plays a major role in the problem's intractability; however, this is not always the case. In complex physical problems from particle and astrophysics, Monte Carlo simulation obtains useful results at relatively low cost.

5.4.1 Loaded die problem

A simple introduction to the idea of Monte Carlo simulation is to consider how we would simulate the results of rolling a loaded die. The object of the Monte Carlo simulation is to use randomly generated numbers to simulate the random aspect of the problem, which in this case is the loading of the die. To illustrate how we would do this for the loaded die case, consider a die loaded as show in Table 5.3.

If we select a random number x from a uniform distribution in the interval $[0, 1)$ and we were to break the interval $[0, 1)$ into six equal size intervals, then there would be an equal probability of x falling in any of those intervals. While this would be adequate to simulate the rolling of a normal die, it is not sufficient to

solve our current problem. So we need to weight the size of intervals by size of the probability of each number. This means that in our example we would divide the interval $[0, 1)$ up into the following subintervals based on cumulative probabilities:

$$[0, .2), [.2, .34), [.34, .56), [.56, .72), [.72, .89) \text{ and } [.89, 1)$$

The probability of x falling in the interval $[0, .2)$ is 0.2 which is the same as rolling a 1 on our loaded die. Similarly the probability of x falling in interval $[.2, .34)$ is 0.14 which is the same as rolling a 2 on our loaded die. So to simulate one roll of our loaded die all we have to do is:

1. generate a random number x from a uniform distribution between $[0, 1)$;
2. check to see in which of the intervals x falls, that is, $[0, .2)$, $[.2, .34)$, $[.34, .56)$, $[.56, .72)$, $[.72, .89)$ and $[.89, 1)$;
3. assign either 1, 2, 3, 4, 5 or 6 to the roll depending which interval x fell in, that is,

$$\begin{aligned} [0, .2) &\longrightarrow 1 \\ [.2, .34) &\longrightarrow 2 \\ [.34, .56) &\longrightarrow 3 \\ [.56, .72) &\longrightarrow 4 \\ [.72, .89) &\longrightarrow 5 \\ [.89, 1) &\longrightarrow 6 \end{aligned}$$

Normally we would not just simulate one roll, but the results of thousands of rolls and a number of repeats of the experiment. A example of a MATLAB program which does this is given below along with the results of running it. Notice that the simulated probabilities are close to those of the load of the dice, as expected.

Example 5.12: The results from running Algorithm 22 twice with the same inputs is shown below. Notice that each run of the program produces different results. This should be the case any time you run a Monte Carlo simulation.

```
-->load_die(4,8,[.2 .14 .22 .16 .17 .11])
 ans  =
    1.    1.    4.    6.    5.    6.    5.    3.
    4.    3.    3.    3.    4.    2.    3.    6.
    4.    1.    1.    6.    1.    4.    5.    1.
    4.    4.    2.    3.    2.    5.    4.    1.

-->load_die(4,8,[.2 .14 .22 .16 .17 .11])
 ans  =
    4.    1.    1.    2.    4.    5.    3.    5.
    4.    2.    5.    3.    4.    2.    5.    3.
    1.    1.    4.    3.    1.    3.    4.    6.
    4.    1.    4.    4.    2.    5.    5.    6.
```

Algorithm 22 loaded die

```
%  Function : load_die
%  Syntax   :  n = load_die(N,M,load)
%  Inputs   : load = row vector containing the probabilities
%                    of casting a  1,2,3,4,5,....
%              NxM = number of throws of the die
%  Outputs:    n = matrix containing throws of the die.
%  Source:     H.Butler 29/2/2000
%              Last Modified: Tony Roberts, 2008
%              Modified: Harry Butler, Jan 2017
function n = load_die(N,M,load)
n = []; % If function fails, output is empty
  %%%%%%%%%%%%%%%% Data checking %%%%%%%%%%%%%%%%%%%%%%%%%%
  if (min(N,M)<=0 | min(load)<0)% Check that all inputs are positive
    error('load_die: all inputs must be positive.');
  end;
  load = load(:);          % ensure load is a vector.
  if  sum(load)~=1
    error('load_die: vector of probabilities must sum to 1');
  end;

  %%%%%%%%%%%%%%%% Count Occurences %%%%%%%%%%%%%%%%%%%%%%%%
  load = cumsum(load); % work with the cumulative probability
  p = rand(N,M);       % generate a matrix of NxM random numbers
  n = ones(N,M);       % establish matrix for number of occurrences
  for i=1:length(load) % loop to work out number thrown
    n=n+(p>load(i));
  end
end
```

To check the frequencies we might execute

```
N=1000
spots=1:6;
n=load_die(N,1,[.2 .14 .22 .16 .17 .11]);
freq=sum(n(:,ones(6,1))==spots(ones(N,1),:))/N;
[spots;freq]
```

and get the following output

```
 ans  =
    1.      2.      3.      4.      5.      6.
    0.19    0.15    0.234   0.127   0.202   0.097
```

which shows that the frequencies in a thousand throws of this loaded die do indeed match reasonably closely the prescribed distribution of probabilities.

Note that no matter how may repetitions N we do the frequencies will not *exactly* match the theoretically expected values.

Beware of round-off When we want to test an unbiased die, we come across an unexpected problem. Equal probabilities can be obtained from `eqp = ones(1,6)/6`. Apparently `sum(eqp)` returns 1 but `1== sum(eqp)` is rejected. Trying `1-sum(eqp)` shows up as 1.1102×10^{-16}. This is a manifestation of round-off errors occasioned by the divisions by 6. The comparison test should be adjusted to be a little less stringent.

5.4.2 Birthday problem

The Birthday problem is another good example of how we can use Monte Carlo simulation techniques to solve problems involving a certain degree of randomness. This problem involves the calculation of the probability that in a group of N people there is a least one repeated birthday. It is a surprise to many people that you only need 23 people to have a better that a 50% chance that this will occur. The probability that no two people in a group of N will share a birthday is

$$\frac{365 \times 364 \times 363 \times \cdots \times (365 - N + 1)}{365^N}. \tag{5.2}$$

While this theoretical expression solves the problem, what we are interested in here is how we can simulate this problem on a computer (that is, capture the randomness of this behaviour). Since we are dealing with birthdays, we somehow need to generate a list of random birthdays for a group of N people. So what is the easiest way to do this ? Well if we consider a year to consist of 365 days, and a we generate a random number between 1 and 365, this will give as the birthday of one of our N people. We then repeat this for the N people in the list. So we now have a list of birthdays for the N people, we than compare all the birthdays and record if we have at least one match in that group. If repeat this M times and recorded the proportion of these events for which at least one match occurred, we will than have an estimate of the probability of this occurring. The example below gives example of how such a process may be programmed into MATLAB.

Algorithm 23 estimate the birthday probabilities

```
%  Function :  birthday
%  Syntax   :  p = birthday(N,M)
%  Inputs   :  N = number of people
%              M = number of repeats of the experiment.
%  Outputs:    p = frequency in each experiment of two
%                  or more people having the same birthday.
%  Source:     H.Butler 29/2/2000
%              Last Modified: Tony Roberts 2008
%              Modified Harry Butler, Jan 2017
function [p] = birthday(N,M)
  p = []; % If function fails output is empty
  %%%%%%%%%%%%%%%%%% Data checking %%%%%%%%%%%%%%%%%%%%%%%%%%
  if min(N,M) <=0
    error('birthday: N and M must be positive.');
  end;

  %%%%%%%%%%%%%%%%%% Counting matches %%%%%%%%%%%%%%%%%%%%%%%%
  bdays=ceil(365*rand(N,M));  % random integer birthdays
  bdays =sort(bdays);  % order so same birthdays are adjacent
  count=sum(bdays(1:N-1,:)== bdays(2:N,:))
  p = sum(count >=1)/M;
end
```

Running this program and comparing the answers to the actual probability using $M = 1000$ repeats of the experiment, gives the results shown in Table 5.4. Increasing the number of repeats improves the accuracy.

Table 5.4: Theoretical and simulated probabilities for the Birthday problem.

People N	Theoretical	Simulated
5	0.027	0.023
10	0.117	0.114
15	0.253	0.232
20	0.411	0.422
22	0.476	0.476
23	0.507	0.511
25	0.569	0.579
35	0.814	0.821
45	0.941	0.949
55	0.986	0.985

Exercise 5.13: Use the above code to verify that increasing the number of repeats of this experiment, improves the accuracy of the answers given by the simulation.

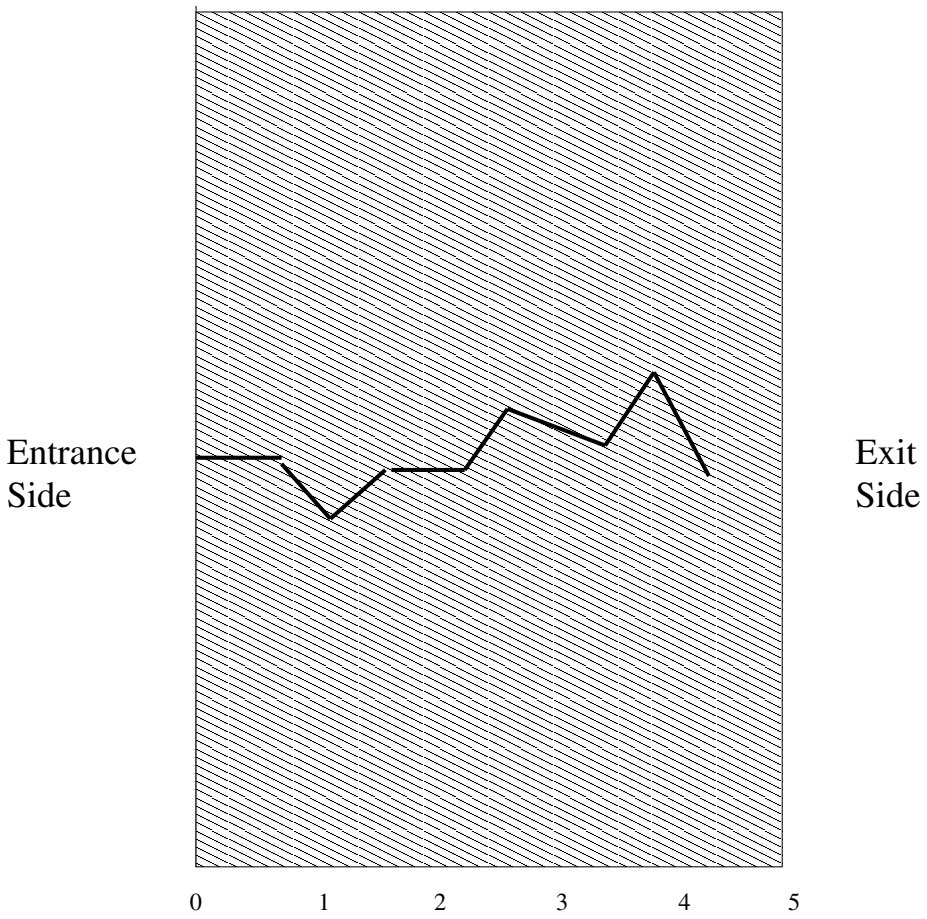


Figure 5.8: Graphical illustration of path followed by neutron.

5.4.3 Neutron penetration problem (A random walk problem)

The final example of Monte Carlo simulation that we explore is the Neutron Penetration problem. In this problem neutrons are fired at some lead shielding, and we are interested in how many neutrons finally end up penetrating the shield. Now we could construct an experiment that would give us the desired answer, but this would be expensive and costly to do. However, if we can capture neutron behaviour in terms of a Monte Carlo simulation, then we can solve this problem with minimum cost and time.

How are we going to capture the behaviour of a neutron as it passes through a lead shield? To answer this, let us consider what happens to a neutron as it passes though the shield. The neutron will happily continue on its path until it hits an atom contained within the shield (see Figure 5.8), once this happens the neutron will be diverted off its original path. It will than continue again in a straight line in its new direction until it hits another atom and is redirected again. This will continue until the neutron has run out of energy. This is the behaviour that we have to capture in our simulation. In constructing our simulation we make a simplifying assumption that the particle travels exactly 1 unit before each collision and that it only has the energy to have a certain number of collisions. By doing this we simplify the calculations involved in each step.

Assuming that a particle can only have so many collisions accounts for the loss of energy of the particle. Now it only remains to account for how the particle moves through the shield. To simplify we assume that the only way the neutron can exit the shield is from either the left or right hand side; that is, we assume that the shield is infinite in the vertical y direction. This means that we need only track the coordinate of the neutron in the horizontal x direction. Now we need to simulate what happens at the collision. To do this we generate a random number between 0 and 2π, which will be the angle θ the particle is deflected after collision. This gives us the new direction that the particle moves. Since the particle moves only a unit length due to our earlier assumption the total x direction it covers after this collision is $\cos\theta_1$. So the total distance covered after the first collision is $1+\cos\theta_1$.

We then check to see if the particle has exited the shield, if it has exited we record which side it exited on and move onto the next particle. However, if it remains in the shield, we repeat the above steps to get another collision, and so on until either the number of collisions is exhausted or the particle has exited the shield, which ever happens first.

We then calculate the proportion penetrating the wall by counting the number of particles that penetrated the shield and dividing it by the total number of particles fired at the shield.

Algorithm 24 implements this simple simulation for shields of different thicknesses. Execute it with, for example, the statement `neutron(50,5,1000)` to simulate 1000 neutrons trying to get through a shield of thickness 5 in no more than 50 collisions: find that only 21% of the neutrons get through.

Note: by simple modifications to the code, the assumptions above can be removed and the simulation improved.

Exercise 5.14: Experiment using Algorithm 24 with the effect of increasing/decreasing the number of collisions possible and the width of the shield. Do these results reflect what you would expect to happen from the physics of the problem. Note: you will need to be systematic in your approach to this problem to get the desired results.

Simulation Monte Carlo simulation is the main tool explored in depth in the course CSC3409 *Simulation* as simulation has enormous use in finance, business and industrial problems. Some financial applications and the underlying mathematics are explored in the course MAT3104 *Random processes to financial mathematics*.

Intelligent agents This last problem of a random walk explicitly is phrased in terms of an individual, here a neutron, as it evolves and interacts with its environment, here its simple progression across a slab. Many people similarly simulate systems made up of many individuals, ranging from dumb particles to sentient animals. They code up rules of interaction between the individuals (also see Exercise 5.17 for another simple example). The coding usually involve randomness either through explicit invocation of random numbers, as we have done here, or through intrinsic randomness generated by the chaos of individual interactions (see the course MAT8102 *Chaos*). Executing the code empowers simulation of complex systems. Such techniques are often called grand names such as *Agent Based*

Algorithm 24 simulate the penetration of neutrons. The `find` command empowers vectorisation of conditional loops: here we stop simulating neutrons that escape the material through either end.

```
%  Function :  neutron
%  Syntax   :  p = neutron(coll,wid,N)
%  Inputs   :  coll = number of collisions allowed
%              wid = width of the lead shield
%              N = number of neurons incident on shield
%  Outputs  :  p = percentage to escape through the shield
%  Algorithm:  This function calculates the percentage of
%              neutrons that emerge from a lead shield of
%              various widths and with a given number of
%              restrictions on the number of collisions
%              possible.
%  Source:     H.Butler 29/2/2000
%              Last Modified: Tony Roberts, 2007
%              Modified Harry Butler, Jan 2017
function [p] = neutron(coll,wid,N)
  p = []; % If function fails output is empty
  % Data checking %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  if min(coll,wid,N) <=0
    error('neutron: parameters must be positive');
  end;

  % Counting collisions %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  x=ones(1,N);  % start at position 1
  n=ones(1,N);  % count the number of steps
  for count=2:coll % take up to coll steps
      j=find((x>=0)&(x<=wid)); % find those not escaped
      x(j)=x(j)+cos(2*pi*rand(1,length(j)));
      n(j)=n(j)+1;
  end
  p = 100*sum(x>wid)/N; % percentage got through
end
```

Simulation or even more grandly *Intelligent Agents*. Frankly, I fail to see what is particularly intelligent about a Monte Carlo simulation executed through even a few thousand lines of code. Nonetheless, Monte Carlo simulation is what these names mean.

5.4.4 Exercises

Ex. 5.15: Suppose that we have a loaded die. If the probabilities associated with the six faces are as shown below:

Outcome	1	2	3	4	5	6
Probability	0.2	0.14	0.22	0.16	0.17	0.11

(a) Write and run a program to simulate 1500 throws of such a die.

(b) Consider a pair of dice loaded as above. Modify your program above to simulate rolling these two dice. Use this new program to calculate via simulation the probability of throwing a 12 in 25 throws of such a dice.

Ex. 5.16: Write a program that simulates drawing 4 cards from a pack of 52 cards. Use this to estimate the probability of drawing 4 aces. Note the exact probability of doing this is: $\frac{4}{52} \times \frac{3}{51} \times \frac{2}{50} \times \frac{1}{49}$.

Ex. 5.17: The *Drunkard's walk problem*: An intoxicated person lives 10 blocks from his local pub, which is in the same street as their home (see Figure 5.9). Our drunk starts at block n, where $2 \leq n \leq 9$ and wanders at random one block at a time, either directly towards the pub or their home. At any intersection, he moves towards the pub with the probability 2/3 and towards home with the probability 1/3. Having reached either the pub or home they remain there. Use simulation to calculate the number of times in 1000 trips in which they start at block 6, that they end up at the pub and the average number of blocks that they have walked. Repeat this for the other starting blocks.

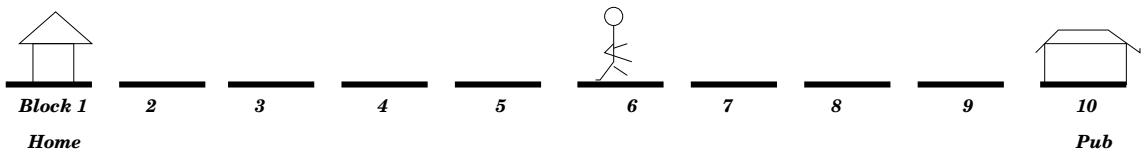


Figure 5.9: Illustration of the drunkards walk

5.5 Cache limits vector lengths

Monte Carlo simulations need vast numbers of repetitions in order to achieve quantitative accuracy. For example, the neutron scattering problem needs about 100,000 simulations to get the percentage of penetrating neutrons correct to a tenth of one percent. Simulations with such vast vectors often incurs cache conflicts that slow down processing. This section introduces the benefits and limitations of cache.

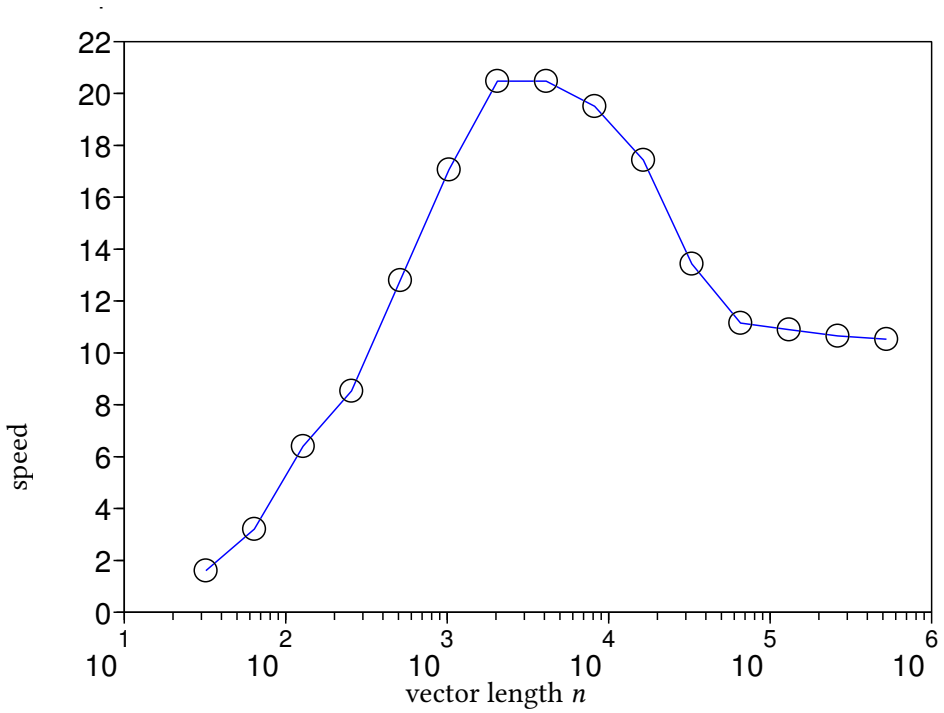


Figure 5.10: speed of simulation per element for various vector lengths.

Example 5.18: Neutron penetration Revisit the neutron penetration problem of Section 5.4.3. But now repeat the calculation for various vector lengths, various numbers of realisations. Algorithm 25 invoke the earlier function with vector lengths that are various powers of two, from 32 to nearly a million. The algorithm times how long the computations take and then plots the time per simulation.

Algorithm 25 repeat the neutron scattering for various numbers of simulations.

```
ns=2.^(5:19);
ts=nan*ns;
for n=ns
    tic
    [n neutron(20,5,n)]
    ts(find(n==ns))=toc();
end
mrate=ns./ts/1e4;
semilogx(ns,mrate,'o-')
```

Figure 5.10 shows that as the vector gets longer, the speed increases, at least up to vector lengths $n \approx 1000$. Then for vector lengths n roughly $1,000 < n < 10,000$ the speed is about its constant maximum. However, for even longer vector lengths, $n > 10,000$ roughly, the speed decreases. That is, remarkably, the program becomes less efficient.

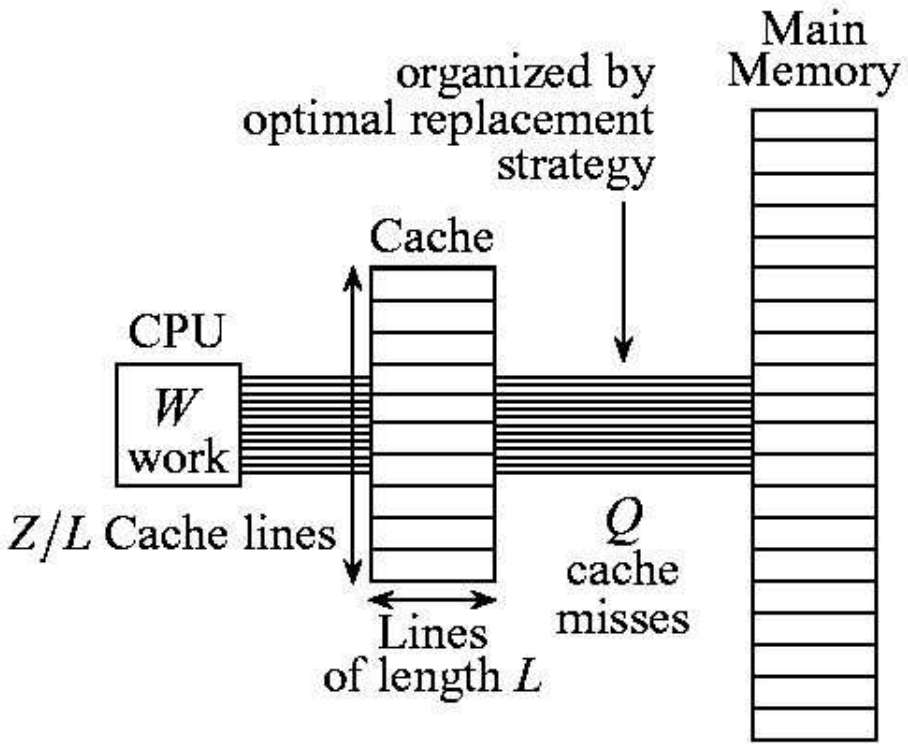


Figure 5.11: a cache lies between main memory and the CPU: actually most modern computers have multiple layers of cache!

Why? Why does the efficiency of the program deteriorate so markedly for very long vectors? Surely it should keep its efficiency. The reason is that the fast cache memory inside the computer limits the effective vector lengths.

Recall that cache lies between the CPU and the main memory, see Figure 5.11. The cache is a quick temporary memory between the fast CPU and the slow main memory. While computed numbers can stay within cache, the numbers are efficiently provided to the CPU for the computations. This is the case here for vectors up to lengths of about $10,000$; they fit in cache and are provided to the CPU without significantly involving main memory.

But when vector lengths get too long, here longer than about $10,000$, the vectors of numbers no longer fit into cache. Hence, the program has to store some of the numbers out into main memory while it deals with other numbers. Later, when the computation needs those numbers again, it has to fetch them from main memory. This storing and fetching of numbers to the main memory is done at the maximum rate that the main memory (and the bus) can sustain. However, this maximum rate is significantly less than that of cache and the CPU. Hence the whole computation slows down markedly.

Block computation for efficiency One way to avoid such cache limitations is to '*block*' your computations. That is, whenever your vectors are bigger than about $10,000$,[2] break your computation into many independent chunks where each chunk only in-

[2] This number depends upon the computer hardware.

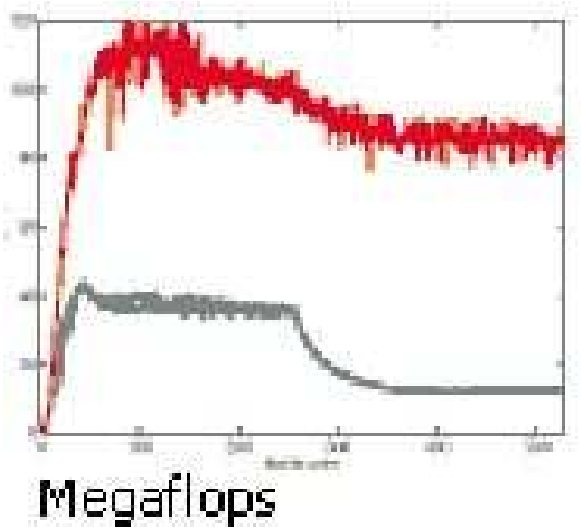


Figure 5.12: very fuzzy picture of the rate of computation versus vector size in MATLAB: bottom grey line, old MATLAB is slower; upper redline, new MATLAB is faster utilising LAPACK and BLAS.

volves vectors whose lengths are in the thousands where cache and CPU work at peak efficiency. Then you will ensure that the computation is done as fast as possible.

Leading numericists use blocking. One of the oldest open source projects utilising the net is the world leading effort to provide computational scientists with effective libraries of routines to compute effectively. Public domain libraries such as LAPACK and ATLAS are built upon BLAS routines: these routines use blocking in order to deliver reliable peak performance to all, for free; for example, see `http://www.netlib.org/lapack/lug/`.

MATLAB used to use its own routines at the heart of its software for computations. The computation rate is indicated in the grey line of Figure 5.12: see the classic flat high performance region for short to medium vectors; and the classic fall off in speed once vectors get too long to fit into cache. Some years ago they changed and instead implemented the combination of LAPACK and BLAS. With the effective blocking techniques in this public domain open source software, MATLAB immediately sped up by at least a factor of two, the red line in Figure 5.12, and also could work with cache to maintain much of that performance even for very long vectors. For peak performance use the routines of this public domain project.

5.6 Summary and key formulae

- For integrals in one dimension, a *Monte Carlo* approximation is obtained by estimating the function average with *randomly generated points*

$$\int_a^b f(x)dx \approx \frac{b-a}{N}\sum_{i=1}^{N} f(x_i)\,.$$

The method, with an error of order $N^{-1/2}$, is normally not competitive with standard methods in one dimension but is a viable option in higher dimensions.

- Monte Carlo methods have strengths and weaknesses.

  Strengths:

  - they are quick to code;
  - they are usually quick to get low accuracy approximations to the required answer.

  Weaknesses:

  - they need an awful lot of computation to get accurate answers;
  - it is easy to make errors in coding that are very difficult to detect.

- Cache limits the maximum length of vectors you should use. Currently, the maximum vector length is about $10,000$ on desktop workstations. Use blocking to efficiently use cache.

Chapter 6 Forecast the future with ordinary differential equations

Chapter contents

6.1 Introduction

Differential equations model an enormously varied range of phenomena in many applications areas in physics, chemistry, biology, sociology, engineering and games. We must solve differential equations to forecast in such applications.

- A simple pendulum is governed by the nonlinear differential equation

$$\frac{d^2\theta}{dt^2} + \frac{g}{l}\sin\theta = 0$$

where $\theta(t)$ is the angular displacement of the pendulum as it evolves in time, g is the constant acceleration due to gravity, and l is the constant length of the pendulum.

- Small displacements of a beam under axial compression and oscillatory transverse loading are governed by the differential equation

$$\frac{d^2y}{dt^2} + \mu\frac{dy}{dt} - \pi^2(\Gamma - \pi^2)y + \frac{1}{2}K\pi^4y^3 = A\cos\omega t$$

where $y(t)$ is the displacement amplitude, Γ is the axial load, μ is the vibration damping coefficient, K is a stiffness measure of the beam, whereas A and ω are the amplitude and frequency of the applied oscillatory load.

- Human heartbeats are modelled by the pair of coupled differential equations

$$\begin{aligned} \epsilon\frac{dx}{dt} &= -(x^3 + ax + b)\,, \\ \frac{db}{dt} &= x - x_a\,, \end{aligned}$$

where $x(t)$ is the time dependent muscle fibre length, and with constants ϵ, a is the tension in the muscle, b is the electrochemical control that stimulates the muscle and x_a is the relaxed muscle fibre length.

These examples from physics, engineering and biology are a tiny sample of the vastly diverse array of real applications for differential equations. To study these phenomena requires solutions to the equations. Solutions expressible in terms of the elementary functions are extremely rare. Various clever analytical techniques have been developed to reveal the properties of these solutions but, at the most basic level, the requirement is to find the function's *values* for various inputs. This module introduces some of the basic techniques and concepts.

General objectives for this module

- To gain a working knowledge and understanding of some commonly used numerical methods for solving initial value problems for ordinary differential equations
- To apply these methods for computing solutions to first order initial value problems both for a single equation and for a system of equations.

Prerequisite knowledge

- First order differential equations
- Taylor series approximations

6.2 Solution of initial value problems

If the position and velocity of a moving ball, pendulum or satellite, are known at some instant of time, then the entire motion of the projectile can be forecast from that instant, given the appropriate (possibly highly complicated) dynamic laws of motion. This is an *Initial Value Problem* (IVP), because the data at some initial time, or initial values, along with the differential equations of motion determine the position and velocity for all later times.

Specific objectives

- Solve IVP's by Euler, Modified Euler, and fourth order Runge–Kutta methods
- Understand the importance of steplengths with regard to accuracy and stability

6.2.1 Euler's method

Example 6.1: Newton's law of cooling provides a model for the cooling of a hot object. Newton's law states that the rate of change of the object's temperature T is proportional to the difference in temperature between it and the surroundings:

$$\frac{dT}{dt} = k(T - T_a),$$

where t is time, k is a negative constant, and T_a is the surrounding, or ambient, temperature. This equation, along with the object's initial temperature T_0 defines an *Initial Value Problem* which is to be solved for the temperature variation with time.

For simplicity take $T_0 = 1$, $k = -1$ and zero ambient temperature T_a so that the IVP becomes

$$\frac{dT}{dt} = -T\,, \quad T(0) = 1\,,$$

for which you may know the analytic solution is $T(t) = e^{-t}$. Here we seek a numerical solution based on the following principle: if the (approximate) temperature is known at any particular time then the *rate of change* of temperature at that time can be calculated from the differential equation. This rate then estimates the temperature *change* that would occur in a small time interval, h. The temperature change plus the initial temperature is an estimate for the new temperature h seconds later.

At the starting time $t = 0$ the temperature T is 1 and the rate of change of temperature at that time is

$$\frac{dT}{dt}(0) = -T(0) = -1\,.$$

Now let h be a small time interval. Approximating the derivative in the above expression by the forward difference

$$\frac{dT}{dt} \approx \frac{T(h) - T(0)}{h}\,,$$

then, since we know $dT/dt = -1$ at $t = 0$, we have

$$\frac{T(h) - T(0)}{h} \approx -1\,,$$

which upon rearranging gives

$$T(h) \approx T(0) - h = 1 - h\,,$$

which is an estimate for the temperature at time h. Choosing time step $h = 0.1$ gives the approximate value $T(0.1) \approx 0.9$.

We just completed one step of *Euler's method*, a simple but effective procedure for the numerical solution of ODE's. Applying the same process at $t = 0.1$ gives an approximate solution at $t = 0.2$:

$$T(0.2) \approx T(0.1) + h\frac{dT}{dt}(0.1) = 0.9 + h(-0.9) = 0.81\,.$$

In general, carry out a step from the approximate solution at (t_n, T_n) to the next approximate solution (t_{n+1}, T_{n+1}) by applying a forward step of length h in the derivative approximation. Thus

$$\frac{dT}{dt}(t_n, T_n) \approx \frac{T_{n+1} - T_n}{h}\,,$$

which upon rearrangement gives the approximation formula

$$T_{n+1} = T_n + h\frac{dT}{dt}(t_n, T_n)\,.$$

Now since the differential equation is $\frac{dT}{dt} = -T$, we get the rule to compute temperature T at time t_{n+1} to be

$$T_{n+1} = T_n - hT_n = (1-h)T_n\,.$$

Figure 6.1 plots the exact solution e^{-t} alongside two Euler computed solutions with $h = 0.2$ and $h = 0.8$, indicating a more accurate result with the smaller stepsize. As the stepsize is increased the method not only loses accuracy but will also become *unstable* when the time step h becomes large enough, giving unbounded, meaningless computed results. For example, taking step[1] $h = 5$ gives $T_1 = -4\,,\ T_2 = 16\,,\ T_3 = -64\,,\ T_4 = 256\,,$ ... which rapidly grows out of control.

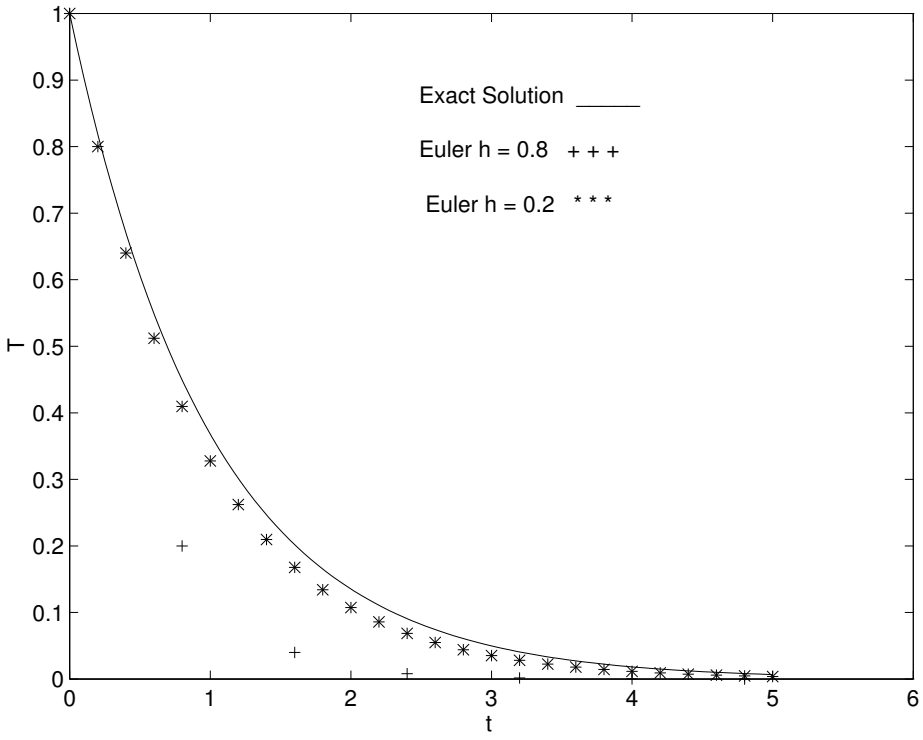


Figure 6.1: Euler computed solutions with two different step lengths for the cooling example, showing the improvement in accuracy produced by a smaller step size.

In general For the general IVP[2]

$$y' = \frac{dy}{dt} = f(t,y)\,, \quad y(t_0) = y_0\,,$$

where $f(t,y)$ is some specified function. Derive Euler's method by considering a Taylor series for $y(t+h)$ about t:

$$y(t+h) = y(t) + hy'(t) + \frac{1}{2}h^2y''(t) + \cdots\,.$$

[1] A pathological case for illustration purposes.

[2] In *autonomous* systems the independent variable t does not appear in f.

Truncating this after the linear, or first order terms gives the tangent approximation

$$y(t+h) \approx y(t) + hy'(t) = y(t) + hf(t,y)\,,$$

or for a typical step of the numerical solution

$$y_{n+1} = y_n + hf(t_n, y_n)\,,$$

where $t_n = t_0 + nh$ and $y_n = y(t_n)$. This is an *explicit formula* for y_{n+1} since all quantities on the right hand side are known from the previous step.

However, if a backward difference derivative approximation is used the method becomes *implicit*

$$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1})\,,$$

since the unknown y_{n+1} appears on both sides of the equation and if $f(t,y)$ is non-linear each step would require a nonlinear solver, for example, Newton's method. We only concern ourselves with explicit methods here.

Error and stability The error incurred at a given Euler step or *local truncation error*, is second order since from the Taylor series

$$y(t+h)_{\text{exact}} - y(t+h)_{\text{Euler}} = \frac{1}{2}h^2 y''(t) + \cdots$$

and so the size of the error is critically determined by the small amount h^2, since h is small. If the multiplier of the first error term is h^k, we say that the error is of order k. Note that since h is the time step size, then $1/h$ is a measure of how many steps you take. So if at each time step the error is of order h^k then you end up with an error of order $(h^k) \times \frac{1}{h} = h$. In the above case, the local errors of order 2 (that is, a multiple of h^2) accumulate over a number of steps, producing an accumulated error of order 1 (that is, a multiple of h) after $1/h$ steps.

Another important consideration regarding steplength is that of stability. In the example above the typical Euler step takes the form

$$T_{n+1} = (1-h)T_n = (1-h)(1-h)T_{n-1} = (1-h)^3 T_{n-2} = \cdots = (1-h)^n T_0\,.$$

From this power law see that if $0 < h < 1$, the sequence converges to zero and so is *stable*. For example if $h = 0.9$ and $T_0 = 1$, then $T_{n+1} = (0.1)^n$ which gives the sequence 0.1, 0.01, 0.001, ... , which is converging towards zero and so is stable. For $1 < h < 2$ the method would produce a positive/negative oscillatory solution which also converges to zero and so is stable. For $h = 2$, the sequence of T values is then $T_{n+1} = (-1)^n T_0$ which just oscillates from $+T_0$ to $-T_0$ for $n = 0, 1, 2, 3, \ldots$ and this is said to be unstable since it is not settling down towards a singe answer. For $h > 2$ The sequence oscillates with ever growing size which is of course unstable. This *instability* occurs in any explicit method if the time step is too large.

6.2.2 The improved Euler method

Euler's method applied to the differential equation $y' = f(t)$ is equivalent to taking left hand Riemann sums as an approximation to the integral $\int_a^b f(t)\,dt$ which is a

solution to the problem. This is a very poor approximation. Each step has errors of order h^2 and thus overall the accumulated error is of order h. Good numerical methods have accumulated errors of order h^2 or better.

The *Improved Euler's Method*, sometimes called *second order Runge–Kutta* or RK2, achieves accumulated errors proportional to h^2. Figure 6.2 shows the geometry for the first step of the Improved Euler method. The tangent line is followed to a new point (t_{n+1}, y^*_{n+1}) but do not record this Euler point. Instead compute the slope at this point; the average of the slope here and at the previous point is used for a straight line approximation that delivers the next approximation point. The improved Euler technique is described as a *predictor-corrector* technique.

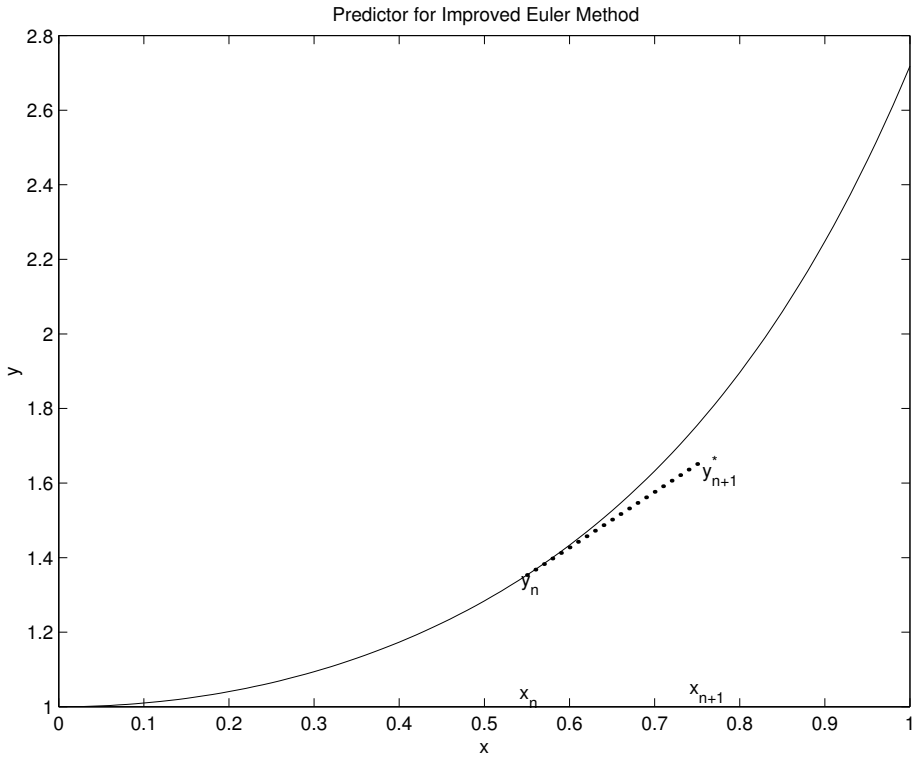


Figure 6.2: Showing the tangent line for the first step of Euler's improved method.

The predictor-corrector formulation is firstly to compute the prediction

$$y^*_{n+1} = y_n + hf(t_n, y_n);$$

secondly, compute the correction

$$y_{n+1} = y_n + h\frac{\left[f(t_n, y_n) + f(t_{n+1}, y^*_{n+1})\right]}{2}.$$

This procedure is sometimes more conveniently written as

$$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_{n+1}, y_n + k_1), \\ y_{n+1} &= y_n + \frac{1}{2}(k_1 + k_2). \end{aligned}$$

Although this form hides somewhat the averaging of the slopes at (t_n, y_n) and (t_{n+1}, y^*_{n+1}), it generalises to the next powerful and commonly used method.

In general As we did with Euler's method, the Improved Euler method can be derived using a Taylor series expansion. Again considering a Taylor series for $y(t+h)$ about t:

$$y(t+h) = y(t) + hy'(t) + \frac{1}{2}h^2y''(t) + O(h^3). \tag{6.1}$$

we would like to improve Euler's method by including the h^2 term in the method—not just dropping it. One way we can do that is to look at the Taylor series for $y'(t+h)$ about t:

$$y'(t+h) = y'(t) + hy''(t) + \frac{1}{2}h^2y'''(t) + O(h^3),$$

this series allows us to write $y''(t)$ in terms of y', thus:

$$y''(t) = \frac{y'(t+h) - y'(t)}{h} - \frac{1}{2}hy'''(t) + O(h^2),$$

the error term must also be divided by h.

Replacing y'' by the Taylor expansion in Equation 6.1 above, we get—

$$y(t+h) = y(t) + hy'(t) + \frac{1}{2}h(y'(t+h) - y'(t)) + O(h^3),$$

or

$$y(t+h) = y(t) + \frac{1}{2}h(f(t+h) + f(t)) + O(h^3),$$

the Improved Euler method with an error at each step of $O(h^3)$.

Example 6.2: temperature decay Return to Example 6.1 and apply the Improved Euler method to

$$\frac{dT}{dt} = -T \quad \text{such that} \quad T(0) = T_0\,.$$

Solution: First the predictor $T^*_1 = T_0 + hf(T_0) = T_0 + h(-T_0) = (1-h)T_0$. Then the corrector

$$\begin{aligned} T_1 &= T_0 + h\frac{(-T_0) + (-T^*_1)}{2} \\ &= T_0 - \frac{h}{2}[T_0 + (1-h)T_0] \\ &= (1 - h + \tfrac{1}{2}h^2)T_0\,. \end{aligned}$$

Similarly, the second time step is first to compute the predictor $T^*_2 = T_1 + hf(T_1) = T_1 + h(-T_1) = (1-h)T_1$. Then the corrector

$$\begin{aligned} T_2 &= T_1 + h\frac{(-T_1) + (-T^*_2)}{2} \\ &= T_1 - \frac{h}{2}[T_1 + (1-h)T_1] \end{aligned}$$

$$= (1 - h + \tfrac{1}{2}h^2)T_1$$
$$= (1 - h + \tfrac{1}{2}h^2)^2 T_0 ,$$

upon using our expression for T_1. Similarly, the third time step computes $T_3 = (1 - h + \frac{1}{2}h^2)^3 T_0$. And so on for n time steps to compute the numerical solution $T_n = (1 - h + \frac{1}{2}h^2)^n T_0$.

This power law solution more matches the analytic solution, $T(t) = T_0 e^{-t}$, to second order because the Taylor series for $e^{-h} = 1 - h + \frac{1}{2}h^2 - \frac{1}{6}h^3 + \cdots$ matches the factor $(1 - h + \frac{1}{2}h^2)$ to a local error of order 3 (that is, proportional to h^3) and so after $n = 1/h$ steps will have an accumulated error of order 2.

Example 6.3: Code into MATLAB the improved Euler method to solve for $1 < t < 2$ the differential equation

$$\frac{dy}{dt} = ty^2 \quad \text{such that} \quad y(1) = 2/3 .$$

Algorithm 26 gives script that computes the following numerical estimates as plotted in Figure 6.3.

t_n	$y(t_n)$
1.0000	0.6667
1.1000	0.7167
1.2000	0.7808
1.3000	0.8648
1.4000	0.9782
1.5000	1.1380
1.6000	1.3771
1.7000	1.7688
1.8000	2.5111
1.9000	4.3416
2.0000	12.4097

Comment on the comparison with the exact solution $y(t) = 2/(4 - t^2)$.

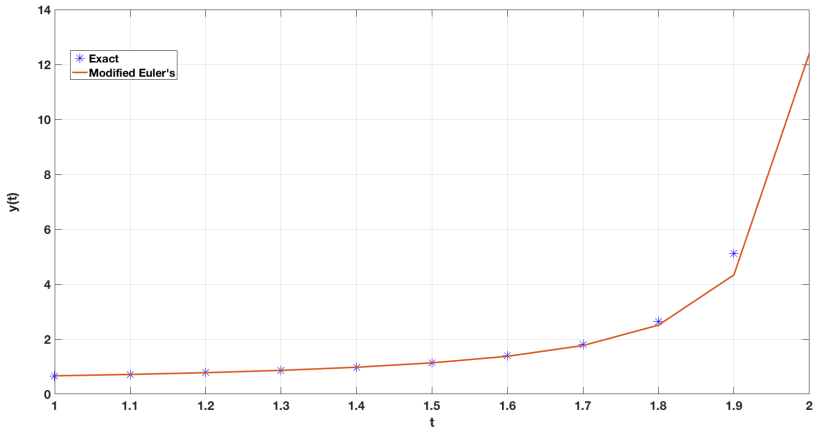


Figure 6.3: improved Euler solution of $dy/dt = ty^2$ with $y(1) = 2/3$ using $N = 10$ time steps.

Algorithm 26 improved Euler script to solve $dy/dt = ty^2$ with $y(1) = 2/3$ using $N = 10$ time steps.

```
% Solve dy/dt=ty^2 with y(1)=2/3 on 1<t<2.
% Use improved Euler method with N time steps.
% Tony Roberts Feb 2008
% Modifed by H Butler 2015
% Modified Harry Butler 2017
N=10
t=linspace(1,2,N+1)';
y=zeros(N+1,1);
h=diff(t(1:2));
y(1)=2/3;
for n=1:N
  k1=h*t(n)*y(n)^2;
  y(n+1)=y(n)+k1;
  k2=h*t(n+1)*y(n+1)^2;
  y(n+1)=y(n)+(k1+k2)/2;
end
[t y]
```

Vectorisation these algorithms to integrate forward in time cannot be vectorised. They are inherently slow. The reason is that one generally has to compute y_n before starting to compute y_{n+1}. The computations cannot be overlapped. The only vectorisation possible is when you have a problem with many components, that is the unknown y is a vector, and then one must try to vectorise over the components of y.

6.2.3 The Runge–Kutta RK4 Method

More sophisticated averaging procedures are incorporated in 'higher order' *Runge–Kutta* techniques. Their derivation is similar to the Improved Euler method deriva-

tion using Taylor series extensively.

The improved Euler method is a simple and effective numerical ODE solver, always useful as a 'first stab' to get a feel for the behaviour of a certain solution. Once this is achieved, perform more refined calculations performed using advanced techniques of higher accuracy, one of which is the fourth order Runge–Kutta scheme (RK4), derived by matching Taylor series terms up to fourth order. Note that the error at each step of the RK4 method is of order h^5, so that the overall accumulated error after about $1/h$ steps is of order h^4.

For the IVP $\frac{dy}{dt} = f(t, y)$, a single RK4 step to advance the solution from (t_n, y_n) to $(t_n + h, y_{n+1})$ is the following five computations:

$$\begin{aligned} k_1 &= hf(t_n, y_n)\,, \\ k_2 &= hf(t_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1)\,, \\ k_3 &= hf(t_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_2)\,, \\ k_4 &= hf(t_n + h, y_n + k_3)\,, \\ y_{n+1} &= y_n + \tfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)\,. \end{aligned}$$

More calculation is required than Euler's methods: there are now four function evaluations per step to achieve the improvement in accuracy.

Here is an idea of the geometry behind the RK4 method. Remember that the corrector in the Improved Euler method used information from the slope of the curve at the end of the tangent line at t_{n+1}. The RK4 method improves on this by additionally using corrector values half way to this point. Hence correctors are calculated using the tangent line as far as $t_n + h/2$ to get the slopes of the curve. You can see this information being used in the calculation of k_2 and k_3. Then calculate k_4 using the full step to $t_{n+1} = t_n + h$. The final calculated value of y_{n+1} is the sum of the previous value of y_n and a weighted average which uses the slopes across the intervals which contribute to the k values.

Example 6.4: Use a steplength of $h = 0.1$ to carry out two RK4 steps on the IVP

$$\frac{dy}{dt} = ty^2\,, \quad y(1) = 2\,.$$

Step 1

$$\begin{aligned} k_1 &= hf(t_0, y_0) \\ &= ht_0y_0^2 \\ &= 0.1 \times 1 \times 4 = 0.4 \end{aligned}$$

$$\begin{aligned} k_2 &= hf(t_0 + \tfrac{1}{2}h, y_0 + \tfrac{1}{2}k_1) \\ &= h(t_0 + \tfrac{1}{2}h)(y_0 + \tfrac{1}{2}k_1)^2 \\ &= 0.1(1.05 \times 2.2^2) = 0.5082 \end{aligned}$$

$$k_3 = hf(t_0 + \tfrac{1}{2}h, y_0 + \tfrac{1}{2}k_2)$$

$$\begin{aligned} &= h(t_0 + \tfrac{1}{2}h)(y_0 + \tfrac{1}{2}k_2)^2 \\ &= 0.1(1.05 \times 2.2541^2) = 0.5335 \end{aligned}$$

$$\begin{aligned} k_4 &= hf(t_0 + h, y_0 + k_3) \\ &= h(t_0 + h)(y_0 + k_3)^2 \\ &= 0.1(1.1 \times 2.5335^2) = 0.7060 \end{aligned}$$

$$\begin{aligned} y_1 &= y_0 + \tfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\ &= 2 + (0.4 + 2(0.5082) + 2(0.5335) + 0.7060)/6 \\ &= 2.5316\,. \end{aligned}$$

Step 2

$$\begin{aligned} k_1 &= hf(t_1, y_1) \\ &= ht_1y_1^2 \\ &= 0.1 \times 1.1 \times 2.5316^2 = 0.7050 \end{aligned}$$

$$\begin{aligned} k_2 &= hf(t_1 + \tfrac{1}{2}h, y_1 + \tfrac{1}{2}k_1) \\ &= h(t_1 + \tfrac{1}{2}h)(y_1 + \tfrac{1}{2}k_1)^2 \\ &= 0.1 \times 1.15 \times 2.8841^2 = 0.9565 \end{aligned}$$

$$\begin{aligned} k_3 &= hf(t_1 + \tfrac{1}{2}h, y_1 + \tfrac{1}{2}k_2) \\ &= h(t_1 + \tfrac{1}{2}h)(y_1 + \tfrac{1}{2}k_2)^2 \\ &= 0.1(1.15 \times 3.0099^2) = 1.0418 \end{aligned}$$

$$\begin{aligned} k_4 &= hf(t_1 + h, y_1 + k_3) \\ &= h(t_1 + h)(y_1 + k_3)^2 \\ &= 0.1(1.2 \times 3.5734^2) = 1.5323 \end{aligned}$$

$$\begin{aligned} y_2 &= y_1 + \tfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\ &= 2.5316 + (0.7050 + 2(0.9565) + 2(1.0418) + 1.5323)/6 \\ &= 3.5706\,. \end{aligned}$$

The exact solution is $y = 2/(2 - t^2)$, giving $y(1.2) = 3.5714$ and so the absolute error is 0.0008 and the relative error is $\frac{3.5714-3.5706}{3.5714} = 0.00022$, or a percentage error of only 0.022% in the numerical value.

From the discussion in the preceding section a theme emerged: namely, more accuracy costs more work. For a given step size RK4 is much more accurate than Euler or modified Euler but more function evaluations are needed per step. The choice becomes a trade-off between work required and accuracy desired.

6.3 Second order equations and first order systems

6.3.1 Extension to systems of ODEs

This previous section shows how to extend the Runge–Kutta scheme in a natural way in order to solve complex initial value problems, complex because of many interacting componennts. Second order differential equations are more common in practice and so this section shows you the way forward by introducing just two interacting components.

The scalar differential equations used to introduce the numerical methods in the previous section are just special cases of the more general system of differential equations

$$\frac{d\boldsymbol{y}}{dt} = \boldsymbol{f}(t, \boldsymbol{y}),$$

in which $\boldsymbol{y}$ and $f(t, \boldsymbol{y})$ are vectors. Applying the methods to systems is exactly the same, except that scalar quantities are now replaced with vector quantities.

Example 6.5: A pair of coupled differential equations modelling the human heartbeat is written in vector form as

$$\frac{d\boldsymbol{y}}{dt} = \frac{d}{dt}\begin{bmatrix} u \\ v \end{bmatrix} = \boldsymbol{f}(\boldsymbol{y}) = \begin{bmatrix} -\alpha(u^3 - \beta u + v) \\ u - \gamma \end{bmatrix},$$

where u represents a muscle fibre length, v is the electrochemical control and α, β and γ are constants. To apply the previous methods to this system simply treat $\boldsymbol{y} = (u, v)$ as a single (vector) variable.

For $\alpha = 80$, $\beta = 2$, $\gamma = 1.2$ and initial condition

$$\boldsymbol{y}(0) = \begin{bmatrix} u_0 \\ v_0 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 0.5 \end{bmatrix}$$

one step of Euler and RK4 will be carried out, using steplength $h = 0.01$.

For Euler

$$\boldsymbol{y}_1 = \boldsymbol{y}_0 + h\boldsymbol{f}(\boldsymbol{y}_0) = \begin{bmatrix} 1.0 \\ 0.5 \end{bmatrix} + 0.01 \begin{bmatrix} -\alpha(1 - \beta + 0.5) \\ 1 - \gamma \end{bmatrix} = \begin{bmatrix} 1.4 \\ 0.4980 \end{bmatrix}.$$

For RK4

$$\begin{aligned} \boldsymbol{k}_1 &= h\boldsymbol{f}(\boldsymbol{y}_0) \\ &= \begin{bmatrix} 0.4 \\ -0.0020 \end{bmatrix}, \end{aligned}$$

$$\begin{aligned} \boldsymbol{k}_2 &= h\boldsymbol{f}(\boldsymbol{y}_0 + \tfrac{1}{2}\boldsymbol{k}_1) \\ &= h\begin{bmatrix} -\alpha(1.2^3 - \beta \times 1.2 + 0.4990) \\ 1.2 - \gamma \end{bmatrix} = \begin{bmatrix} 0.1384 \\ 0 \end{bmatrix}, \end{aligned}$$

$$\boldsymbol{k}_3 = h\boldsymbol{f}(\boldsymbol{y}_0 + \tfrac{1}{2}\boldsymbol{k}_2)$$

$$= h\begin{bmatrix} -\alpha(1.0692^3 - \beta \times 1.0692 + 0.5) \\ 1.0692 - \gamma \end{bmatrix} = \begin{bmatrix} 0.3329 \\ -0.0013 \end{bmatrix},$$

$$\begin{aligned} \boldsymbol{k}_4 &= h\boldsymbol{f}(\boldsymbol{y}_0 + \boldsymbol{k}_3) \\ &= h\begin{bmatrix} -\alpha(1.3329^3 - \beta \times 1.3329 + 0.4987) \\ 1.3329 - \gamma \end{bmatrix} = \begin{bmatrix} -0.1607 \\ 0.0013 \end{bmatrix} \end{aligned}$$

$$\begin{aligned} \boldsymbol{y}_1 &= \boldsymbol{y}_0 + \frac{1}{6}(\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4) \\ &= \begin{bmatrix} 1.1970 \\ 0.4995 \end{bmatrix}. \end{aligned}$$

Convert higher order differential equations to systems of first order equations and solved by the above methods.

Example 6.6: Consider the pendulum equation

$$\frac{d^2\theta}{dt^2} + \frac{g}{l}\sin\theta = 0,$$

which is a second order nonlinear differential equation. Write $u = \theta$ and $v = \frac{d\theta}{dt} = \dot{\theta}$, so that

$$\frac{du}{dt} = \frac{d\theta}{dt} = v,$$

and

$$\frac{dv}{dt} = \frac{d\dot{\theta}}{dt} = \frac{d^2\theta}{dt^2},$$

results in the system of two first order differential equations

$$\begin{aligned} \frac{du}{dt} &= v, \\ \frac{dv}{dt} &= -\frac{g}{l}\sin u, \end{aligned}$$

for the angular position $\theta = u$ and angular velocity $\dot{\theta} = v$.

For a simple pendulum with $l = 2$ and $g = 9.8$ initially at the rest position $\theta(0) = 0.2$, $\dot{\theta}(0) = 0$, or

$$\boldsymbol{y}(0) = \begin{bmatrix} 0.2 \\ 0 \end{bmatrix},$$

one RK4 step with $h = 0.1$ is as follows

$$\begin{aligned} \boldsymbol{k}_1 &= h\boldsymbol{f}(\boldsymbol{y}_0) = \begin{bmatrix} 0 \\ -0.0973 \end{bmatrix}, \\ \boldsymbol{k}_2 &= h\boldsymbol{f}(\boldsymbol{y}_0 + \tfrac{1}{2}\boldsymbol{k}_1) = \begin{bmatrix} -0.0049 \\ -0.0973 \end{bmatrix}, \\ \boldsymbol{k}_3 &= h\boldsymbol{f}(\boldsymbol{y}_0 + \tfrac{1}{2}\boldsymbol{k}_2) = \begin{bmatrix} -0.0049 \\ -0.0962 \end{bmatrix}, \end{aligned}$$

$$\boldsymbol{k}_4 = hf(\boldsymbol{y}_0 + \boldsymbol{k}_3) = \begin{bmatrix} -0.0096 \\ -0.0950 \end{bmatrix}$$

$$\boldsymbol{y}_1 = \boldsymbol{y}_0 + \frac{1}{6}(\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4) = \begin{bmatrix} 0.1952 \\ -0.0966 \end{bmatrix}.$$

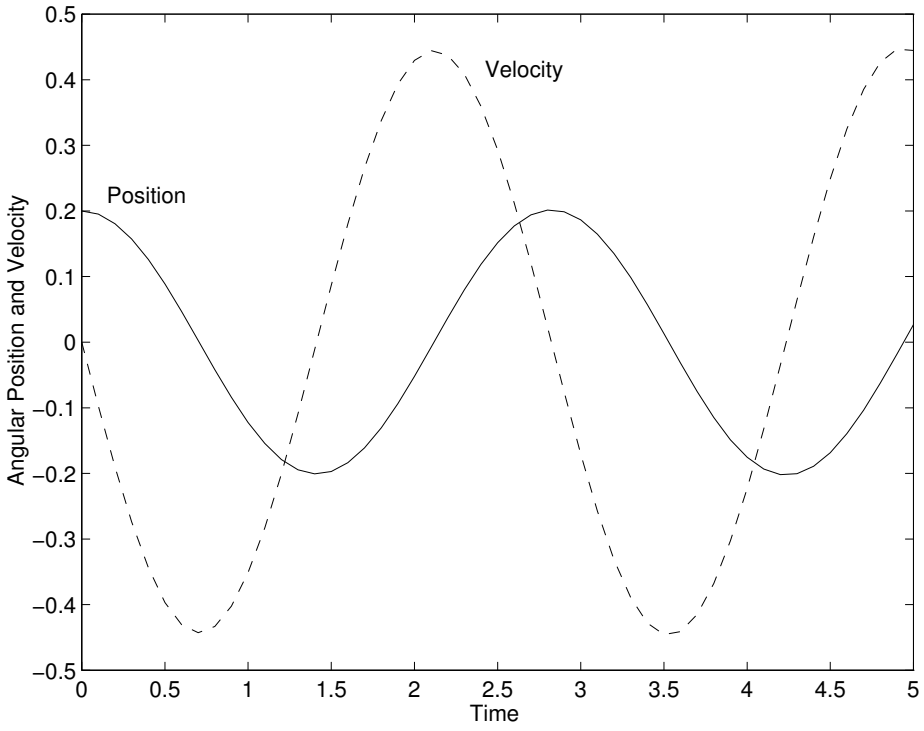


Figure 6.4: Computed solutions of the pendulum equation, giving position θ and angular velocity $\dot{\theta}$ against time.

Continuing the process (on computer) produces the periodic solution curves in Figure 6.4 which gives θ and $\dot{\theta}$ against time.

In MATLAB the functions `ode23` or `ode45` (look them up using `help ode23` for example) could be used to achieve similar results.

`ODE23(ODEFUN,TSPAN,Y0)` with `TSPAN = [T0 TFINAL]` integrates the system of differential equations $\boldsymbol{y}' = f(t, \boldsymbol{y})$ from time `T0` to `TFINAL` with initial conditions `Y0`. Function `ODEFUN(T,Y)`, where `T` is a scalar input and `Y` is a vector of values, must return a column vector corresponding to $f(t, \boldsymbol{y})$. Each row in the solution array `Y` corresponds to a time returned in the column vector `T`. To obtain solutions at specific times `T0,T1,...,TFINAL` (all increasing or all decreasing), use `TSPAN = [T0 T1 ... TFINAL]`.

6.4 Summary and key formulae

- *Euler's method* for solving the ODE

$$\frac{dy}{dt} = f(t, y)$$

Algorithm 27 use ODEFUN in MATLAB for the pendulum problem

First define this function.

```
% Function: pen
% Syntax:  Q = pen(t,Y)
% Inputs:  t = time (a scalar)
%          Y = column vector of values
%              for angle and angular speed
% Outputs: Q = column vector of rates of change
%              of angle and angular speed
function [Q] = pen(t,Y)
L=2;    g=9.8;  % set pendulum length and g value
u=Y(1); v=Y(2); % Equations are u'=v; v'=-(g/L)sin(u)
Q = [v;-g*sin(u)/L];
```

Then to solve the system from $\boldsymbol{y}(0) = (0.2, 0)$ up to $t = 5$ and plot the results against time use the commands

```
y0=[0.2;0];
[T,Y]=ode23('pen',[0,5],y0);
plot(T,Y);
```

takes the form

$$y_{n+1} = y_n + hf(t_n, y_n)$$

and offers simplicity without enough accuracy. The error at each step is of order h^2 and the accumulated error is of order h.

- The *improved Euler method* uses a *predictor* equation

$$y^*_{n+1} = y_n + hf(t_n, y_n),$$

which is then used in the *corrector* equation

$$y_{n+1} = y_n + \frac{1}{2}h\left[f(t_n, y_n) + f(t_{n+1}, y^*_{n+1})\right],$$

to get the next approximate value. This can conveniently be written as

$$\begin{aligned} k_1 &= hf(t_n, y_n), \\ k_2 &= hf(t_{n+1}, y_n + k_1), \\ y_{n+1} &= y_n + \frac{1}{2}(k_1 + k_2). \end{aligned}$$

The error at each step is of order h^3 and the accumulated error is of order h^2.

- The *Runge–Kutta RK4 Method* uses a more sophisticated method with *higher accuracy*:

$$\begin{aligned} k_1 &= hf(t_n, y_n), \\ k_2 &= hf(t_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1), \\ k_3 &= hf(t_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2), \\ k_4 &= hf(t_n + h, y_n + k_3), \\ y_{n+1} &= y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4). \end{aligned}$$

The error at each step is of order h^5 and the accumulated error is of order h^4.

Appendix A Programming with Matlab

Chapter contents

This appendix will lead you through numerical programming using MATLAB. When studying this appendix[1], peruse all of the appendix first so that you have some idea of what can be found in it should you need it in the course of writing programs for the other modules. Then, in conjunction with the numerical modules in the Study Book you should refer to this appendix repeatedly as you attempt the programming exercises throughout the course. This way your programming skills will build up through writing programs. You cannot learn how to write programs through reading alone any more than you can learn how to drive a car from a manual.

[1] Everything in this appendix will be required when implementing the numerical methods discussed in the rest of the course. Use this module as a reference throughout the rest of the course.

A.1 Introduction

MATLAB is an interactive programming environment. It allows the user to interact directly with the MATLAB interpreter through a *command window*. Commands typed in at the command window are executed immediately. Coupled with the rich graphical capabilities of MATLAB, the immediacy of the *command window* makes it a good tool for data visualisation and investigating ideas and concepts.

Using MATLAB for numerical methods requires you to develop skill in writing functions to perform various algorithms. Not only must you have it clear in your mind what has to be done (the algorithm) but you also need to analyse carefully which of the tasks will be useful stand-alone tools. The MATLAB distribution includes toolboxes for Signal Analysis and Symbolic manipulation but there are many more that are available at extra cost. Toolboxes in widely used areas such as structural analysis are developed for sale by various people, while others are developed for use in a particular work situation and have little application beyond that workplace.

Functions are used repeatedly in their area of application, either to supply answers to well defined problems or to build other functions to answer work specific problems. The important thing about a function is that, having identified a well known *task*, what *data* are required to complete the task and a way (or *algorithm*) for completing the task, all this information is stored away and can be recalled at will. If written carefully, functions are reusable across projects and problems.

The skill of writing an effective function cannot be acquired by just reading text like this module. To develop skills in writing numerical programs, requires more than reading, it requires you to write programs and to learn from the experience of writing programs. We recommend that this module be used as a reference throughout the course. Everything in this module will be required when implementing the numerical methods discussed in the course. Your skill as a MATLAB programmer should develop as the course progresses and you are called on to write more complex programs. As you write and use functions you will begin to appreciate the features that make a particular function useful and what features limit a function's usefulness.

When data is to be collected from the console or file, there should be special functions designed to do just that job—they may present on-screen menus, boxes to be filled in, icons to be clicked on and so forth. Designing a user interface is a whole area of study in itself. MATLAB does provide some interface tools which lead to quite a professional looking product but it would take us the whole course to master these and for the most part we will leave this activity to other courses. Ours will be a study of efficient numerical techniques and display of results will be plain and simple: that is why this course is called *High Performance Numerical Computing*.

A.2 Getting started with MATLAB

Starting MATLAB will give a command window similar to the one shown in Figure A.1. Commands are entered at the cursor (>>). Each command is executed when either the `return` or `enter` key is pressed. Any command execution can to be interrupted by pressing `Ctrl` & `C` keys together. To exit MATLAB simply type `exit` or `quit` at the command prompt[2]. Earlier commands executed in a session

interrupt a running command

`quit, exit`

can be recalled using the up (↑)/down (↓) keys on the keyboard. Once recalled the command can either be edited using the keyboard or re-executed by pressing the `return/enter` key. It is also possible to combine multiple commands on the one command line, by separating the commands using either a comma (`,`) or a semi-colon (`;`). The semi-colon (`;`) suppress any output from the command being executed.

suppressing output from a command

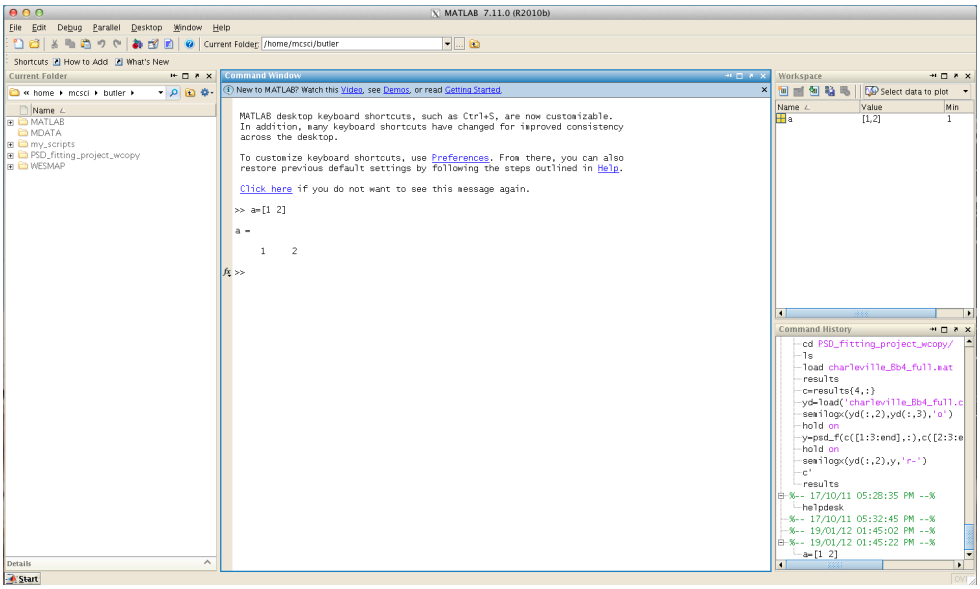


Figure A.1: The default MATLAB Command Window

`helpdesk`

Help about commands and functions can be obtained by typing `helpdesk` at the command plot. This will bring up a window where you can search for help on version topics. Note you can short-cut this window if you know the name of the command or function for which you require help by typing `help command_or_function_name`

`help`

at the command prompt. For example, to get help on the log function you would simply type `help log` at the command prompt. In MATLAB it is also possible to search for commands and/or functions which contain certain keywords using the

`lookfor`

`lookfor` command. As example, the command `lookfor logarithm` will list all the commands in MATLAB relating to logarithms.

Example A.1: Sample commands in MATLAB.

```
>> 3^2-5-6/3*2
ans =
     0
>> 3^2-5-6/(3*2)
ans =
     3
>> 4*2^2+1
ans =
    17
>> sin(pi/2)
```

[2] Note as with all commands in MATLAB this will not be executed until either the `return` or `entry` is pressed

```
ans =
     1
>> sqrt(25)
ans =
     5
>> sqrt(-4)
ans =
        0 + 2.0000i
>> a1=5,b1=3,c1=sqrt(a1+b1)
a1 =
     5
b1 =
     3
c1 =
    2.8284
>> ans
ans =
        0 + 2.0000i
>> t=linspace(0,pi/2,6)
t =
   0   0.3142   0.6283   0.9425   1.2566   1.5708
>> y=cos(t)
y =
   1.0000   0.9511   0.8090   0.5878   0.3090   0.0000
```

Example A.1 illustrates an important feature of the MATLAB computational engine. If an expression is just typed in and evaluated the answer is stored in a special variable called `ans`. This variable contains the value of last unassigned (see §A.3 for details on how variables are assigned) evaluation performed. Hence, the value stored in `ans` changes each time an unassigned calculation is performed in MATLAB as illustrated in Example A.1. Consequently, if you intend to reuse values of calculations you should store the results in a variable (see §A.3) to avoid the possible of using the incorrect result.

ans

Activity A.1 →

To familiarise yourself with some of the basic features of MATLAB and how to use it for elementary arithmetic and as a sophisticated calculator try some of the examples in Example A.1. Make sure you understand what MATLAB did to calculate the answer. Also as you work through the appendix attempt all the examples and exercises as you come to them.

A.3 Variables

Variables are used in a programming language as tags to memory locations in the computer. The data stored in the memory location is accessed using the variable name in the program. The variable is not attached to the data but the memory location. This means that the value contained within a variable can change.

In languages such as C every variable must be declared before it is used, specifying

both its name and its type. The name and type of variables do not need to be pre-declared in MATLAB. In MATLAB any variable can take integer, real, complex, scalar, string, or matrix values. Languages that pre-declare variables have clearly stated at the start of a program or function the type of every variable used. The MATLAB interpreter's liberal approach to variable types has both advantages and disadvantages.

assigning values to variables

The equality sign (=) is used to *assign* the values to a variable. This is an unfortunate choice of symbol and means that when we want to test whether two quantities are equal or not, we cannot use the equal symbol, =, but must use the more cumbersome ==. Thus `5+2 == 3+4` is true but `5+2 = 3+4` is a syntax error. In MATLAB programs, the equals sign does not have its mathematical meaning. It is an assignment operator. The statement to the right of the equals sign is evaluated and the result is placed in the memory location tagged by the variable name on the left. For example, the statement `x = 3 + 5` means MATLAB evaluates the statement on the right and stores the result in the memory location tagged by the variable `x`. This means that the statement `3+ 5 = x`, which is equivalent to `x= 3 + 5` in mathematics, will lead to an error message in MATLAB.

The following are examples of variable types in MATLAB (it is not complete but contains all the variable types that you will encounter in this course).

`x = 3;` Assign the scalar integer value 3 to the variable `x`.

`x = 7.297e17;` Assign the scalar real value 7.297×10^{17} to the variable `x`. The number has been defined using the scientific exponent notation. The number following the `e` is the power of 10 to multiply the number in front of the `e`. Either `e` or `E` can be used.

`x = 'North';` Assign the string *North* to the variable `x`. Strings of characters are defined by surrounding the characters with apostrophes. The construct `'South'` defines a constant string containing the characters `South`.

`x = 0:5:100;` Assign the row vector containing 21 elements from 0 to 100 in steps of 5 to the variable `x`.

`x = [1.2 1.3; 2.5 6.7];` Assign the 2×2 matrix to the variable `x`, so that

$$x = \begin{bmatrix} 1.2 & 1.3 \\ 2.5 & 6.7 \end{bmatrix}$$

`x = 5.1+6.3i;` Assign the complex number $5.1 + 6.3i$ to the variable `x`.

Variable names can be of any length, but must be unique within the first 31 characters. Variable names in MATLAB are *case sensitive*, that is, the variables `time` and `Time` are distinct. However, functions and scripts (which we discuss later) are stored in separate files (with extension `.m`) outside the interpreter and may or may not be case sensitive depending on the outside environment (in a DOS environment, for example, they will not be).

MATLAB allows overloading of names. This means that if you choose a name that has been predefined by MATLAB you loose the built in meaning. For example: `cos = 6.34;` defines a scalar variable `cos`, but it is also the same name for the

Table A.1: Special variables and constants in MATLAB.

Name	Description
`pi`	The value of π to the precession of the machine is returned.
`i, j`	Recognised as the $\sqrt{-1}$.
`Inf`	This is MATLAB's representation of ∞. This is a recognised value in a variable and typically appears as a result of division by zero, or logarithm of zero
`NaN`	Stands for *Not a Number*. This is a recognised value in a variable and appears as a result of a mathematically undefined operation, such as, division of zero by zero, or $\infty - \infty$.
`eps`	Floating point relative accuracy. That is, `eps` returns the smallest difference resolvable between two floating point numbers. For example, the next largest representable number beyond `1.0` is `1.0+eps` . MATLAB cannot represent any floating point number that falls between `1.0` and `1.0+eps` .
`ans`	This variable contains the last computed expression that was not stored in a variable.
`date`	This variable contains a string representation of today's date, in the format dd-mmm-yyyy. For example, `01-Apr-2000`
`realmax`	Is the largest floating point number representable on the computer running MATLAB.
`realmin`	Is the smallest positive floating point number representable on the computer running MATLAB.

Table A.2: MATLAB Arithmetic Operators

Operator	Meaning	MATLAB
^	Exponentiation: x^y	x^y
*	Multiply: $x \times y$	x*y
/	Right Division : $x/y = \frac{x}{y}$	x/y
\	Left Division: $y\backslash x = \frac{x}{y}$	y\x
+	Addition: $x + y$	x+y
-	Subtraction: $x - y$	x-y

MATLAB cosine function. The variable assignment means that the cosine function will no longer be accessible in this session. All references to the name `cos` will assume the variable not the function. Take care not to use variable names that might disable built in functions or constants that you may need later. When writing scripts for others to use, the names of all variables that will be over-written by the running of the script must be listed for easy reference (and one should read these references before running any script). This is one of the reasons why it is nearly always preferable to write *functions* rather than scripts. Variables defined in the course of running a function's algorithm remain *local* to the function and do not over-write workspace variables.

The results of all calculations should be assigned to a variable, so that this value can be used later. If this is not done the result of a calculation will be stored in a special variable called `ans` (see Table A.1). The value stored in this variable is overridden the next time a calculation is done that is not assigned to a variable. This means that the result from the previous calculation is then lost.

A.4 Arithmetic expressions

Table A.2 lists the basic arithmetic operators to be found in MATLAB. With the exception of the left division operator, the arithmetic operators in MATLAB follow standard notation.

A.4.1 Precedence

When more than one operation appears in a mathematical statement, the programming language needs to define in which order the operations must be performed, or the *precedence*. That is, which parts are evaluated first.

If precedence was not defined then the same statement can be evaluated to a different result: for example, is `18/6+3` equal to `2`? or is `18/6+3` equal to `6`? The first statement is evaluated from the right using the precedence rule that usually applies in mathematical statements: that is, work from right to left, as for example in 'sin log tan x' where we start with a value of x, compute the tangent of that value (tan x) then take the logarithm of the resulting number (log(tan x)) and finally take the sine of what this delivers. However, it is usual (because it is much easier to write polynomials with this convention) to give multiplication and division precedence over addition and subtraction. The second statement recognises the precedence of the division over addition and evaluates it first. Right to left precedence has a much

Table A.3: Precedence of arithmetic expressions

Operator	Precedence	Order of evaluation
()	1	Left to right, innermost first
\	2	Left to right
* /	3	Left to right
\	4	Right to left
+ −	5	Left to right

simpler parsing tree than schemes which give precedence to particular operators, but the common arithmetic precedences are so widely accepted and expected that most computer language designers will not risk the possible 'consumer backlash' that breaking with this tradition would involve. So with every computer language one must search out and learn the precedence rules that it adopts. The usual arithmetic precedences will normally be observed, but note that even they are not unambiguous. For example, which divide has precedence in 4/16/32 ? Working right to left, one would get the answer of 8, but MATLAB returns (the decimal for) 1/128 —it works from left to right. Table A.3 lists the operators and their precedence value.

The rules for evaluation of operators are as follows

- Operators having high precedence (low numerical value) will be evaluated before those having a low precedence (high numerical value).
- If an expression contains more than one operator at the same level of precedence, then the left-to-right rule of evaluation apply.

Parentheses `(...)` always have the highest precedence. Whatever is within the parentheses is evaluated first. This means, *when in doubt about the order of any statement place parentheses around the statement* to ensure that the operations are performed in the order you expect.

A.4.2 Matrix arithmetic

One of the strengths of MATLAB is its ability to execute matrix arithmetic[3]. Matrix arithmetic must be implemented in other languages with many lines of code or special functions. MATLAB is especially designed to handle matrix arithmetic. The * represents matrix multiplication or scalar multiplication of a matrix if one of the arguments is a scalar. Since scalars are 1×1 matrices ordinary scalar multiplication can also be performed by the * operator. The dot product of two vectors, however requires that the first vector be a row vector and the second a column vector. To multiply the corresponding elements in two vectors together requires a different operator `.*` so that

```
>  a = [2 3 4] ; b = [6, 7,  8];
   a .* b
ans =
    12.   21.   32.
```

[3] Remember, vectors are examples of matrices. A column vector of length n is an example of a $n \times 1$ matrix.

However, because it is a matrix operation

```
>  a + b
ans =
     8.   10.   40.
```

The standard arithmetic operators described in Table A.2 are also used with matrices. The normal rules of matrix addition, subtraction, and multiplication apply.

The left and right division of matrices are also generalisations of scalar division. Right division `B/A` is roughly the same as `B*inv(A)`. Left division `A\B` is roughly the same as `inv(A)*B`. The difference is in how the division is performed. Different algorithms are used when `A` or `B` are square or not.

MATLAB also defines a transpose operator for matrices. The transpose of the matrix `A` is `A'`. For example, if

$$A = \begin{bmatrix} 2.5 & 6.3 \\ 1.2 & 0.9 \end{bmatrix}, \quad \text{then} \quad A' = \begin{bmatrix} 2.5 & 1.2 \\ 6.3 & 0.9 \end{bmatrix}.$$

A.4.3 Accessing matrix elements

Individual elements of a matrix can accessed and used in statements by specifying the matrix index of an element. The *index* of an element is its position in the matrix. For example, consider the following matrix

$$A = \begin{bmatrix} 6 & -1 & 4 \\ 3 & 5 & 7 \\ -3 & 6 & 1 \end{bmatrix}$$

Define this in MATLAB with the command

```
A=[ 6 -1 4; 3 5 7; -3 6 7];
```

to display the contents of matrix `A` in MATLAB, just type the variable name `A`. To display or use one element from the matrix specify the element. For example `A(1,3)` specifies the element in the first row and third column, that is, `4` for the above matrix.

To change a specific entry just assign a value to it. For example,

```
A(3,1)=-9;
```

changes the value of the third row, first column element from `-3` to `-9`.

To display or use a whole row or column, use the `:` operator. For example:

```
>      A(2,:)
ans =
      3      5      7
>      A(:,2)
ans =
     -1
      5
      6
```

It is also possible to pick out a sub-matrix by the specifying the row and column entries required. For example: To pick out the sub-matrix of A containing the elements $A_{11}, A_{13}, A_{21}, A_{23}$ the command `A(1:2,1:2:3)` could be used as shown below.

```
>      A(1:2,1:2:3)
ans =
      6      4
      3      7
```

Note: `1:2:3` means from 1 to 3 in steps of 2. This technique, along with the assignment operator can be used for values of whole rows/columns or sub-matrices in one command. For example:

```
>      A(1:2,1:2:3)=[10,11;12,13]
A =
     10     -1     11
     12      5     13
     -3      6      7
```

Note that the elements $A_{11}, A_{13}, A_{21}, A_{23}$ now contain the values 10, 11, 12 and 13 respectively, instead of their previous values.

Index notation is logically an unfortunate phenomenon since it clashes with standard function notation. On the face of it `A(3)` should be the value of a function called A with input 3, but when it is interpreted as index notation, A is a *variable*. However, index notation is so familiar and useful to those familiar with classical mathematical notation that many developers of mathematical software are not prepared to risk a "consumer backlash" to providing a functional way of extracting elements from an array, such as for example, `3 pick A` to pick out the third element of A. Similarly, we as teachers feel we would be doing students a disservice if we failed to introduce them to the index notation that abounds in mathematical texts, usually in the form of subscripting, as in A_3. Logic does not always win an argument. In the commercial world of computing, an established user base carries more weight than mathematical logic just as in mathematics tradition may well also fly in the face of logical consistency.

A.4.4 Array arithmetic

MATLAB has two forms of arithmetic operations that can be used with matrices, *matrix arithmetic* or *array arithmetic*. Matrix arithmetic follows the normal rules of matrix algebra. Array arithmetic, which is also called *element-by-element* arithmetic, is where operations are performed on matrices element by element.

Table A.4 lists the MATLAB array operators.

The array addition and subtraction are identical to matrix addition or subtraction. Both array and matrix addition and subtraction operate element by element. Array addition and subtraction require that both arrays have the same size. The only exception to this rule in MATLAB occurs when we add or subtract a *scalar* to or from an array. In this case the scalar is added or subtracted from each element in the array.

Table A.4: MATLAB array operators

Operator	Meaning	MATLAB
.^	Exponentiation	x.^y
.*	Multiply	x.*y
./	Right Division	x./y
.\	Left Division	y.\x
+	Addition	x+y
-	Subtraction	x-y

For example,

```
>      [6 5 6] + 3
ans =
      [9 8 9]

>      [6 4 5 8] + [3 1 2 4]
ans =
      [9 5 7 12]

>      [4 3] - 7
ans =
      [-3 -4]
```

Array multiplication and division is very different to matrix multiplication and division. Array multiplication or division is an element by element multiplication or division that produces an array with the same dimensions as the starting arrays. As with addition and subtraction, array multiplication and division require that both arrays have the same dimensions.

Example A.2: Consider the two row vectors x=[2 4 7 1] and y=[6 1 3 2]. MATLAB does not recognise the possibility of matrix multiplication of these two vectors. They are not compatible: the number of columns of the pre-multiplier must be the same as the number of rows of the post-multiplier. An attempt to execute x * y will bring an error message.

Obtain the usual dot product of the two row vectors by transposing the post-multiplier, as in x*y' or y*x'. The matrix product x'*y produces a matrix of 4 rows and 4 columns.

Array multiplication produces the result:

```
>  x.*y
ans =
     [12 4 21 2]
```

by multiplying corresponding elements of each vector. The result is a row vector with the same dimensions as x and y, that is a 1×4 vector.

Example A.3: Consider the row vectors x=[2 4 7 1] and column vector y=[6 1 3 2]'. Array multiplication of these two matrices is not possible because their dimensions do not agree.

Matrix multiplication `x*y` produces the result `39`, that is the inner product or dot product of the vectors. Note: vectors as such do not exist in MATLAB, there are only single rowed or single columned arrays.

MATLAB not only allows arrays to be raised to powers, but also allows scalars and arrays to be raised by array powers. To perform exponentiation on an element-by-element basis, use the `.^` symbol. For example, if `x=[3 5 8]`, then `x.^3` produces the matrix `[3^3 5^3 8^3] = [27 125 512]`.

A scalar can be raised to an array power. For example, if `p=[2 4 5]`, then typing `3.^p` produces the array `[3^2 3^4 3^5] = [9 81 243]`.

With element-by-element array operations it is important to remember that the dot (.) is a part of a two-character symbol for a particular array operator, such as (`.*`), (`./`), (`.\`), (`.^`). Sometimes this can cause some confusion: the dot in `3.^p` is not a decimal point associated with the number 3. The following operations, with the value of `p=[2 4 5]`, are equivalent:

```
3.^p
3.0.^p
3..^p
(3.).^p
3.^[2 4 5]
```

A.5 Built–in functions

A.5.1 Arithmetic functions

Arithmetic expressions often require computations other than addition, subtraction, multiplication, division, and exponentiation. For example, many expressions require the use of logarithms, exponentials, and trigonometric functions. MATLAB has predefined a large number of functions that perform these types of computations (see Tables A.5 and A.6).

An example of a predefined function is `sin`, which returns the sine of a given angle:

```
m = sin(angle);
```

where the variable `angle` contains the *angle in radians*, and the result is stored in the variable `m`. The variable `angle` is called the *argument or parameter of the function*. If the angle is stored in degrees, it must be converted to radians; for example,

```
m = sin(angle*pi/180);
```

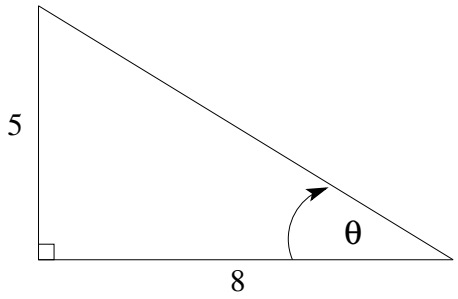
or

```
radians =angle*pi/180;
m = sin(radians);
```

In MATLAB the *arguments or parameters* of the function are contained in *parentheses following the name of the function*. This is referred to as *pre-fix* notation. Many common functions in mathematics have a syntax using *in-fix* notation where two arguments appear to the right and left of the function name, as, for example, in

$x + y$, $a \times b$, $n \bmod r$, etc. MATLAB does not support in-fix notation for any but the arithmetic functions. Classical mathematical notation also has examples of *post-fix* notation— $n!$, x squared —and other hybrids like $|x|$, while MATLAB has `a:h:b` to confound the purist.

A function may *contain no arguments, one argument, or many arguments* depending on its definition. If a function contains *more than one argument, it is vital to give the arguments in the correct order.* For example, consider the problem of finding θ in the following triangle, using the `atan2` command.



Now by hand calculation the answer is 0.5586 radians. If we use `atan2` as below, we get the answers shown.

```
>>     atan2(5,8)
ans =
     0.5586
>>     atan2(8,5)
ans =
     1.0122
```

The first answer is the correct one. The second answer is incorrect, since we passed the values to the function in the incorrect order.

Some functions also require that the arguments be in specific units. For example, the trigonometric functions, assume that the arguments are in radians.

In MATLAB, some functions use the number of arguments to determine the output of the function. For example, the `zeros` function can have one or two arguments, which then determine the output (check this with the command `help zeros`).

A *function reference* cannot be made on the *left hand side of an assignment*, since a function call always returns a value(s) whereas an assignment is always to a variable. Functions can, of course, appear on the right side of an assignment since they return values to be assigned to some variable. Functions may be applied to the output of other functions and may be called in the algorithm defining a function; for example,

```
logx = log(abs(x));
```

When one function is used to compute the argument of another function, be sure to enclose the argument of each function in its own set of parentheses.

MATLAB allows recursive functions, that is, functions that call themselves in their defining algorithm.

Table A.5: Some common mathematical functions defined in MATLAB.

Function	Description
`abs(x)`	Computes the absolute value of `x`
`sqrt(x)`	Square root of `x`. The parameter can be negative.
`round(x)`	Round `x` to the nearest integer.
`fix(x)`	Round `x` to the nearest integer toward zero.
`floor(x)`	Round `x` to the nearest integer toward $-\infty$.
`ceil(x)`	Round `x` to the nearest integer toward $+\infty$.
`sign(x)`	Returns -1 if `x` is less than zero, zero if `x` equals zero, and 1 otherwise.
`rem(x,y)`	Returns the remainder of x/y. For example `rem(25,4)` is 1.
`exp(x)`	Returns e^x
`log(x)`	Returns the natural logarithm of `x`. That is, the logarithm to the base e. Parameter can be negative, but cannot be zero.
`log10(x)`	Returns the common logarithm of `x`. That is, the logarithm to the base 10. Parameter can be negative, but cannot be zero.

Exercise A.4: Use examples to show the differences between the functions `round(x)`, `fix(x)`, `floor(x)` and `ceil(x)`.

Exercise A.5: The functions listed in Tables A.5, A.6 are only a small sample of all of MATLAB's mathematical functions.

Using MATLAB's `help` command, list all of the hyperbolic functions available. Also list any restrictions on the values of the function arguments.

A.5.2 Utility functions

In addition to the arithmetic functions MATLAB has several utility functions (see Table A.7). These functions allow users to track variables (`whos`,`who`), feedback an error message if an error occurs in the code (`error`), clear the variables form the memory (`clear`) etc. Essential these functions allow users to debug and perform bookkeeping operations inside their code.

A.6 Visualising data in MATLAB

`mesh & surf`
`plot`

One of the big advantages in using MATLAB is the ability to visualise data and results of complex calculations (see Figure A.2 as example of the 3d capabilities). While MATLAB is capability of complex 3d graphics (see help pages for `mesh` and `surf`) in this section we will focus on the 2d plotting command (`plot`). Hence, in this section we will focus on:

1. plotting mathematical functions from within MATLAB;
2. changing stylistic elements of the plot;
3. adding titles, labels and legends to the plot;

Table A.6: Some common trigonometric functions defined in MATLAB.

Function	Description
`sin(x)`	Sine of x, where x is in radians.
`cos(x)`	Cosine of x, where x is in radians.
`tan(x)`	Tangent of x, where x is in radians.
`asin(x)`	Arcsine or inverse sine of x, where x must be between -1 and 1. The return angle is in radians between $-\pi/2$ and $\pi/2$.
`acos(x)`	Arccosine or inverse cosine of x, where x must be between -1 and 1. The return angle is in radians between 0 and π.
`atan(x)`	Arctangent or inverse tangent of x. The return angle is in radians between $-\pi/2$ and $\pi/2$.
`atan2(x,y)`	Arctangent or inverse tangent of x/y. The return angle is in radians between $-\pi$ and π depending on the sign of x and y.
`sec(x)`	Secant $(1/(\cos x))$ of x, where x is in radians.
`csc(x)`	Cosecant $(1/(\sin x))$ of x, where x is in radians.
`cot(x)`	Cotangent $(1/(\tan x))$ of x, where x is in radians.

Table A.7: Some of MATLAB's utility functions

Function	Description
`clear`	Clear variables from memory
`help`	Display the first set of comment lines from scripts
`who`	List the variables in memory
`whos`	List the variables in memory with their size
`error`	Terminates the execution of the M-file and outputs the supplied error message.
`echo`	Echo the lines of a script file
`type`	Display the contents of a script file
`what`	List the M-files in a directory
`addpath`	Add a directory to MATLAB's search path
`which`	Print out the path to a given MATLAB function
`pause`	Causes the execution of a script to pause until the user presses a key

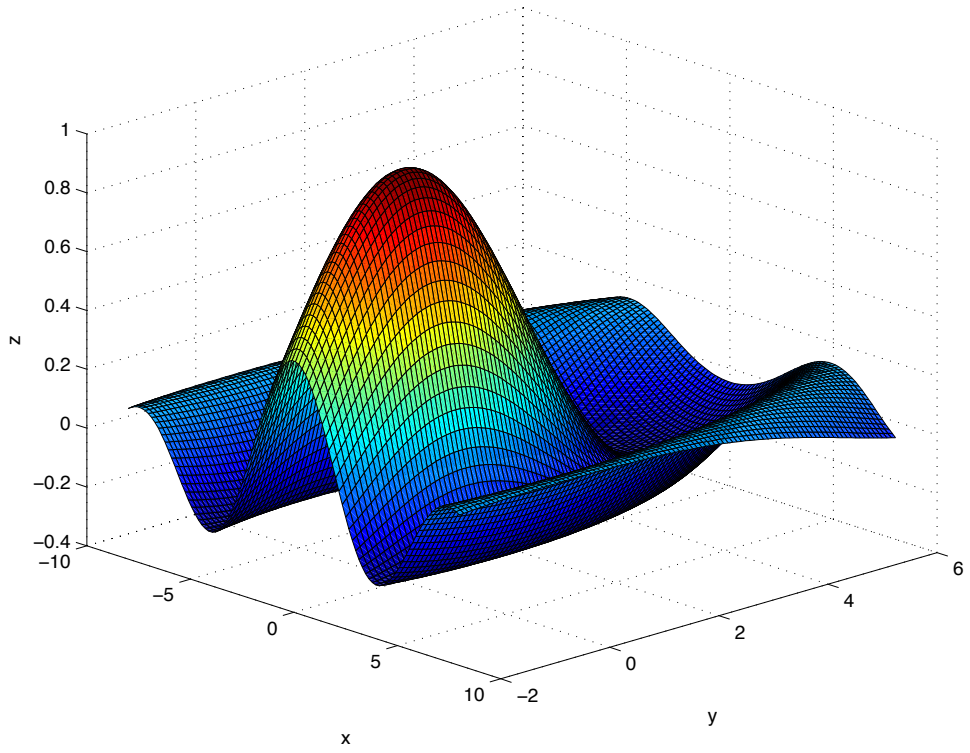


Figure A.2: Plot of $z = \sin(\sqrt{x^2 + y^2})/\sqrt{x^2 + y^2}$ illustrating the 3d capabilities of MATLAB.

4. including mathematical notation in titles, labels and legends; and
5. plotting data stored in an external ASCII file.

A.6.1 Plotting mathematical functions from within MATLAB

To plot a mathematical function in MATLAB three steps are required. These are:

1. Define the domain of the function (e.g. `x=linspace(0,2*pi)`)
2. Calculate the values of function at the selected points in the domain (e.g. `y=sin(x)`)

`plot`

3. Final step is to plot the function using the `plot` command (e.g. `plot(x,y)`).

Using these steps it is possible to produce a basic plot of any mathematical function (see Figure A.3). Note that the plot shown in Figure A.3 does not have a title, axes labels or a legend, all of which are crucial for a good plot. The MATLAB commands to add these elements to the plot are given in the following sections.

Changing stylistic elements of the plot

An optional argument can be given to the plot command to customise each new line aspect. This argument is a concatenated string containing information about colour, line style or markers based on the specifiers shown in Table A.8. To specify

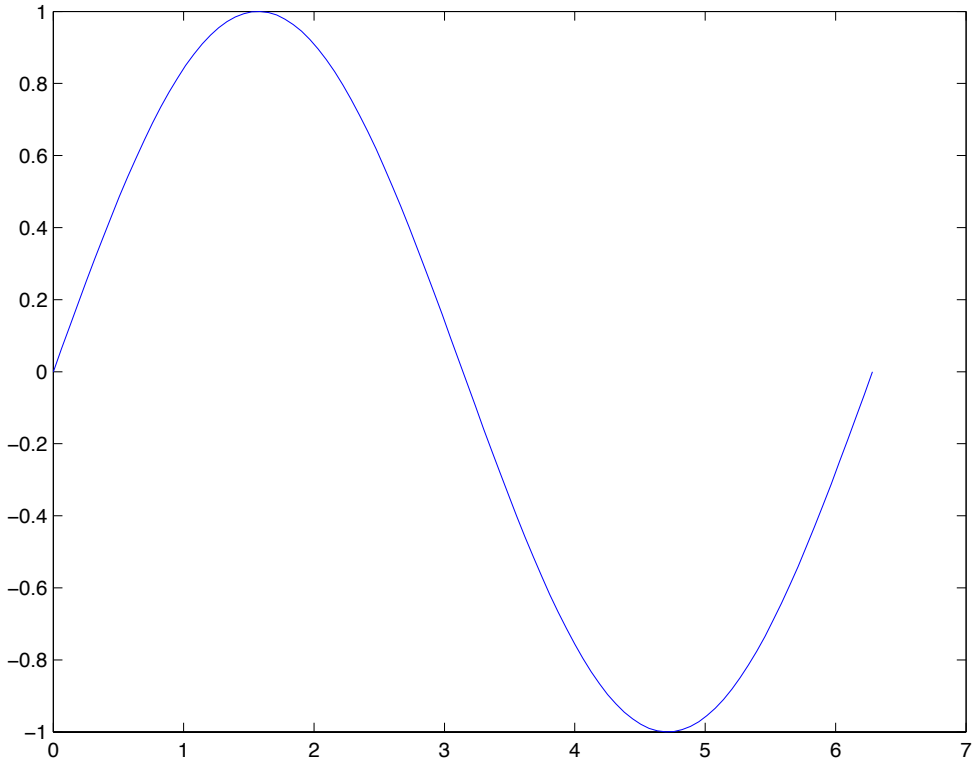


Figure A.3: The default plot of $y = \sin(x)$ in MATLAB. Note that by default axes are not labelled, and there is no title or legend. These need to added using the additional commands `xlabel`, `ylabel`, `title` and `legend`.

the plot be drawn with a red longdash-dot line with diamond marker, the string that would be passed to `plot` command would be 'r-.d'. The string, which is a concatenation (in any order) of the three types of properties shown in Table A.8, must be unambiguous. Furthermore, the string specification is not case sensitive. An example of specifying various line styles is shown in Example A.6 and Figure A.4.

Table A.8: Plot specifiers required to change line styles, colours and markers in plots

Specifier	Line style	Specifier	Colour	Specifier	Marker style
–	Solid line (default)	r	Red	+	Plus sign
– –	Dashed line	g	Green	o	Circle
:	Dotted line	b	Blue	*	Asterisk
–.	Dash-dotted line	c	Cyan	x	Cross
		m	Magenta	s	Square
		y	Yellow	'none'	No marker(default)
		k	Black		
		w	White		

Example A.6: The following commands illustrate how line styles can be used to distinguish between the plots of $\sin(2\pi t)$, $\sin(2\pi t + \pi/6)$, and $\sin(2\pi t + \pi/2)$.

```
t=linspace(0,1,30)
y=sin(2*pi*t)
```

```
y1=sin(2*pi*t+pi/6)
y2=sin(2*pi*t+pi/2)
plot(t,y,'r-o','LineWidth',1.5)
hold on
plot(t,y1,'g-.s','LineWidth',1.5)
plot(t,y2,'c--d','LineWidth',1.5)
legend('y=sin(2\pi{t})','y=sin(2\pi{t}+\pi/6)',...
       'y=sin(2\pi{t}+\pi/2)')
xlabel('t')
ylabel('y')
```

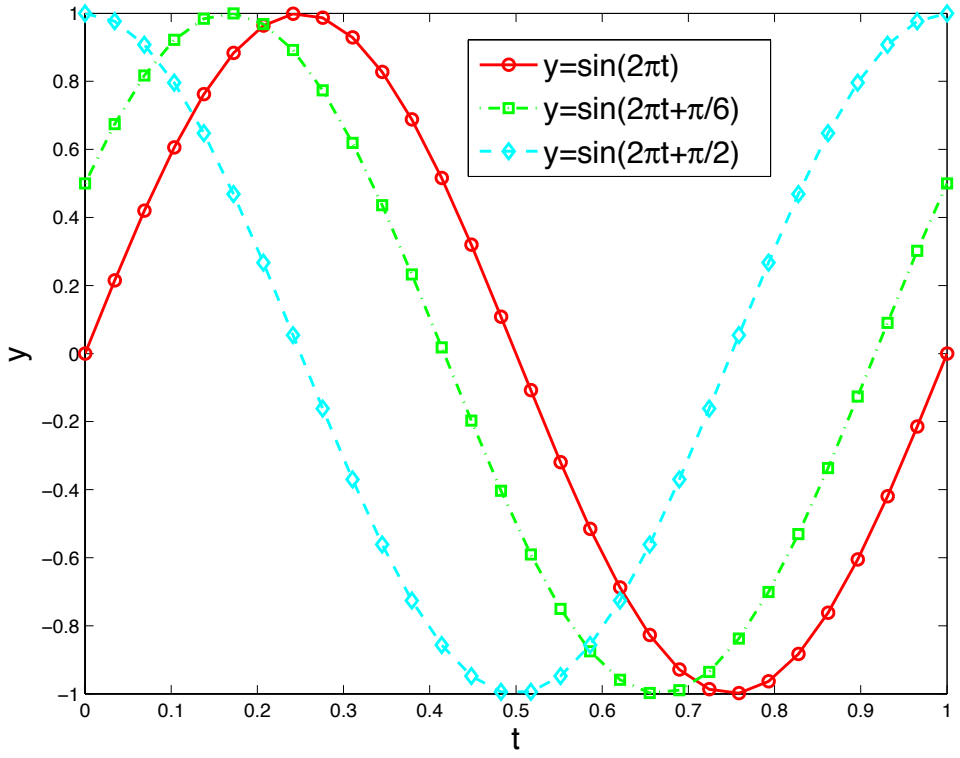


Figure A.4: Resulting plotting after issuing the commands given in Example A.6.

hold

Example A.6 illustrate a number of key commands relating to the creation of multiple plots in MATLAB. The `hold` command holds the current plot and axis properties so that all subsequent plots are added to the existing plot.

To distinguish each plot, a different line specification is passed each plot command. In the example, $y = \sin(2\pi t)$ is plotted using a red solid line with data points indicated using a red circle, as indicated by the string `'r-o'`; $y = \sin(2\pi t + \pi/6)$ is plotted using green dashed-dotted line with data points marked by a square (`'g-.s'`); while $y = \sin(2\pi t + \pi/2)$ is plotted using a cyan dashed line using diamonds to mark the data points (`'c--d'`). Finally, to complete the plot axes labels and a legend has been added to plot, the commands for doing this are discussed in §A.6.2.

A.6.2 Adding titles,labels and legends to the plot

All good plots should have the following properties:

1. The style of figure is appropriate to the information being conveyed;
2. The contents are well organised, so it is easy for readers to interpret the information;
3. Legends, labels etc. are clear and concise (e.g. contain units if required, do not obstruct parts of the plot etc.); and
4. A caption or title that summarises the information presented.

`bar,polar`

Like most mathematical applications MATLAB has a wide variety of different plotting styles that can be used (e.g. look up the help for `bar`, `polar` etc.). This allows you to select the most appropriate style to display and organise your data.

`xlabel, ylabel, title, and legend`

The MATLAB commands `xlabel`, `ylabel`, `title` and `legend` allows you to add axes labels, titles and legends to plot. As shown in Example A.6 the commands `xlabel`, `ylabel`, and `title` take a string input containing the title or label text. For the `legend` command, a string is required for each separate plot. Note the order of the strings in the `legend` command reflects the order in which the each of plots were made.

Including mathematical notation in titles, labels and legends

It is possible to include mathematical characters such as π in MATLAB plots. This is done by inserting special commands into either the title, legend or label string. These commands are based on the abbreviations for mathematical notation used in programs such as LaTeX, MathType etc. A brief summary of the commands available in MATLAB for typesetting mathematical notation is given in Table A.9. To insert these commands into a text string, simply include the commands in the string as shown in Example A.7.

Example A.7: MATLAB commands to produce the labelled plot of $y = \sin\left(2\pi t + \frac{\pi}{4}\right)$ shown in Figure A.5.

```
t=linspace(0,1)
y=sin(2*pi*t+pi/4)
plot(t,y)
title('y=sin(2\pi t+\pi/4)','FontSize',20)
xlabel('x','FontSize',14)
ylabel('y','FontSize',14)
gtext('y=sin(2\pi t+\pi/4)','FontSize',14)
```

`title`

There are a few key concepts to note about the Example A.7. The first can be seen in the string being passed to the `title` command. This string contains the commands for the mathematical symbols required in the title. If superscripts and subscripts are required in a title or label, these are obtained by adding the subscript character (_) and the superscript character (^) to the string. These characters modify the character or substring defined in braces immediately following. For example, `x_{2}` gives x_2, while `e^{2x}` gives e^{2x}.

superscripts & subscripts

`xlabel,ylabel`

Note in Example A.7 the commands `xlabel` and `ylabel` has been used to set the title and axes labels in Figure A.5. In the example we have also customised the font size to make the text more readable, by passing an option to the `title`, `xlabel`,

Table A.9: Basic mathematical symbols available in MATLAB for including mathematical notation in titles, labels and legends.

command	symbol	command	symbol	command	symbol
\alpha	α	\upsilon	υ	\sim	$\sim$
\angle	$\angle$	\phi	ϕ	\leq	$\leq$
\ast	$\ast$	\chi	χ	\infty	∞
\beta	β	\psi	ψ	\clubsuit	$\clubsuit$
\gamma	γ	\omega	ω	\diamondsuit	$\diamondsuit$
\delta	δ	\Gamma	Γ	\heartsuit	$\heartsuit$
\epsilon	ϵ	\Delta	Δ	\spadesuit	$\spadesuit$
\zeta	ζ	\Theta	Θ	\leftrightarrow	$\leftrightarrow$
\eta	η	\Lambda	Λ	\leftarrow	$\leftarrow$
\theta	θ	\Xi	Ξ	\Leftarrow	$\Leftarrow$
\vartheta	ϑ	\Pi	Π	\uparrow	$\uparrow$
\iota	ι	\Sigma	Σ	\rightarrow	$\rightarrow$
\kappa	κ	\Upsilon	Υ	\Rightarrow	$\Rightarrow$
\lambda	λ	\Phi	Φ	\downarrow	$\downarrow$
\mu	μ	\Psi	Ψ	\circ	$\circ$
\nu	ν	\Omega	Ω	\pm	$\pm$
\xi	ξ	\forall	$\forall$	\geq	$\geq$
\pi	π	\exists	$\exists$	\propto	$\propto$
\rho	ρ	\ni	$\ni$	\partial	∂
\sigma	σ	\cong	$\cong$	\bullet	$\bullet$
\varsigma	ς	\approx	$\approx$	\div	$\div$
\tau	τ	\Re	$\Re$	\neq	$\neq$
\equiv	$\equiv$	\oplus	$\oplus$	\aleph	$\aleph$
\Im	$\Im$	\cup	$\cup$	\wp	$\wp$
\otimes	$\otimes$	\subseteq	$\subseteq$	\oslash	$\oslash$
\cap	$\cap$	\in	$\in$	\supseteq	$\supseteq$
\supset	$\supset$	\lceil	$\lceil$	\subset	$\subset$
\int	$\int$	\cdot	$\cdot$	\o	ø
\rfloor	$\rfloor$	\neg	$\neg$	\nabla	∇
\lfloor	$\lfloor$	\times	$\times$	\ldots	$\ldots$
\perp	$\perp$	\surd	$\surd$	\prime	$\prime$
\wedge	$\wedge$	\varpi	ϖ	\O	Ø
\rceil	$\rceil$	\rangle	$\rangle$	\mid	$\mid$
\vee	$\vee$	\copyright	©	\langle	$\langle$

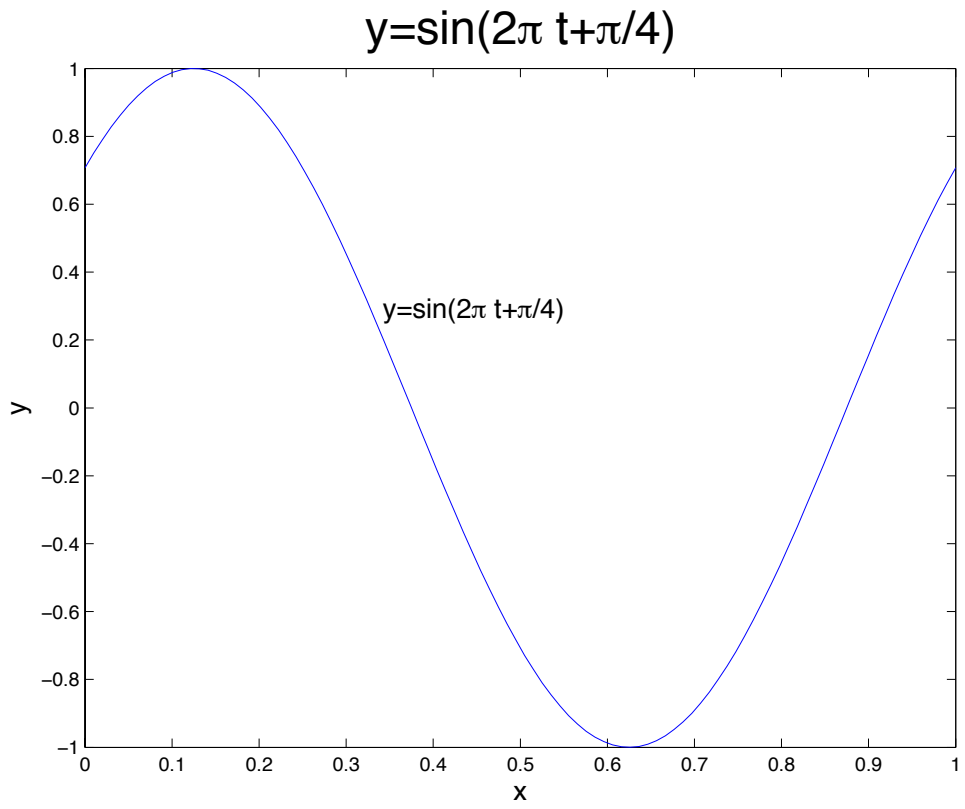


Figure A.5: Plot of $y = \sin\left(2\pi t + \frac{\pi}{4}\right)$ illustrating the use of mathematical notation in labels

gtext

ylabel and gtext commands command. Finally, the gtext command adds the text string to plot at the location specified by clicking on the point required in the plot window.

A.6.3 Plotting numeric data stored in an ASCII file

load, textscan, fscanf, and importdata

As data can be stored in a wide variety of ways in external files MATLAB has a wide variety of functions which can be used to import the data stored in these files (see help for the commands load, textscan, fscanf and importdata). Since data can be stored in such a wide range of formats, we will confine our discussions here to the situation where numeric data is stored in a ASCII (text) file where each row of the file contains the same number of entries.

coffeedata.txt (Windows)
coffeedataunix.txt (Unix/Mac)

To illustrate the procedure for plotting data stored in ASCII file, we will consider the data obtained measuring the temperature of a cup of coffee every minute for a total of 100 mins. The data recorded is stored in the a file called coffeedata.txt[4], which simply contains the two columns of numbers shown in Table A.10. As an example, the first 10 lines of the text file contains the following numbers[5].

```
0.0000 100.0000
1.0000 95.0000
2.0000 90.3125
3.0000 85.9180
4.0000 81.7981
5.0000 77.9357
6.0000 74.3147
7.0000 70.9201
8.0000 67.7376
9.0000 64.7540
```

edit

Note the current version of MATLAB has a builtin text editor which can be used to view such text files. To access this editor simply type the command edit at the command prompt. Once open you can use the graphical interface to open the required file.

load

The first issue is to get the data into MATLAB so it is available for plot command. In MATLAB the command load can used to import the data and assign it to a variable. In the current example, we would use the following command to import the data and assign it to a variable called cdata assuming the data is in the current directory[6]:

```
cdata=load('coffeedata.txt')
```

We now have a variable called cdata which has two columns of data. The first column contains the time, while the second contains the Temperature records. Hence, we can plot it using the following command:

[4] The file coffeedata.txt has been attached and should be available for saving provided you are using Adobe Reader. To save the file right click on the purple Windows version or the red MacOS/Linux version in the margin and select save file from the options. Ensure you save the file as coffeedata.txt.

[5] normally you will have received the data file in this format.

[6] You can check what directory MATLAB is in by typing the command pwd

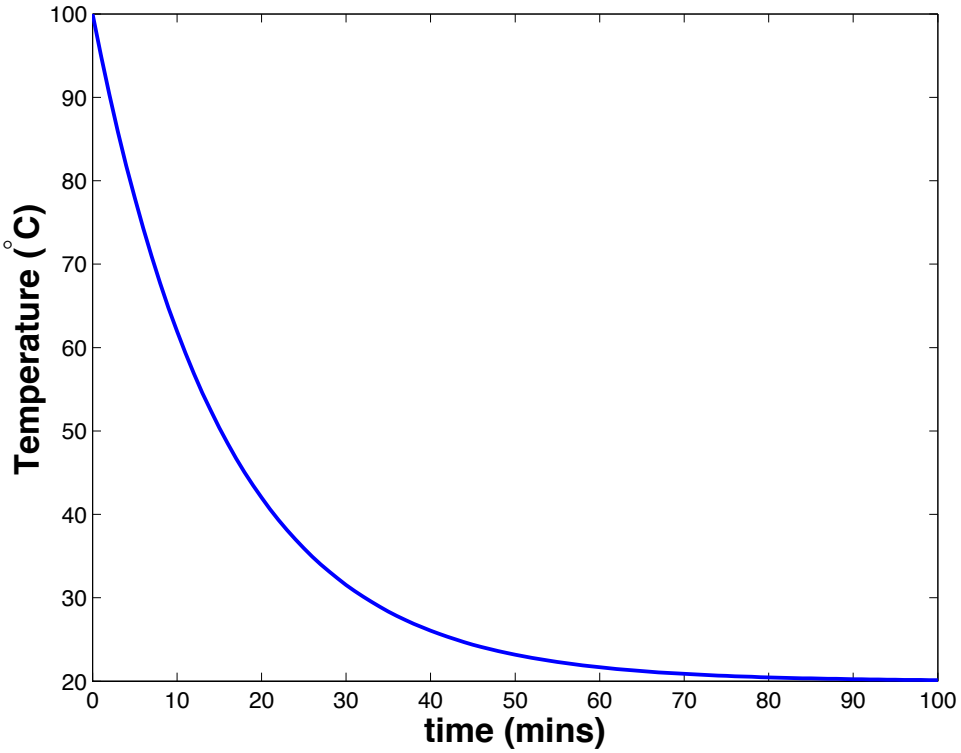


Figure A.6: Temperature of coffee based on the data recorded in the file `coffeedata.txt`.

```
plot(cdata(:,1),cdata(:,2))
```

This gives the plot shown in Figure A.6, after appropriate labels have been added as described above.

A.7 Script files

Executing commands entered at the keyboard or keypad is the most primitive use of a calculating device. It is only when sequences of operations (to perform complex tasks) are stored away and re-called for repeated application with different inputs that we have a *programmed* operation. Complex programs must necessarily have an interface with the user. The basic interface tool in MATLAB, capable of executing sequences of commands that are stored in a file is the **script**. The *file names* must have an *extension* of `.m`, for example `program1.m`, `assignment1.m`. These files are called *Script M-files* or simply *scripts*.

> A script file is a file of standard keyboard characters that define commands to drive a program. A script file can be generated using any text editor or word processor.

To *invoke a script* in MATLAB, *enter the name of the M-file, without the* `.m` extension. When the script runs it is exactly the same as if you typed the commands into the command window from the keyboard. This means that the M-file can use any variables you have defined prior to invoking the script. Any variables created in

Table A.10: Coffee temperature recordings at one minute intervals.

Time (mins.)	Temperature (°C)	Time (mins.)	Temperature (°C)
0.0000	100.0000	51.0000	22.9759
1.0000	95.0000	52.0000	22.7900
2.0000	90.3125	53.0000	22.6156
3.0000	85.9180	54.0000	22.4521
4.0000	81.7981	55.0000	22.2988
5.0000	77.9357	56.0000	22.1552
6.0000	74.3147	57.0000	22.0205
7.0000	70.9201	58.0000	21.8942
8.0000	67.7376	59.0000	21.7758
9.0000	64.7540	60.0000	21.6648
10.0000	61.9568	61.0000	21.5608
11.0000	59.3345	62.0000	21.4632
12.0000	56.8761	63.0000	21.3718
13.0000	54.5714	64.0000	21.2860
14.0000	52.4107	65.0000	21.2057
15.0000	50.3850	66.0000	21.1303
16.0000	48.4859	67.0000	21.0597
17.0000	46.7056	68.0000	20.9934
18.0000	45.0365	69.0000	20.9313
19.0000	43.4717	70.0000	20.8731
20.0000	42.0047	71.0000	20.8186
21.0000	40.6294	72.0000	20.7674
22.0000	39.3401	73.0000	20.7194
23.0000	38.1313	74.0000	20.6745
24.0000	36.9981	75.0000	20.6323
25.0000	35.9357	76.0000	20.5928
26.0000	34.9397	77.0000	20.5557
27.0000	34.0060	78.0000	20.5210
28.0000	33.1306	79.0000	20.4885
29.0000	32.3100	80.0000	20.4579
30.0000	31.5406	81.0000	20.4293
31.0000	30.8193	82.0000	20.4025
32.0000	30.1431	83.0000	20.3773
33.0000	29.5092	84.0000	20.3537
34.0000	28.9148	85.0000	20.3316
35.0000	28.3577	86.0000	20.3109
36.0000	27.8353	87.0000	20.2915
37.0000	27.3456	88.0000	20.2733
38.0000	26.8865	89.0000	20.2562
39.0000	26.4561	90.0000	20.2402
40.0000	26.0526	91.0000	20.2252
41.0000	25.6743	92.0000	20.2111
42.0000	25.3197	93.0000	20.1979
43.0000	24.9872	94.0000	20.1855
44.0000	24.6755	95.0000	20.1739
45.0000	24.3833	96.0000	20.1631
46.0000	24.1093	97.0000	20.1529
47.0000	23.8525	98.0000	20.1433
48.0000	23.6117	99.0000	20.1344
49.0000	23.3860	100.0000	20.1260
50.0000	23.1743		

the script are also available when the script is finished *and* any variable whose name was assigned a value in the script will be over-written.

If you give a script the same name as any variable that it computes in the running of the script, then MATLAB will give precedence to the workspace variable over a script. As stated in §A.3 the variable name will hide the script name after the script has run. MATLAB will take the name as referring to the variable and not to the defined script.

MATLAB also *needs to know where to find the script to executed.* If MATLAB *cannot find the script* it will report that you have *an undefined function or variable.* To check if MATLAB can find the script the command called `which`, can be used. This command will either report what directory the script is in or that it can't find the script. For example:

`which`

```
> which dot
   /usr1/local/matlab/toolbox/matlab/specfun/dot.m
> which dot1
   dot1 not found
```

If MATLAB reports that it cannot find the script, the default MATLAB search path[7] needs to be appended. There are a number of ways to do this in MATLAB. The simplest way to do this is to use the `addpath` command. For example if all the script files for MAT2409 are to be stored in a directory called `MAT2409_scripts` on C drive under windows, than add this directory to the search path as follows:[8].

`addpath`

```
addpath('c:\MAT2409_scripts')
```

The `path` command can now be used to check that this directory is now contained in the MATLAB search path. This command will list all the directories that MATLAB searches.

`path`

The disadvantage of this approach is that you need to run this command each time you start MATLAB and want to access scripts in this directory. To avoid this it is possible to add the directory permanently to the default MATLAB search path. To do this under you have to do the following:

1. From the File menu, select Set Path...
2. Click on the Add to Path button.
3. Select the directory required as per normal.
4. Click OK.
5. Click Save Settings button.
6. Click on the Close button.

The exact procedure for creating and saving M-files depends on the MATLAB version, the operating system (MS Windows, Macintosh, or Unix/Linux), and the text editor you use. The current version of MATLAB includes a built-in editor which can

[7] This is the standard directories that MATLAB searches for scripts
[8] The procedure under UNIX/Macintosh is similar see MATLAB help for details

be used to create and edit M-files and text files. You also need to ensure that your editor saves the file in a directory that is in the MATLAB search path.

`edit` To create a new M-file under the current version of MATLAB simply type `edit newfilename.m`. This will open up the MATLAB editor with a blank document called `newfilename`.

A.7.1 Script input and output

A script is a named file (with the extension `.m`) containing a sequence of MATLAB instructions which, when its name is executed in a session performs those instructions in order as though they were entered at the console.

There are two common uses for scripts. Pragmatically we may want to save a particular computational sequence for later use or reference. If the tasks involved are to be repeated often, then the nature of the tasks should be analysed, formulated into useful *functions* (which we will discuss later) and the whole procedure presented in a single function. But, if the tasks are only likely to get occasional use, then the script may be adequately commented and stored as a script file. It is a quick and dirty way of saving the instructions for doing a particular job, when we do not want to go to the trouble of writing the functions for the job. It is not a highly recommended modus operandi, but useful all the same.

The second use is for the script to act like a *storyboard* presenting information to a user in a structured and controlled way permitting the user to see the steps of a computational procedure and their outputs up on the screen. At a more sophisticated level, the script takes on the proportions of an *application* controlling a large and complex interaction with advanced interface features. We will not take our study of MATLAB programming to this level, but work mostly at the storyboard level using the simplest of MATLAB's interface tools.

All the variables created in a script become global variables defined in the workspace and available for perusal. As an application it is assumed that a script controls the entire workspace and the variables it creates are 'the data' of the application. When scripts are used as storyboards on a small scale then it cannot be assumed that the variables they create deserve global recognition and it is courteous to erase these variables when the script is finished unless they have some possible further use. An unfortunate side-effect is that if variables with the same name already exist in the workspace before a script is run then these variables will be over-written and lost.

Used as storyboards, scripts allow the user to cut, paste and re-try the various steps of a procedure and to modify the script to suit his own needs. Storyboards may be used for demonstrations of toolbox utilities or extended documentation/explanation of functions. However, if there is any danger that variables defined by the script may over-write workspace variables then the demo or documentation should be re-fashioned into a function that leaves the workspace variables untouched (more on this later).

Scripts will often read data from pre-prepared files and we will discuss the functions provided by MATLAB to perform this task below.

Gathering data entered by the user really falls into the realm of interface design but we only briefly consider the simplest interface function provided by MATLAB. Similarly, we briefly consider the functions for providing the simplest forms of output presentation.

Keyboard input

`input`

MATLAB has a built-in function `input` that can be used to prompt the user to enter the value of a variable from the keyboard. For example,

```
Tc = input('Please enter the temperature in degrees Celsius:');
```

will print the message

```
Please enter the temperature in degrees Celsius:
```

and MATLAB will wait for the input. The input is terminated by pressing the `Enter` (or `Return`) key.

The input entered can be any valid MATLAB expression, which will be evaluated, and the answer assigned to the variable (in this case `Tc`) in the input statement. For example:

```
Please enter the temperature in degrees Celsius: [0:25:100]
```

where the `[0:25:100]` has been entered by the user in response to the input prompt. The variable `Tc` will be assigned the row vector [0, 25, 50, 75, 100].

Example A.8:

```
% Script :  Quad_0s
% Variables created/overwritten:
%   A, B, C, D, x1, x2, x, y, a, b, w
% Purpose: To illustrate the use of the
%          quadratic formula for solving
%          a quadratic equation of the form
%                Ax^2 + Bx + C = 0
%          with the user to enter A,B,C

%          Source: Leigh Brookshaw, Mar 15, 2000;
%          Last Modified: W Spunde, May 2002
% ------------------
% Ensure all subsequent steps are displayed with:
clc, echo on
%    **********************************
%         Calculating the solutions
%    of the equation  Ax^2 + Bx + C = 0
%    **********************************
%         [Tap any key to start]
    pause, echo off

 A = input('Enter value of A in Ax^2+Bx+C: ');
 B = input('Enter value of B in Ax^2+Bx+C: ');
```

```
 C = input('Enter value of C in Ax^2+Bx+C: ');
 disp(' ')
 disp('First we compute the discriminant')
 disp(' ')
 disp('     [Tap any key to continue]')
    pause, echo on

 D = B^2 - 4*A.*C

%       [Tap any key to continue]
    pause
% The quadratic formula gives:
 x1 = (-B + sqrt(D))./(2*A);
 x2 = (-B - sqrt(D))./(2*A);
 sort([x1,x2])
%       [Tap any key to continue]
    pause
%  Check this from a graph of the function.
    echo off
if D<0
   a=-B/(2*A); b=-B/(2*A); w=4*abs(a); x1=a; x2=a;
else
   a=min([x1,x2]); b=max([x1,x2]); w=b-a;
end
x=linspace((a-w/4),(b+w/4),100);
y=A*x.^2+B*x+C;
plot(x,y,x1,0,'*',x2,0,'*'), grid
clear x y a b w
```

There are several things to note about this script.

(a) Echoing the instructions on the screen is turned off when the `input` function is to be used and comments are then put onto the screen with the `disp` function. This method of displaying comments could be kept throughout the script.

(b) The `pause` function may be used to punctuate the presentation so that it does not all just scroll past the viewer without his having the opportunity to read how the computation progresses.

(c) The graphing of the function involves a few considerations that we do not wish to trouble you with, since we want you to concentrate of the quadratic formula. The variables used in this part of the script are erased with `clear`—but note that if any variable in the workspace is called `a`, `b`, `x`, `y` or `w` then the result will be that it is erased.

(d) If you write a similar script for solving say two equations in two unknowns then you will see that most of the time will be spent on perfecting the layout of the presentation rather than on the solution process.

Formatted output

The semicolon at the end of a line in MATLAB suppresses the display of output from a computation, but allowing the output to be displayed in MATLAB's default mode is sometimes not very convenient.

`fprintf` MATLAB provides the function `fprintf` (a direct implementation of the C function of the same name) for formatted output. The form the command takes is:

```
fprintf('format',A,B,...);
```

`format` The string `'format'` describes how to display the variables `A,B,...`. The format string contains text to be displayed plus special commands that describe how to display each number. The special commands takes the general form

```
%[-][number1.number2]C
```

where anything in square brackets is optional.

The % sign tells MATLAB to interpret the following characters as codes for formatting a variable. The `number1` specifies the minimum field width.[9] If the number to be printed will not fit within this width, then the field will be expanded. The `number2` specifies the number of digits to print, to the right of the decimal point. That is, the precision of the output. If the precision of the variable is larger than the precision of the output field, then the last digit is rounded before printing. Normally the numbers printed are right justified, the minus sign indicates that they should be left justified.

The final character in the format specifier 'C' is the format code. This is not optional, it must appear to specify what form the number is to be printed. Table A.11 lists the most useful format codes.

Example A.9: Some examples of using the `fprintf` command.

```
fprintf('The speed is (kph): %3.1f',Speed);
```

if `Speed` = 62.34, this produces the output

```
The speed is (kph): 62.3
```

If `A` = 3.141, and `B` = 6.282, then the command

```
fprintf('Pi=%5.3f, TwoPi=%5.3f',A,B);
```

produces the output

```
Pi=3.141, TwoPi=6.282
```

If `M=[1.2 1.5; 0.6 0.3]`, then the command:

```
fprintf('%3.1f ',M);
```

produces the output

```
1.2 0.6 1.5 0.3
```

[9] this is the minimum number of characters the number will occupy, including the decimal point and ± signs. For example the field width of −2.345 is 6.

Table A.11: The `fprintf` format string number types

Format code	Description
`%f`	Decimal format.
`%e`	Exponential format with lowercase `e`
`%E`	Exponential format with uppercase `E`
`%g`	`%e` or `%f`, whichever is shorter
`%G`	`%E` or `%f`, whichever is shorter

Table A.12: The `fprintf` format string escape sequences.

Escape Sequence	Description
`\n`	start a new line in the output
`\b`	Backspace
`\t`	Tab
`\\`	Print a backslash
`%%`	Print a percent
`''`	Print an apostrophe

The `fprintf` command prints by column when a matrix is to be printed.

Example A.10: An example of the rounding that can occur when using `fprintf` . The output from the command

```
fprintf('Pi=%7.5f, Pi=%6.4f',pi,pi);
```

is

```
Pi=3.14159, Pi=3.1416
```

The format string can also contain ordinary text which will be output unchanged. Escape sequences can be embedded in the format string that have special meaning. Table A.12 lists the most useful escape sequences.

File input and output

Variables can be defined from information that has been stored in a data file. MATLAB recognizes two different type of data files: MAT-files and ASCII-files. A MAT-file contains data stored in a space-efficient binary format. Only a MATLAB program can read MAT-files. If data is to be shared with other programs, then use the ASCII file type. An ASCII (American Standard Code for Information Interchange) file contains only the standard keyboard characters, and is readable and editable on any platform using any text editor.

To save variables to a MAT-file, use the command `save`. For example:

```
save data1 x y;
```

will save the matrices `x` and `y` in the file `data1.mat`. To restore these matrices in a MATLAB program, use the command[10]:

```
load data1;
```

[10] Remember this file must be in the MATLAB search path

This command will define the variables x and y and load them with the values stored in the MAT-file. The advantage of saving data in a MAT-file is that when it is loaded the variables are restored. The disadvantage is that the file is a binary file and is only readable by MATLAB.

An ASCII data file that is going to be loaded into a MATLAB program must only contain numeric information, and each row of the file must contain the same number of data values. The file can be *generated using a spread sheet program, a text editor, or by any other program.* It can also be generated by a MATLAB program using the `save` command. To save the variables x and y to an ASCII file called `data1.dat`, the command is:

```
save data1.dat x y -ascii;
```

Each *row* of the matrices x and y will be written to a separate line in the data file. The `.mat` extension is not added to an ASCII file. However, it is good practice, as illustrated here, to add the extension `.dat` to all ASCII data files so that they can be distinguished from other MATLAB file types, and are easily recognisable as data files.

Exercise A.11: The `save` command has a number of modifiers, other than `-ascii`. Use the MATLAB `help` command and explain the meaning of the modifiers `-double`, and `-append`

The `load` command is used to read ASCII files as well as MAT-files. If the filename used with the `load` command does not contain a type extension then a MAT-file is assumed. If the filename contains any type extension, other than `.mat` then an ASCII file is assumed. For example:

```
load data1;
load data2.mat;
```

In both cases MATLAB will look for a MAT-file, `data1.mat` and `data2.mat`. To load an ASCII file then the file type extension is required, for example:

```
load data3.dat;
load data4.tab;
```

In both cases MATLAB will look for ASCII files. The data in files are loaded into matrices with the same name as the file (without the extension). So, after the commands above, the data in file `data3.dat` will be loaded into the variable `data3`, and the data in file `data4.tab` will be loaded into variable `data4`.

Example A.12: If the file `data3.dat` contains the following data:

```
0.00   0.00
0.01   0.01255
0.02   0.2507
0.03   0.37815
```

The command:

```
load data3.dat;
```

Table A.13: Some of MATLAB's data input/output functions.

Function	Description
`input`	Request input from the keyboard. Any valid MATLAB expression is allowed in reply.
`disp`	Display variables in free format
`fprintf`	Display/print variables in a specified format
`save`	Save variables to a file
`load`	Load data from a file
`dlmread`	Load data from an ASCII file. Data is delimitered by the specified variable.
`dlmwrite`	Write ASCII data to a file. Delimiter the data by the specified variable.
`csvread`	Load data from a comma separated value ASCII file.
`csvwrite`	Write data as a comma separated value ASCII file.

will create a variable `data3` with 2 columns and 4 rows, that will contain the data from the file. Then typing `data3` at the prompt will give

```
>     load data3.dat
>     data3

data3 =
         0         0
    0.0100    0.0126
    0.0200    0.2507
    0.0300    0.3781
```

Then the columns of data in the file can be stored in the variables x and y by simply doing the following: `x=data3(:,1)` and `y=data3(:,2)`.

Exercise A.13: Create the ASCII data file `xyz.dat` containing the following data

0.00	0.01	0.02	0.03
0.12	0.32	0.45	0.87
1.34	1.21	1.78	1.47

Use the MATLAB editor or a text editor of your choice.

Load the data file into MATLAB so that the three variables x, y, and z each contain a row of data from the file.

Note that other more advanced commands exist in MATLAB and can be used to read/write ASCII files (see Table A.13). However these are only used when a specific need arises and are therefore not covered in detail here (see MATLAB Help for more details).

A.7.2 Formatting script files

An important aspect of writing script files is to ensure that the script is readable and easily understood. This ensures that anyone reading the script file (this includes the author of the file) can understand the purpose of the script.

To help make the script understandable it should be commented throughout describing all aspects of the script. In MATLAB all text on a line following the % character is treated as a comment. The first comment line before any executable statement in a script is the line that is searched by the `lookfor` command. This line should contain the name of the function and keywords that help locate the script. The `help` command displays all the comment lines, up to the first blank line or executable line, that occur at the start of a script. A description of the script should be placed here

`lookfor`

Lines in a script file should not be too long. The *lines* should be visible in the editor's window/(printout) without wrapping or horizontal scrolling. This applies to both comment lines and statement lines.

Statements can be broken across multiple lines using the continuation symbol '. . .', three dots in a row. If three dots in a row appear at the end of a line it means that the statement is continued on the next line.

`Continuation symbol(...)`

Normally, while an script file is executing, the commands are not displayed in the command window. The command `echo on` ensures that all lines of the script are displayed as it executes. Echoing is turned off when the interpreter is required to do some computations that need not been seen by the reader.

`echo on`

Example A.14 is an example of a script file that explains how to make various temperature conversions.

Example A.14:

```
% Script :  Temperatures
% Variables created/overwritten:
%   Tc, Tf, Tk, Tr T_all
% Purpose: To illustrate the conversion of
%          Celsius temperatures (Tc) to
%          Fahrenheit (Tf), Kelvin (Tk),
%          and Rankine (Tr) and to produce
%          a table of equivalents to Tc values
%          running from 0 to 100 in steps of 10
%
%          Source: Leigh Brookshaw, Mar 15, 2000;
%          Last Modified: W Spunde, May 2002
% ------------------
echo on
% We create the list of Celsius values with
Tc = [0:10:100];  pause

% Calculate the Fahrenheit equivalents with
Tf = 9*Tc/5 + 32; pause
```

```
% Calculate the Kelvin equivalents with
Tk = 273.16 + Tc; pause

% Calculate the Rankine equivalents with
Tr = 9*Tk/5;  pause

% Create a matrix with equivalent temperatures
% in corresponding rows:
T_all = [Tc; Tf; Tk; Tr];  pause

% and print out the result:
echo off
% Turn echoing off as the rest is instructional
fprintf('Temperature Conversion Table\n');
fprintf('  Tc     Tf     Tk     Tr\n');
fprintf('%6.2f %6.2f %6.2f %6.2f\n',T_all);
% The print instruction works because fprintf
% prints a column at a time.
%  \n  starts a new line in the output

% As a courtesy clear out the
% introduced/over-written variables.
clear Tc Tf Tk Tr T_all
```

Exercise A.15: Code the Example A.14 and name the script file `temperatures.m`. Test that your script file is giving the correct answers. Test the answers by calculating a few of the values in the output table using the MATLAB command window.

Run your script with `echo` on when testing, then comment out the lines when you are satisfied the script works.

Exercise A.16: Test that the command `help temperatures` returns the comments at the start of your script. Test also that the `lookfor` command works.

If the commands fail to find your script, you may need to add the directory the script is in to the MATLAB search path. To do this use the command `addpath`.

Exercise A.17: The growth of bacteria in a colony can be modeled with the following exponential equation

$$y_{\text{new}} = y_{\text{old}} e^{1.386\,t}$$

where y_{old} is the initial number of bacteria in the colony, y_{new} is the new number of bacteria in the colony, and t is the elapsed time in hours.

Write a script that uses this equation to predict the number of bacteria in the colony after 6 hours if the initial population is 1. Print a table that shows the number of bacteria every half hour up to 6 hours. Also plot the answer.

Exercise A.18: Modify the script of Exercise A.17 so that the user enters the elapsed time in hours. Use the `input` command to ask the user for the input.

Exercise A.19: Modify the script of Exercise A.18 so that the user enters the initial population of the colony.

Exercise A.20: Carbon dating is a method for estimating the age of organic substances, such as bones, wooden artifacts, shells, and seeds. The technique compares the amount of carbon 14, a radioactive isotope of carbon, contained in the object with the amount it would have had when it was alive. While living the object would have the same amount of carbon 14 as its surrounding environment. For the last tens of thousands of years the ratio of carbon 14 to ordinary carbon has remained relatively constant in the atmosphere, this allows an estimate to be made of age based on the amount of surviving carbon 14 in an object. The equation is

$$\text{age(years)} = \frac{-\log_e(\text{carbon 14 proportion remaining})}{0.0001216}$$

If the proportion of carbon 14 remaining is 0.5, then the age is 5700 years. Which as you would expect is approximately the half life of Carbon 14.

Write a script that allows the user to enter the proportion of carbon 14 remaining. Print the age in years.

What happens if the user enters a proportion ≤ 0? What happens if the user enters a proportion ≥ 1?

Exercise A.21: Modify the script of Exercise A.20 so that the age is *rounded* to the nearest year.

A.8 Function files

While a script is intended to be either the controlling unit for an entire application or a storyboard for a minor 'application', the computational work, the various specialised tasks that make up an application are done through the use of *functions*. We have seen numerous examples of functions already—the built-in or *primitive* functions of MATLAB such as `input`, `floor`, and `ceil`, the arithmetic functions, `+`, `-`, `*`, `/`, `^` and elementary functions of mathematics such as `sin`, `cos`, `tan`, `exp`, `log`. But in most applications, there are special tasks that recur and for which it would be useful to have a function. MATLAB permits the user to define his own functions and to store them in separate M-files called *function files*.

Functions must have a particular syntax. Only pre-fix notation is allowed and the first executable line of a function M-file must be preceded by a line of the form

```
function [u,v,w] = Fname(x,y,z,t)
```

where `Fname` is the unique name of the function which requires one or more (or no) inputs `x`, `y`, `z`, `t` and returns one of more (or no) outputs `u`, `v`, `w`. The following lines of the function file list in order the steps to be executed on whatever particular inputs are given in the function call (they need not be called `x`, `y`, `...`) and the outputs can be stored under any variable names whatsoever.

Unlike a script file, all the variables used in a function are *local* to that function. This means that the values of these variables are only used within the function and do *not* over-write variables in the workspace. For example if you assign the variable `pi` a value of 3 in the algorithm for some function (the sequence of steps executed when the function is called) then you need not be concerned that it might over-write the standard value of `pi` in the workspace. The variables used to identify the inputs and outputs of a function are also local to the function.

If a function is called without assigning its outputs to variables then the first of the outputs, only, is displayed on the screen.

Functions are the building blocks of larger programs. They feed their results back to other functions and ultimately to the program that interfaces with the user, that is, to a script file. The interface tools (or functions) used by the script file may be more or less sophisticated, but the hard computational work is done in the component functions.

The fact that variables defined within a function are local to that function, means that functions can be used in any program without fear that the variables defined within the function will clash with variables defined in the main part of the program. This also means the programmer need not know the contents of a function, only the information it needs and the information it will return. This is exactly the case with built-in (primitive) functions—the algorithm used is not open to the user and in fact may be proprietary information (information that might be thought to give this software a commercial edge over competitors.) Many of the functions in a sophisticated package like *Mathematica* fall into this category.

Some important points

- The first executable line in a function file must be preceded by a *syntax line*, that lists all the input variables and all the output variables. This line distinguishes a function M-file from a script M-file. The form of the syntax line is:

  ```
  function [output_vars] = function_name(input_vars);
  ```

- The output variables are enclosed in square brackets, as shown.
- The input variables are enclosed with parenthesis, as shown.
- A function does not have to have either input variables or output variables, but most functions will both have inputs and deliver outputs.
- The first comment line of a function is searched for an input string by the `lookfor` function; all comment lines before the first executable line of the file are re-produced by the `help` command. Any user-defined (non-primitive) functions called by the algorithm should be listed in the help comments.
- The function name should be the same as the M-file filename. If the function name is `Temp` then the M-file name should be `Temp.m`.
- MATLAB is *case sensitive* by default, but your operating system may not be. For example, while MATLAB will recognize that `Temp` and `temp` are two dif-

ferent variables, your operating system might not recognize that `Temp.m` and `temp.m` are two different files.

- Functions are executed by calling the function with the correct syntax, in exactly the same way as built-in functions are used (that is functions such as `sin`, `ceil`).
- if the function M-file is not in the same directory as the calling program, then the `addpath` command must be used to add the function directory to MATLAB's search path.
- values needed by the function should be passed as input variables to the function. The use of the `input` command in functions should be avoided unless it is a user interface function.
- the last statement of the function is the last statement in the M-file. When the last statement is executed the function returns and execution precedes from where the function was called.
- the `return` statement can be used to force an early return from a function.
- It is a good practice to place an `end`[11] on the last line of a function. This makes it clear to other users exactly where the function ends.

Example A.22: The following function returns the distance traveled and the velocity of an object that is dropped. The input variable is the elapsed time of falling.

```
% Function:  fall
% Syntax:    [dist, vel] = fall(t,v0)
% Input:      t  =  list of elapsed times in seconds since
%                   body began a fall under gravity from
%                   rest with no resistance
%            v0  =  initial speed at the start of the fall
% Outputs:  dist =  list of distances fallen in metres
%                   from beginning of fall to times t
%             vel =  speed in metres/sec at time t
% Algorithm:  Assumes vertical fall under constant gravity
%             with no air or other resistance forces
% Source:     Leigh Brookshaw 22/3/2000
%             Last Modified: Walter Spunde, May, 2002

function [dist, vel] = fall(t,v0)
% The gravitational acceleration is assumed to be
g = 9.8;  %  metres/second/second
% Under constant acceleration (g) speed t is given by
vel = g*t + v0;
% and distances travelled by
dist = 0.5*g*t.^2 + v0*t;
end
```

[11] This is optional in MATLAB but is a good practice.

The following gives examples of using this function:

```
>    [drop, velocity] = fall(5,0)
drop =
    122.5000
velocity =
    49
```

The output variables `drop` and `velocity` are scalars that have been assigned the return values from the function.

```
>     sec = 5;
>     [drop, velocity] = fall(sec,0)
drop =
    122.5000
velocity =
    49
```

The input can be variables or explicit constants.

```
>     secs=[1:5];
>     [drop, velocity] = fall(secs,0)
drop =
    4.9000   19.6000   44.1000   78.4000  122.5000
velocity =
    9.8000   19.6000   29.4000   39.2000   49.0000
```

Input variable `t` can be a list. The output variables are then vectors of corresponding values for distance and speed (and consequently of the same length.

```
>     fall(5,0)
ans =
  122.5000
```

Here the result is not assigned to a variable. Note that in this case the only answer returned is the value of the first variable in the function definition, namely `dist`. This means to get all output from a function, you must assign all output variables.

Exercise A.23: Use the above function to plot (on the one graph) the distances travelled over a given number of seconds for bodies starting their falls with different speeds by applying the command `plot(drop,secs)` and `hold on` after each execution of the function.

Example A.24: The following code rewrites the script of Example A.8 so that the roots of a quadratic are calculated within a function call

```
%  Function :  quadroot
%  Syntax   :  [x] = quadroot(A,B,C)
%  Input    :   A  =  coefficient of  x^2
%               B  =  coefficient of  x
%               C  =  constant coefficient
%               in a quadratic polynomial
```

```
%  Outputs  :   x  =  a list of the solutions
%              of the equation Ax^2 + Bx + C = 0
%  Algorithm:  Uses the quadratic formula
%  Source:     Leigh Brookshaw 15/3/2000
%              Last Modified: Walter Spunde, May, 2002

function [x] = quadroot(A,B,C)
% Calculate the square root of the discriminant
disc = sqrt(B^2- 4*A*C);
% Calculate and display the two roots
x  = (-B + disc*[1,-1])/(2*A);
end
```

The result of running the function might be stored in a variable name and passed to other functions for further use.

Exercise A.25: Use the function of Example A.24 to assist you in deciding on the plotting window for various quadratic polynomials.

Since functions represent often repeated tasks that may be useful in maintaining an application they are chosen and written with some care and are well documented. In MATLAB the first block of comments in a function can be accessed via the `help` command . The first comment line is returned by the `lookfor` command. In our layout convention this simply returns an identification of the M-file as either a script or a function. `lookfor xyz` looks for the string `xyz` in the first comment line of the help text in all M-files found on MATLABPATH. For all files in which a match occurs, `lookfor` displays the matching lines. Thus this function is not intended to be used as a documentation tool: `help` is the appropriate function for that. For more sophisticated documentation, we would add *keywords* or phrases that might be used by a user searching for a function to do a particular task. Once the user has found the appropriate function name, the user can then use `help` to get more detail on the function's use.

1. The first comment line should contain the function's name and other keyword descriptors that might help someone locate this function when they have a particular task to do but don't know the name of the function that does it;
2. the next line should be the syntax line showing how this function is to be called;
3. then the input variables should be listed, with units if applicable;
4. followed by the output variables, with units if applicable;
5. input and output listings should be followed (or preceded) by a description of what the function does if this is not made clear by the description to the input and output variables; this statement should also include a description of the algorithm that is used, if this has any special or unusual features—for example, 'iterative solution by Jacobi's method' or 'iterative solution by Gauss–Seidel method';

6. divide the function code into recognisable blocks such as 'Data Checking', 'Special Cases' etc by the use of comments if appropriate and also add comments to highlight where any special features are implemented. Do not add inane comments like

   ```
   v= v+2     % adds 2 to the value of v
   ```

   but do not assume that even you will be able to follow your own brilliant logic when you have long forgotten about how you wrote the function, let alone someone else trying to read the code for the first time.

A.8.1 Local variables

The name of the input variables given in the function definition line are local to that function. This means that when the function is called the input variables used in the call statement can have different names. All variables used in the function are cleared from memory after the function finishes executing. The function in Example A.22, can be called using the statement

```
[drop, velocity] = fall(sec);
```

The variable `sec` is the elapsed time in seconds, but inside the function this is called `t`. If, inside the function the variable `t` was changed, it would not affect the variable `sec` in the calling program. This is what is meant by the function's variables being *local* to the function. This *feature* means that functions can be written without concern that the calling program uses the same variable names and therefore function files are *portable*, and need not be rewritten every time they are used in a different program.

A.9 Control structures

In the execution of the function control passes from one line to the next. However there are a number of situations in which it is natural to want control to pass to different lines depending on the value of computed variables. For example, input data may maybe be treated differently depending on when it is received (late payment, assignments, etc).

Decisions to execute different lines of code in different circumstances are called **branching** decisions and most computer languages will provide a branching control structure for implementing such decisions. MATLAB provides the `if...then` and the `switch` branching control structures. The tests that are made to determine the blocks of code that should be executed often involve more than one consideration and MATLAB provides both relational and logical operators to simplify the expression of these conditions.

`if & switch`

Statements that alter the linear flow of the script are called *control structures*

A control structure is also required when blocks of code are to be repeated until a certain condition is met, for example, the computation of interest on a loan balance until such time as the loan balance is paid off or repeated approximation to the solution of a problem until sufficient accuracy is achieved. The repeated execution of a block of code may be achieved by an **iterative** control structure or **loop**. MATLAB

Table A.14: MATLAB's relational operators

Relational operator	Interpretation
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
==	equal
~=	not equal

`while loop`
`for loop`

provides a `while loop` loop which continues repeating the block until a certain condition is met, and a `for loop` which repeats the block a given number of times.

The flow of control may also be disrupted by a *recursive* function definition—that is, a function definition which includes a call to the function itself (with different inputs of course). In this case, the current function execution is suspended until a result is returned from the within-function function call. Suspension will stack upon suspension until a within-function function call produces a result. Every recursively defined function must ensure that after some finite number of recursions there will be a branch to an executable line rather than to a further recursion. Such functions will be very slow in executing and while they are very useful for solving problems theoretically they are never as efficient as an equivalent iterative solution and we will not use them much in this course.

A.9.1 Relational operations

To be able to create branching code we need to be able to ask questions. For example, typical sorts of questions a code may ask are:

- Is the temperature greater than 100°C?
- Is the height of the building less then 700 m?
- Is that a car and is it's speed equal to 60 km/h?

Questions that can be asked in a program only allow for two possible answers `True` or `False`. MATLAB represents `True` with the integer 1 and `False` with the integer 0.

Relational operators are operators that compare two matrices of equal size. The operator does an element by element comparison between the two matrices and returns a matrix of the same size with the elements set to 1 if the comparison is true, and to 0 otherwise.

An expression that contains a relational operator is a logical expression because the result is a matrix containing ones and zeros that can be interpreted as true or false values, respectively.

Table A.14 contains a list of MATLAB's relational operators.

Example A.26: Consider the following MATLAB code:

```
a = [3 4 6];
```

Table A.15: MATLAB's logical operators.

Logical operator	Symbol
and	&
or	|
not	~

```
b = [3 5 1];

c = a < b;
d = a ~= b;
e = a > b;
```

The result of this code is that the variables c, d, and e have the values:

```
c = [0 1 0]
d = [0 1 1]
e = [0 0 1]
```

A.9.2 Logical operations

The logical expressions can be combined using the logical operators. The logical operators defined in MATLAB are *and*, *or* and *not*, and are represented by the symbols in Table A.15

Logical operators are used to combine logical statements created using the relational operators into more complex logical expressions; for example,

```
a<b & b<c
```

This operation is only valid if the two resultant matrices (represented by a<b and b<c) are the same size. The entry in the matrix represented by this logical expression is 1 (*true*) if the values in the corresponding entries in a, b, and c, are such that a<b<c; otherwise the entry is 0 (*false*).

When two logical expressions are joined by or, entries in the resulting matrix are 0 (*false*) if both expressions are false, otherwise it is 1 (*true*). If two expressions are joined by and, entries in the resulting matrix are 1 (*true*) if both expressions are true, otherwise it is 0 (*false*). Logical expression preceded by *not* reverses the value of the relational operation. For example, if a>b is true, then ~(a>b) is false. In this example, the relational operation is surrounded by brackets to modify the default precedence. See Table A.16 for a list of the order of logical operations.

As it does with mathematical expressions, MATLAB evaluates logical expressions are evaluated from left to write. Unlike mathematical expressions, if at any point in the calculation the logical answer is known then the rest of the logical expression is ignored. For example, if x=4 then the statement,

```
(x>3) |(cos(z)<0)
```

must evaluated to true, as the first conditional part is true. This means that the value of cos(z) need not be evaluated, and it is not. Once a logical statement evaluates to a known result, extra conditionals are ignored. The statement,

Table A.16: Precedence of operators

Operator	Precedence	Order of evaluation
()	1	Left to Right, Innermost first
~	2	Right to Left
^	3	Left to Right
* /	4	Left to Right
\	5	Right to Left
+ -	6	Left to Right
< > <= >= == ~=	7	Left to Right
& |	8	Left to Right

```
(x<3) & (exp(z)<1.0e4)
```

is false, irrespective of the value of `exp(z)`. MATLAB will not bother to check whether `exp(z)<1.0e4` because it is not needed.

MATLAB's logical operators can also be applied to variables, not just used with relational operations exclusively. For example, if the two matrices `A` and `B` are the same size, then the result of `A & B` will be a matrix with elements set to one where both `A` and `B` have nonzero elements and zeros otherwise. The logical operators *or* and *not* behave similarly.

Example A.27: If `A = [6 4 0 1]`, `B = [0 2 6 2]`, then

```
A&B = [0 1 0 1]
 ~A = [0 0 1 0]
A|B = [1 1 1 1]
A<B&A = [0 0 0 1]
```

Due to operator precedence the last example is equivalent to `(A<B)&A`

Exercise A.28: Let `A = [3 2 7 0]`, `B = [4 0 3 1]`, `C = [8 1 0 5]`. Without using MATLAB, solve the following using the precedence table and the definitions for the operators `~=`

```
~A;    A | C;    A <= B;    A+B > C;
~C+B;    C-4 > B;    A ~= 7;
```

Check your answers using MATLAB.

Exercise A.29: MATLAB also has many logical functions. That is, functions that return logical matrices. Using Table A.17 and the MATLAB `help` command, work out the result of the following commands. Do not use MATLAB to solve the problems, only to check your answers.

Let `A=[3 2 7 0]`, `B=[4 0 3 1]`, `C=[8 1 0 5]`, and `D=[]`. What is the result of:

```
find(B);  isempty(D);  ~isempty(A);  find(A>B);

any(C);  all(C);
```

Table A.17: Some of MATLAB's logical functions

Function	Description
`find(x)`	Returns the indices of the nonzero elements of the vector x. If x is a matrix it returns the row and column indices of the nonzero elements
`isnan(x)`	Returns a matrix with ones where the elements of x are Nans, zeros otherwise.
`isempty(x)`	Returns a one if x is an empty matrix, zero otherwise
`any(x)`	Returns true if any element of a vector is nonzero. If x is a matrix, it operates on the columns of the matrix returning a row vector.
`all(x)`	Returns true if all elements of a vector are nonzero. If x is a matrix, it operates on the columns of the matrix returning a row vector.
`exist('x')`	Returns 0 if the name does not exist, returns 1 if it is a variable, 2 if it is an M-file or file of unknown type. See help for more return values
`finite(x)`	Returns a matrix with ones where the elements of x are finite, zeros otherwise.
`isstr(x)`	Returns a one if x is a string, zero otherwise
`strcmp(x,y)`	Compares two strings,x and y, character by character and returns one if the two strings are identical, zero otherwise.

A.9.3 Branch controls (conditional statements)

The `if` statement

The `if` statement is the most widely used conditional statement. The form of a *simple* `if` statement is

`if` *logical expression*
  statement group
`end`

If the logical expression is true, the statements between the `if` statement and the `end` statement are executed. If the logical expression is false, then MATLAB immediately jumps to the statement following the `end` statement.

It is important to indent the statements within an `if` structure in order to see the control flow easily.

Example A.30: One important use of an `if` statement, is to ensure that only valid parameters are passed to functions:

```
if x > 0
   y = log(x);
end

if (x >= -1) & (x <= 1)
   y = acos(x)*180/pi;
end
```

`If` statements can be nested when required. For example

```
if logical expression 1
   statement group A
   if logical expression 2
      statement group B
   end
   statement group C
end
statement group D
```

Example A.31: The following function extends the code of Example A.24 to allow the coefficients of a number of quadratics with real coefficients to be entered at once and to return not only the zeros but also the vertices of the parabolas they represent. Checks are made for error conditions. For example, The coefficient of the x^2 term cannot be zero. All coefficient vectors must be the same length.

```
% Function :  quadroots
% Syntax   :  [x, v, M] = quadroots(A,B,C)
% Input    :  A = coefficients of  x^2
%             B = coefficients of  x
%             C = constant coefficients
%             in the quadratic polynomials
% Outputs  :  x = the solutions of Ax^2 + Bx + C = 0
```

```
%             v = the vertex coordinates
%                 both returned as 2 column matrices
%                 with one row for the results
%                 for each quadratic
%             M = the original coefficients returned
%                 in the form of a matrix with one row
%                 for each quadratic
% Source:     Leigh Brookshaw 15/3/2000
%             Last Modified: Walter Spunde, May, 2002

function [x, v] = quadroots(A,B,C)

% If function fails output is empty
x = [];  v = [];
% Data checking
% Check that all the coefficients have the same length
if (length(A) ~= length(B)) | (length(B) ~= length(C))
  error('Not all coefficient arrays are of equal length');
end

% Check that all the coefficients are real.  Tests such as
% these are madein case the coefficients have been passed
% to this function from some other computation.
if ~isreal(A)
  error('Not all elements of A are real!');
end
if ~isreal(B)
  error('Not all elements of B are real!');
end
if ~isreal(C)
  error('Not all elements of C are real!');
end
% Check that no value of A is zero
if find(A==0)
  error('No elements of A are allowed to be zero');
end

% Computing the Output
 M    = [A;B;C]';              % the coefficients in a matrix
 xs   = -B./(2*A);             % vertex abscissae
ords = A.*xs.^2 + B.*xs + C;  % ordinates of vertices
 v    = [xs;ords]';            % vertex coordinates
disc = B.*B - 4*A.*C;         % the value of the discriminant
 x   = ([xs;xs]+([1,-1]'*sqrt(disc))./[(2*A);(2*A)])' % zeros
end
```

The `if-else` statement

The `else` clause added to an `if` statement means that two blocks of code can be specified. One to be executed if the the logical expression is true, the other if it is false. For example

```
if logical expression
    statement group A
```

```
else
    statement group B
end
```

If the logical expression is true, then statement group A is executed. If the logical expression is false, then statement group B is executed. The statement groups can also contain `if` and/or `if...else` statements to provide a nested structure.

Example A.32: The velocity of a cable car between two towers. Assume that the scalar `r` is the distance from the nearest tower. If the cable car is within 30 metres of the tower, the velocity is calculated using the equation velocity = $0.425+0.00175\times r^2$. If the cable car is further away from the tower, the velocity is calculated using the equation velocity = $0.625 + 0.12 \times r - 0.00025 \times r^2$.

The correct velocity is calculated with the statements

```
if r <= 30
  velocity = 0.425 + 0.00175*r^2;
else
  velocity = 0.625 + 0.12*r - 0.00025*r^2;
end
```

Notice that control structures can be executed in the interactive environment, besides being used within functions.

The `if-elseif-else` statement

When several levels of `if...else` statements are nested, it may be difficult to determine which logical expressions must be true (or false) to execute each set of statements. In this case, the `elseif` clause is often used to clarify the program logic.

The `else` and `elseif` clauses are optional in an `if` statement. The general form of the `if` statement is:

```
if logical expression 1
    statement group A
elseif logical expression 2
    statement group B
elseif logical expression 3
    statement group C
    ⋮
else
    statement group D
end
```

If logical expression 1 is true, then only statement group A is executed. If logical expression 1 is false and logical expression 2 is true, then only statement group B is executed. If logical expressions 1 and 2 are false and logical expression 3 is true, then statement group C is executed. If none of the logical expressions are true then

statement group D is executed. If more than one logical expression is true, the first logical expression that is true determines which statement group is executed.

The `switch` statement

The `switch` statement provides an alternative to an `if...elseif...else` construct. Though anything programmed using a `switch` statement can be programmed using an `if` structure, often the code is more readable using the former.

The structure of the `switch` statement is:

```
switch input expression
   case value 1
      statement group A
   case value 2
      statement group B
      ⋮
   otherwise
      statement group D
end
```

The *input expression* of the commswitch statement must evaluate to a scalar or a string. The value of the *input expression* is compared to each `case` value. If a match is found then the statements following the `case` statement are executed.

At most only one `case` statement group is executed, then execution resumes with the statement after the `end` statement.

If no `case` statement matches the *input expression* then the statements following the `otherwise` statement are executed. The `otherwise` statement is optional.

Below is an example of a case statement, where the `switch` variable `direction` contains a character string (`lower` is a MATLAB primitive function that takes a text as input and returns it all in lower case.)

```
switch lower(direction)
   case 'north'
      angle = 0;
      disp('Direction: North');
   case 'south'
      angle = pi;
      disp('Direction: South');
   case 'east'
      angle = 0.5*pi;
      disp('Direction: East');
   case 'west'
      angle = 1.5*pi;
      disp('Direction: West');
   otherwise
      angle = NaN;
      disp('Direction: Unknown');
```

```
end
```

A.9.4 Iteration controls (explicit loops)

MATLAB contains two constructs for explicitly looping, the `for` statement and the `while` statement. The `for` statement is used when the number of times the block of code must be looped over is known. The `while` statement is used when the number of times the block of code must be looped over is unknown. The looping is terminated when a specified condition is satisfied.

The `for` statement

The general form of the `for` statement in MATLAB is as follows:

```
for index = expression matrix
   statement group
end
```

The statements in the statement group are repeated as many times as there are columns in the expression matrix (which could also be a scalar or a vector). Each time through the loop, the index has the value of one of the columns in the expression matrix.

Some examples of the `for` statement:

- calculating the factorial of 20

```
factorial = 1;
for i=2:20
   factorial = factorial*i;
end
```

  Although this computation is far better done by simply `prod(1:20)`.

- Multiply the elements of an array

```
product = 1;
A = [6 8 9 10 2 -5 1 7];
for i=A
   product = product*i;
end
```

  This is also much better done by `prod(A)`.

The rules for a `for` loop are the following:

- The `index` of the `for` loop must be a variable.
- If the expression matrix is the empty matrix, the loop will not be executed. Control will pass to the first statement following the `end` statement.

- If the expression matrix is a scalar, then the `for` loop will be executed once only. The `index` variable will contain the value of the scalar.
- If the expression matrix is a row vector, then each time through the loop, the index will contain the next value in the the vector.
- If the expression matrix is a matrix, then each time through the loop, the index will contain the next *column* in the the matrix.
- On exiting the `for` loop, the `index` contains the last value used.
- The colon operator can be used to define the expression matrix. The syntax is:

  ```
  for index=initial:increment:limit
  ```

  The number of times the loop will be executed can be calculated from the equation:

  $$\text{floor}\left[\frac{\texttt{limit} - \texttt{initial}}{\texttt{increment}}\right] + 1$$

  If this value is negative, the loop is not executed. If `increment` is omitted, it is assumed to be one.
- Lastly, and *the most important rule of all, a* `for` *loop must never, never, ever be the innermost loop in a computation.* The innermost loop must always, always, always be a natural vector/matrix command. Two examples, are the two examples given above: the `for` loop to compute a product must be replaced by the `prod` function. See further discussion in Section A.10.

Example A.33: Suppose the cable car distance from the pylon `r`, of Example A.32, is a vector. Then Example A.32 will have to modified to calculate a vector of velocities.

```
% Initialize the velocity array to same size as 'r'
% by using the 'size' command.
velocity = zeros(size(r));

% Loop over the distance array and calculate velocities
for k = 1:length(r)
   if r(k) <= 30
      velocity(k) = 0.425+0.00175*r(k)^2;
   else
      velocity(k) = 0.625+0.12*r(k)-0.00025*r(k)^2;
   end
end
```

However, the above code has a `for` *loop as the innermost loop and so violates the last criterion above. Rewrite it using natural matrix/vector modes of computation.* After initialising the `velocity` to a correct size matrix, simply

```
k=find(r<=30);
velocity(k) = 0.425+0.00175*r(k).^2;
k=find(r>30);
velocity(k) = 0.625+0.12*r(k)-0.00025*r(k).^2;
```

Use the `for` loop only when the inside of the loop is less trivial.

The `while` statement

The `while` loop is a structure for repeating a set of statements as long as a specified condition is true. The general form of the `while` loop is:

```
while expression
   statement group
end
```

If the *expression* is true or all elements nonzero, then the statement group is executed. The *expression* is normally a logical expression. After the statements are executed, the condition is re-tested. If the condition is still true, the group of statements are executed again. When the condition is false, control jumps to the first statement following the `end` statement. If the expression is always true (or is a value that is nonzero), the loop becomes an infinite loop!

Example A.34:

```
% To sum the positive elements of the vector x
% use the 'length' command to find the
% number of elements in x

sum = 0; k = 1;
while (k <= length(x)) & (x(k) >= 0)
   sum = sum + x(k);
   k = k + 1;
end
```

Of course, again this violates our last criteria. Instead, in practise code simply

```
sumx=sum(x(find(x>=0)));
```

Use `while` loops when the computation inside the loop is not so simple.

Exercise A.35: Write a function containing the code of Example A.34. Test the function and also modify it so that the condition statement of the while loop is reversed, that is

```
while (x(k) >= 0) & (k <= length(x))
```

Now test the function again. Explain why is there now an error in the script.

A danger of `while` loops is that they may never terminate. This obviously occurs if the condition is always true. A loop counter can be added to the condition to ensure that the loop terminates. For example, in the code below, the termination condition we are interested in is `(a>=0)`, this is assumed to occur within 100 loops, if it does not then an error has occurred. The variable `count` ensures the loop will not be infinite.

```
count = 1;
while (a < 0)
```

```
  ...
  count = count + 1;
  if count > 100
     error('Loop failed to terminate!');
  end
end
```

To break out of an infinite loop, and terminate execution of a script or function, you can press `Control C` on the keyboard. That is, hold down the `Ctrl` key and press `C` simultaneously.

Exercise A.36: Timber management needs to determine how much of an area to leave uncut so that the harvested area is re-forested in a certain period of time. Assume that the reforestation takes place at a known rate per year depending on climate and soil conditions. If the reforestation rate is η, then the reforestation equation is (Forested area this year) = (Forested area last year)$\times (1+\eta)$. Assume that there are 11,000 hectares total, with 4,500 hectares uncut, and the reforestation rate is 0.03 hectares/year. Write a function that returns a reforestation vector with the following help lines in its definition.

```
% Function: reforest
% Syntax  : hectares = reforest(years, rfrate, inhec)
% Input:
%       years  - the number of years to calculate
%                the reforestation
%       rfrate - the reforestation rate in hectares/year
%       inhec  - the initial forest area in hectares.
% Output:
%       hectares -  a vector that lists the forest area
%                   in hectares for each year. The length
%                   of the vector is the same as the
%                   number of years input parameter.
%
function hectares = reforest(years, rfrate, inhec)
```

Exercise A.37: Write a script to print a table showing the number of hectares reforested at the end of each year, after asking the user to enter the number of years to be used in the table.

Exercise A.38: Modify the program of Exercise A.36 so that the user can enter the number of hectares, and the program will determine how many years are required for the number of hectares to be re-forested.

The `break` statement

Use the `break` statement to break out of a `for` loop or a `while` loop before it has completed. This statement is useful if an error condition is detected within the loop. This statement is also useful for terminating an iteration when the desired result has been computed. The loop is terminated at the point where the `break` command is executed.

A.10 Vector Programming

If you are familiar with more traditional programming languages such as FORTRAN, Javascript, Python, C or C++, the discussion on looping in the previous section will be familiar. The `for` and `while` looping discussed has its counterparts in many languages. However, since MATLAB was designed to make matrix manipulations easy one would not expect to have to use loops to implement operations like matrix multiplication or vector addition but to find them encapsulated in primitive functions. It turns out that these primitives find surprising uses and that in many situations where a classical scalar language needs loops to complete a calculation, array processing primitives handle the computation without any apparent looping.

For example, consider the following traditional scalar code that multiplies together all the positive elements of a vector:

```
product = 1; k = 1;
while (k <= length(x)) & (x(k) > 0)
   product = product * x(k);
   k = k + 1;
end
```

A bit of research through MATLAB help finds there is a function `prod` that will return the product of all the elements of a vector or array. But in our case we don't want all the elements we want only the positive elements.

One possibility is the following scalar code—

```
new = []; k = 1;
while (k <= length(x)) & (x(k) > 0)
   new = [new x(k)];
   k = k + 1;
end
prod(new);
```

to create a new array containing only those values greater than zero which is then passed to `prod`

An alternate method is to avoid explicit loops and use the fact that MATLAB was designed to work with arrays implicitly—

```
bool = (x > 0);
indices = find(bool);
new = x(indices);
prod(new);
```

- `bool` is created by evaluating the logical expression for each value of `x`. This creates a logical array with the same dimensions as `x`.
- `find(bool)` returns an array containing the indices where `bool` is non-zero. These are the same indices where `x>0`.
- Create a new array containing only those values in `x` with the indices found in the array `indices`.

This can be written more concisely without the intermediate variables as:

```
prod(x(find(x>0)));
```

See also Examples A.33 and A.34.

`find` These codes use the very useful function `find` which is used to create a vector `k`, that contains the indices of all the elements in a passed array that are non-zero. If the passed array is created from a logical expression `find` will return the indices only of those value that are true.

For example, consider a vector $\boldsymbol{x}$ that is defined as follows:

$$\boldsymbol{x} = \{7, -3, 2, 4, -5, -6\}.$$

The command `k=find(x>0)` returns the following:

```
>      k=find(x>0)
k =
     1     3     4
```

This means that elements x_1, x_3 and x_4 are the only positive values in vector $\boldsymbol{x}$.

To create a new vector containing only those positive values then the statement

```
>      xp=x(k)
xp =
     7     2     4
```

will do it. If `k` was a scalar this statement would be perfectly valid in any language—but because MATLAB allows arrays to be used anywhere are scalar can be used then when `k` is an array the statement `x(k)` is an *implicit loop* and behaves as you would expect.

MATLAB was designed to process matrix operations rapidly. The more traditional `for` and `while` loops have to be interpreted which takes an enormous amount of time. Implied loops utilize specially designed operators within MATLAB. This means the processing is extremely rapid. An implied loop in MATLAB executes up to a hundred times faster than the equivalent explicit `for` or `while` loop.

A disadvantage of the implied loop is that MATLAB code can become terse and hard to understand. If you develop your skill in thinking in vector, matrix and array terms then this brevity will soon appear as elegance of expression and the scalar version will seem clumsy and pedestrian. You should make every effort to think in these terms as it constitutes 'vector processing' at the software level.

Example A.39: Consider the following code

```
newv=zeros(size(vec))
myprod=1;
for i=1:length(vec)
 myprod = myprod * vec(i);
 newv(i) = myprod;
end
newv
```

Assuming that the variable `vec` is a predefined vector this code will create a vector of the same dimension which contains the cumulative product of the elements of `vec`—

- The first line creates a new vector `newv` filled with zeros, and the the same dimensions as `vec`
- Loop over all the indices of `vec` calculating the cumulative product and storing it in `newv`.

This can be achieved in one line with the command:

```
newv = cumprod(vec);
```

Exercise A.40: The MATLAB functions `sum`, `prod`, `cumsum`, and `cumprod` have well defined meanings when the passed array is a vector—What if the passed array is multi-dimensional? What is returned?

Example A.41: Write a function `repvec` the is passed a vector and the number of times each element is to be duplicated. The function should return the resulting vector. For example,

```
> repvec([1 -2 3 -4], 2)
ans=
    1   1 -2 -2   3   3 -4 -4
```

The problem with dealing with arrays as distinct from scalars is that you must check the dimension of the passed variable—operations that will work on a column vector may not work on a row vector!

```
function repvec(v,n)
%Force a to be a row vector
v = v(:)';
%Find the length of the output vector
l = length(v)*n
%Construct output vector
reshape(v(ones(1,n),:),1,l);
end
```

Notes:

- Accessing a single element in an array you use the element's index, to access an entire row or column use the colon (`:`) operator (See §A.4.3). The notation `v(:)` returns all the elements of the vector `v` as a single column (irrespective of whether it is a column or row vector). So `v(:)'` will always return a row vector. This assumes that `v` is a vector! What if it is not?
- `lengthv` returns the number of elements in the vector `v` (Compare this with the function `size` which returns the dimensions of a matrix).
- To select the first row of a matrix you use the notation `m(1,:)` (first row, all columns). Same for the first (only) row of a vector `v(1,:)`. But in MATLAB anywhere we use a scalar we can use a matrix—so what

does `v([1 1 1],:)`? It means "select the first row three times" and return a matrix with the three rows repeated:

```
> a=[1 2 3 4];
> a([1 1 1],:)
ans =
  1 2 3 4
  1 2 3 4
  1 2 3 4
```

- The function call `ones(1,n)` returns a matrix with dimensions `(1,n)` filled with ones. So `v(ones(1,n),:)` returns a matrix with `n` rows filled with copies of `v`.
- The function `reshape` takes as input a matrix and new dimensions and reshapes the passed matrix to the new dimensions. The number of elements in each matrix must be the same.

Example A.42: A company is calibrating a precision instrument by measuring the radius and height of one cylinder 10 times; the measurements are stored in the vector variables `r` and `h`. Write vectorised code to find the volume from each trial, given by $\pi r^2 h$. Also some measurements are known to be invalid (< 0) these need to be removed.

One possible vectorised solution is:

```
k = find(r>0 & h>0);
v = pi*r(k).^2.*h(k)
```

Notes:

- First find the indices of the vectors where both `h` and `r` are positive
- Then use the index vector returned from `find` to only calculate the volume for valid radius and height values.
- The length of `v` may be smaller then the length of `r` and `h`.

Example A.43: The mathematician Euler proved the following:

$$\frac{\pi^2}{6} = 1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} + \ldots \tag{A.1}$$

verify whether the above conjecture is true.

One way to verify the conjecture is to progressively calculate the sum and show that it is getting closer to the limit—

```
n = 100;
terms = 1./([1:n].^2);
error = pi*pi/6-cumsum(terms);
plot(error)
```

Example A.44: Write a function called `geoser` that will be passed the values of `r` and n. and will calculate and return the sum of the geometric series:

$$1 + r + r^2 + r^3 + r^4 + \ldots + r^n \tag{A.2}$$

```
function [s] = geoser(r,n)
% Function to calculate the sum of
% the geometric series of n terms in r
% input:
%       r: scalar value
%       n: (n+1) the number of terms

% test input parameters
% r must be scalar
  if length(r)~=1
     s=0;
     return;
  end
% n must be positive scalar
  if length(n)~=1 | n<0
     s=0;
     return;
  end
% Simple case
  if n==0
     s=1;
     return;
  end

% Start Calculation
% Create a vector containing the value r
  terms = r*ones(1,n);
% Calculate the cumulative product - this
% gives us the powers in r.
  series = cumprod(terms);
% Sum and add 1
  s = 1+sum(series);
end
```

Example A.45: The example above assumes that `r` will be a scalar—but what if it is a vector? How does the code need to be altered to deal with this case?

```
function [s] = geoser(r,n)
% Function to calculate the sum of
% the geometric series of n terms in r
% input:
%       r: vector value
%       n: (n+1) the number of terms
% output:
```

```
%       s: the gemetric sum with the
%          same dimensions as r.

% test input parameters
% r must be a vector only
  [x,y] = size(r);
  if x~=1 & y~= 1
     s=zeros(x,y);
     return;
  end
% n must be a positive scalar
  if length(n)~=1 | n<0
     s=zeros(x,y);
     return;
  end
% Simple cases
  if n==0
     s=ones(x,y);
     return;
  end
  if n==1
     s=1+r;
     return;
  end

% Start Calculation
% Must have a row vector so force it
  r = r(:)';
% Create an array containing the values r
% each r repeated down columns
  terms = r(ones(1,n),:);
% Calculate the cumulative product - this
% gives us the powers in r down columns
  series = cumprod(terms);
% Sum down the columns and add 1
  s = 1+sum(series);
  s = reshape(s,x,y);
end
```

Notes:

- By forcing `r` to be a row vector we can then repeat each value down columns.
- The choice of row vector was made because `cumprod` (unless told otherwise) will calculate the cumulative product down columns only. Same with `sum`

A.11 Using functions and scripts together

As a mini-application a script can be used as a *driver* for a particular function. As such it can interrogate a user for the required inputs and make the function call with the correct syntax. This use is particularly attractive when the syntax for a function is rather tedious to find or remember. Because of the danger of over-writing workspace variables when one uses a script, if scripts are going to be used in this way to drive functions then it is a wise practice to either give the variables long and unlikely names or as the first interaction with the user to check that the names used can be safely over-written. For example, in a workspace for calculus computations the names `d` and `S` might very well be used for functions that compute differences between list elements and sequences of partial sums of series. Such names will be very convenient in that setting so that one could write, for example, `d(y)./d(x)` for derivative approximations, and `S(y.*d(x))` for approximations to the integral (as we shall do in other modules of this course). Of course these simple single character names are in grave danger of being over-written by careless assignments in the interactive session or a script.

Exercise A.46: A small rocket is being designed to make wind shear measurements in the atmosphere. Before testing begins, the designers have developed a model of the rocket's trajectory. The height in metres above the ground at time t in seconds is

$$\text{height (metres)} = 20 + 2.13\,t^2 - 0.0013\,t^4 + 0.000034\,t^{4.751}$$

Write a function that takes the time in seconds and returns the height. It should return a vector of heights when the time is given as a vector.

Write a driver script to run your function.

Exercise A.47: Using the function from Exercise A.46, write a script to compute the height of the rocket from $t = 0$ to the time it hits the ground. Use increments of one second. If the rocket has not hit the ground after 80 seconds, stop the program and print out an error message. Have the script print out a table of times and heights of a size determine by the user and on request plot the results.

Exercise A.48: Using the function from Exercise A.46, write a program to compute the time at which the rocket reaches the highest part of its trajectory and the time at which it impacts the earth. Both should be accurate to one second.

A.12 Numerical programming

A.12.1 Modular Programs

Any problem, that can be solved by a computational program, can be solved in one monolithic program contained within one file. In fact, since a program is just a string of bits, we could convert this bit string into a number to base 10 and present this single number as an solution to the problem. This is not particularly helpful if you have the job of maintaining the program (that is, modifying it to meet current needs).

There are a number of reasons for avoiding a monolithic program:

- the project as a whole, and it could be a large one, has to be understood in its entirety to write a monolithic program. If it is not understood in its entirety then many rewrites will be necessary to incorporate all aspects of the project.
- large cumbersome files are difficult to work with.
- large monolithic programs are difficult to understand, for the author and anyone else.

Sensible programming attempts to identify the key operations required to solve the problem and the tools that will help in putting the solution together. These tasks are of course solved by running functions. The basic rule about functions is that each function should perform one task and one task only. The project analysis should be task oriented, each task coded and tested separately. The main program pulls together these disparate pieces with a script that mainly contains calls to functions. The advantage of this approach is:

- many programmers can work on the problem simultaneously.
- a monolithic project is broken down into smaller manageable chunks.
- if each function performs one task, then the final code is easier to read and understand.
- it is easier to work with smaller files.

Functions themselves should also be written in a block like manner. That is similar tasks should appear together in the function. Initialization of all variables should appear together at the start of the function, all checks on input variables should appear together. For example:

1. Help comments: name and keywords; syntax, inputs and outputs
2. `function [output variables] = fname(input variables)`
3. Test that the input variables are valid.
4. Initialize all variables that need it.
5. Load any data that might be needed.
6. Computational algorithm for function.

The above is an outline for the design of a function. Particular circumstances will dictate the design of any individual function, but the steps above should be borne in mind whenever a function is written.

A.12.2 Debugging

Debugging a program is the process of finding and removing the *bugs* or errors. Such errors usually fall into one of the following categories:

- *Syntax errors* such as omitting a parenthesis or comma or spelling a command name incorrectly. MATLAB usually detects the more obvious errors and displays a message describing the error and its location.
- *Runtime errors* occur while the program is running and fall into a number of categories:
  - Errors due to incorrect logic. They do not necessarily occur every time the program is executed; the occurrence may depend on the type of input data. A common example is division by zero. A logic error is caused by the author not anticipating all possible eventualities.
  - Errors due to incorrect method. This can be a very subtle error, recognising that the numerical method being used is incorrect for the problem being solved. Normally this can only be fixed by rewriting the program using a better method for the problem. Unfortunately, this error might only be recognised after an immense amount of effort has been expended writing the program. It is important when deciding on which numerical method to use to solve a problem that the limitations of the method are well understood.

The MATLAB error messages usually allow you to find syntax errors. However, runtime errors are more difficult to locate and are more difficult to spot. It is important to check *all* answers produced by programs. Never trust the answer from a program. Some tests that should always be employed with computer solutions are:

- always check the order of magnitude of the solution. Is it reasonable?
- if the solution has units, check that the order of magnitude of the solution is reasonable for those units.
- plot the answers, and see if the plot looks reasonable.

If you have tested the solution and think there is an error, then to locate the error, try the following:

- Always test your program with a simple version of the problem, whose answer can be checked by hand calculations. If your program uses functions, test the functions individually, by writing test scripts for each function.
- Display any intermediate calculations by removing semicolons at the end of statements.
- Use the M-file Debugger. The MATLAB Debugger is a set of commands that allow you to stop a script during execution and examine or modify variables.

Table A.18: Some of the debug commands available in MATLAB. Use the command `help debug`, for a complete list of commands.

Name	Description
`dbstop`	This command is used to temporarily stop the execution of an M-file. This command creates *breakpoints* in the M-file where execution will stop. When a breakpoint is encountered the user is placed in the debug command line environment where variables can be examined or changed.
`dbstep`	This command allows the user to execute one or more lines of an M-file. When the lines have been executed MATLAB reverts back to the debug command line environment.
`dbcont`	This command resumes execution of the M-file following a stop in execution. Execution will continue until either another breakpoint is encountered, or the M-file stops normally.
`dbclear`	Clear breakpoints set by `dbstop`.
`dbtype`	List an M-file with line numbers. This will assist when setting breakpoints.
`dbquit`	This command terminates debug mode immediately so that the M-file will continue to run as if the debugger had never been invoked.

Matlab debugger

From version 5 MATLAB has been supplied with a useful debugger. A debugger is a program that allows the programmer to control execution of an M-file, either a script M-file or a function M-file. Table A.18 lists some of the commands available with the MATLAB debugger.

Suppose you have a MATLAB script called `quadroot.m` that you which to debug. The the first step is to ensure that when the script runs the debugger is activated. This can be done by ensuring that a breakpoint is set in the script before it is run. To do this type the command:

```
dbstop in quadroot;
```

This means that a breakpoint will be set at the first executable line of the M-file. When the script is run, execution will stop (break) before the first line is executed and control will be given to the debugger. Inside the debugger, you can type the command

```
dbtype;
```

which will list the lines of the currently active M-file. The advantage of the `dbtype` command is it adds line numbers to the output. This means you can now set breakpoints in the script using line numbers. For example,

```
dbstop in quadroot at 46;
dbstop at 46;
```

both of these commands will set breakpoints at line 46 of M-file `quadroot.m`. They are equivalent commands because the debugger recognizes it is currently in `quadroot.m`.

When execution stops at a breakpoint, control is given back to the debugger. The debugger looks exactly like the normal command line interface. This means that you can display the contents variables have at that point of the code, you can modify the contents of variables, in fact, you can execute any MATLAB command at this point.

The execution of your code can be resumed line by line with the `dbstep` command or continue until the next breakpoint with the `dbcont` command.

Activity A.8 → Use the MATLAB `help debug` and read the help information on using the debugger.

Exercise A.49: Use the debugger on one of your working scripts. Set breakpoints with the `dbstop` command, display variables, step through your code using both the `dbstep` and `dbcont` commands.

Matlab Commands