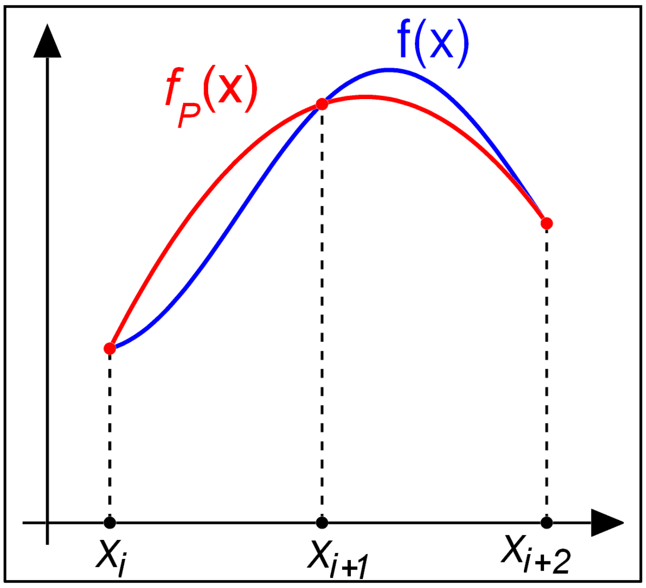
University College Dublin
An Coláiste Ollscoile, Baile Átha Cliath

School of Mathematical Sciences
Scoil na nEolaíochtaí Matamaitice

Computational Science (ACM 20030)



Dr Lennon Ó Náraigh

Lecture notes in Computational Science, January 2014

Computational Science (ACM 20030)

- Subject: Applied and Computational Maths
- School: Mathematical Sciences
- Module coordinator: Dr Edward Cox; Lecturer: Dr Lennon Ó Náraigh
- Credits: 5
- Level: 2
- Semester: Second

Typically, problems in Applied Mathematics are modelled using a set of equations that can be written down but cannot be solved analytically. In this module we examine numerical methods that can be used to solve such problems on a desktop computer. Practical computer lab sessions will cover the implementation of these methods using mathematical software (Matlab). No previous knowledge of computing is assumed.

Topics and techniques discussed include but are not limited to the following list. **Computer architecture:** The Von Neumann model of a computer, memory hierarchies, the compiler. **Floating-point representation:** Binary and decimal notation, floating-point arithmetic, the IEEE double precision standard, rounding error. **Elementary programming constructions:** Loops, logical statements, precedence, array operations, vectorization. **Root-finding for single-variable functions:** Bracketing and Bisection, Newton–Raphson method. Error and reliability analyses for the Newton–Raphson method. **Numerical integration:** Midpoint, Trapezoidal and Simpson methods. Error analysis. **Solving ordinary differential equations (ODEs):** Euler Method, Runge–Kutta method. Stability and accuracy for the Euler method. **Linear systems of equations:** Gaussian elimination, partial pivoting. **The condition number of a matrix:** quantifying the idea that a matrix can be 'almost' singular, investigating the consequences of this idea for the robustness of numerical solutions of linear systems. **Fitting data to polynomials** using the method of least squares. **Random-number generation** using the linear congruential method.

What will I learn?

On completion of this module students should be able to

1. Describe the architecture of a modern computer using the Von Neumann model.
2. Describe how numbers are represented on a computer.
3. Use floating-point arithmetic, having due regard for rouding error.
4. Do elementary operations in Matlab, such 'for' and 'while' loops, logical statements, precdence.
5. Do array operations using loops; and equivalently, using vectorization.
6. Describe elementary root-finding procedures, analyse their robustness, and implement them in Matlab.
7. Describe elementary numerical integration integration schemes, analyse their accuracy, and implement them in Matlab.
8. Solve ODEs numerically uzing standard algorithms, analyse their accuracy and stability, and implement them numerically.
9. Solve systems of linear equations using Gaussian elimination.
10. Analyse ill-conditioned systems of equations.
11. Fit data to polynomials.

Editions

First edition: January 2013

This edition: January 2014

Contents

Chapter 1

Introduction

1.1 Module summary

Here is the executive summary of the module:

You will learn enough numerical analysis to enable you to solve ODEs, integrate functions, find roots, and fit curves to data. At the same time, you will learn the basics of Matlab. You will also learn about Matlab's powerful built-in functions that make numerical calculations effortless.

In more detail, we will follow the following programme of work:

1. The architecture of a modern computer: Von Neumann model, memory hierarchies.
2. Represetation of numbers on a computer: binary versus decimal. Floating-point arithmetic. Rounding error.
3. Elementary operations in Matlab: 'for' and 'while' loops, logical statements, precdence.
4. Array operations using loops; the superseding of these loop calculations by vectorization.
5. Root-finding: the Intermediate Value Theorem, Bracketing and Bisection, Newton–Raphson method.
6. Failure analysis for the Newton–Raphson method, including analysis of iterative maps.
7. Numerical integration (quadrature) using the Midpoint, Trapezoidal, and Simpson's rules. Error analysis for the same.
8. Solving ODEs numerically: Euler and Runge–Kutta methods. Error analysis for the Euler method. Stability analysis for the same.

9. Solving systems of linear equations using Gaussian elimination.
10. Analysis of ill-conditioned systems (i.e. systems of linear equations that are 'barely solvable'). The condition number.

1.2 Learning and Assessment

Learning

- 36 contact hours, 3 per week, with the following possibilities:
  - Three hours of lecturers (theory), no computer-aided labs;
  - Two hours of lectures, one hour of labs;
  - One hour of lectures, two hours of labs.

  The split will happen on an ad-hoc basis as the module progresses.

  Note finally, there will be precisely three contact hours per week, in spite of appearances to the contrary on the official timetable.
- The lab sessions will involve using the mathematical software Matlab. No prior knowledge of Matlab or programming is assumed. The students will be taught how to use Matlab in these lab sessions.
- Supplementary reading and Matlab coding practice.

Assessment

- Three homework assignments, $6\frac{2}{3}\%$ each, for a total of 20%
- **One** midterm exam, for a total of 20%
- One end-of-semester exam, 60%

Note that percentage-to-grade conversion table is the one used by the School of Mathematical Sciences, see

`http://mathsci.ucd.ie/tl/grading/en06`

Resitting the module

Assessment of resit students will be by one end-of-semester exam only, which will be assessed in the usual way on a pass/fail basis.

Textbooks

- Lecture notes will be put on the web. These are self-contained. They will be available *before* class. It is anticipated that you will print them and bring them with you to class. You can then annotate them and follow the proofs and calculations done on the board in class.
- The lecture notes will also be used as a practical Matlab guide in the lab-based sessions.
- You are still expected to attend all classes and lab sessions, and I will occasionally deviate from the content of the notes, and give revision tips for the final exam.
- Here is a list of the resources on which the notes are based:
  - *Afternotes on Numerical Analysis*, G. W. Steward, (SIAM, 1996).
  - For issues concerning numerical linear algebra: Dr Sinéad Ryan's website:

    `http://www.maths.tcd.ie/~ryan/TeachingArchive/161/teaching.html`
  - For issues concerning computer architecture and memory, the course *Introduction to high-performance scientific computing* on the website

    `www.tacc.utexas.edu/~eijkhout/Articles/EijkhoutIntroToHPC.pdf`
- Other, more advanced works are referred to very occasionally:
  - *Chebyshev and Fourier Spectral Methods*, J. P. Boyd (Dover, 2001), and the website

    `http://www-personal.umich.edu/~jpboyd/BOOK_Spectral2000.html`
  - *The art of Computer Programming*, Volume 2, D. Knuth (Addison-Wesley, 3rd Edition, 1997)
  - *Numerical Recipes in C*, W. H. Press *et al.* (CUP, 1992):

    `http://apps.nrbook.com/c/index.html`

Module dependencies

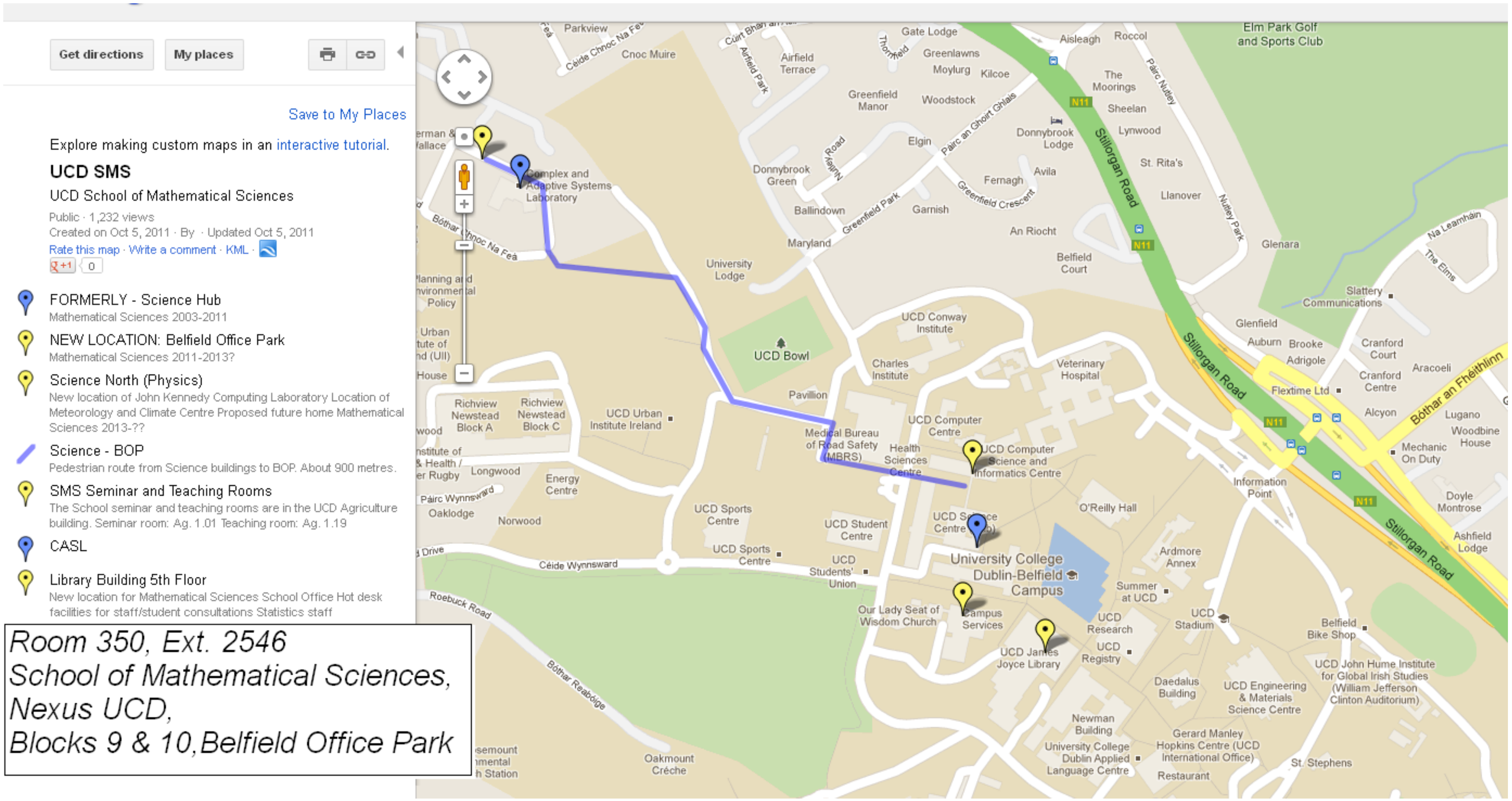
Some knowledge of Linear Algebra and Calculus is assumed. Important theorems in analysis are referred to. For a reference, see the book *Analysis: An Introduction*, R. Beals (CUP, 2004).

Office hours

I do not keep specific office hours. If you have a question, you can visit me whenever you like – from 09:00-18:00 I am usually in my office if not lecturing. It is a bit hard to get to. The office number, building name, and location are indicated on a map at the back of this introductory chapter.

Otherwise, email me:

`onaraigh@maths.ucd.ie`


Get directions · My places

Save to My Places

Explore making custom maps in an interactive tutorial.

UCD SMS

UCD School of Mathematical Sciences

Public · 1,232 views
Created on Oct 5, 2011 · By · Updated Oct 5, 2011
Rate this map · Write a comment · KML ·
+1 0

FORMERLY - Science Hub
Mathematical Sciences 2003-2011

NEW LOCATION: Belfield Office Park
Mathematical Sciences 2011-2013?

Science North (Physics)
New location of John Kennedy Computing Laboratory Location of Meteorology and Climate Centre Proposed future home Mathematical Sciences 2013-??

Science - BOP
Pedestrian route from Science buildings to BOP. About 900 metres.

SMS Seminar and Teaching Rooms
The School seminar and teaching rooms are in the UCD Agriculture building. Seminar room: Ag. 1.01 Teaching room: Ag. 1.19

CASL

Library Building 5th Floor
New location for Mathematical Sciences School Office Hot desk facilities for staff/student consultations Statistics staff

Room 350, Ext. 2546
School of Mathematical Sciences,
Nexus UCD,
Blocks 9 & 10, Belfield Office Park

Stillorgan Road
N11
Elm Park Golf and Sports Club
University College Dublin-Belfield Campus
UCD Bowl
UCD Computer Science and Informatics Centre
UCD Computer Centre
Health Sciences Centre
Medical Bureau of Road Safety (MBRS)
UCD Science Centre
UCD Student Centre
UCD Students' Union
Our Lady Seat of Wisdom Church
Campus Services
UCD James Joyce Library
UCD Research
UCD Registry
O'Reilly Hall
Veterinary Hospital
UCD Conway Institute
Charles Institute
Pavilion
UCD Sports Centre
UCD Urban Institute Ireland
Complex and Adaptive Systems Laboratory
Richview Newstead Block A
Richview Newstead Block C
Energy Centre
Information Point
UCD Stadium
Daedalus Building
UCD Engineering & Materials Science Centre
Newman Building
Gerard Manley Hopkins Centre (UCD International Office)
University College Dublin Applied Language Centre
UCD John Hume Institute for Global Irish Studies (William Jefferson Clinton Auditorium)
Belfield Bike Shop
Belfield Court
Céide Wynnsward
Roebuck Road
Bóthar Reabóige
Oakmount Crèche
University Lodge
Bóthar an Fhéithlinn
St. Stephens
Restaurant

Chapter 2

Floating-Point Arithmetic

Overview

Binary and decimal arithmetic, floating-point representation, truncation, truncation errors, IEEE double precision standard

2.1 Introduction

Being electrical devices, ‘on’ and ‘off’ are things that all computers understand. Imagine a computer made up of lots of tiny switches that can either be on or off. We can represent any number (and hence, any information) in terms of a sequence of switches, each of which is in an ‘on’ or ‘off’ state. We do this through **binary arithmetic**. An ‘on’ or an ‘off’ switch is therefore a fundamental unit of information in a computer. This unit is called a **bit**.

2.2 Positional notation and base 2

One of the crowing achievements of human civilization is the ability to represent arbitrarily large and small **real numbers** in a compact way using only ten digits. For example, the integer $570,123$ really means

$$570,123 = (5 \times 10^5) + (7 \times 10^4) + (0 \times 10^3) + (1 \times 10^2) + (2 \times 10^1) + (3 \times 10^0)$$

Here,

- The leftmost digit (5) has five digits to its right and therefore comes with a power 10^5,

- The digit second from the left (7) has four digits to its right and therefore comes with power of 10^4,

- And so on, down to the rightmost digit, which, by definition, has no other digits to its right, and therefore comes with a power of 10^0.

In contrast, the Romans would have struggled to represent this number:

$$570,123 = \overline{\mathrm{D}}\,\overline{\mathrm{L}}\,\overline{\mathrm{X}}\,\overline{\mathrm{X}}\,\mathrm{C\,X\,X\,I\,I\,I},$$

where the overline means multiplication by $1,000$.

Rational numbers with absolute value less than unity can be expressed in the same way, e.g. 0.217863:

$$0.217863 = (2 \times 10^{-1}) + (1 \times 10^{-2}) + (7 \times 10^{-3}) + (8 \times 10^{-4}) + (6 \times 10^{-5}) + (3 \times 10^{-6}).$$

Other rational numbers have a decimal expansion that is infinite but consists of a periodic repeating pattern of digits:

$$\begin{aligned} \tfrac{1}{7} = 0.142857142857\cdots &= (1\times 10^{-1})+(4\times 10^{-2})+(2\times 10^{-3})+(8\times 10^{-4})+(5\times 10^{-5})+(7\times 10^{-6}) \\ &+ (1 \times 10^{-7}) + (4 \times 10^{-8}) + (2 \times 10^{-9}) + (8 \times 10^{-10}) + (5 \times 10^{-11}) + (7 \times 10^{-12}) + \cdots \end{aligned}$$

Using geometric progressions, it can be checked that $1/7$ does indeed equal $0.142857142857\cdots$, since

$$\begin{aligned} 0.142857142857\cdots &= 1\left(\frac{1}{10} + \frac{1}{10^7} + \frac{1}{10^{13}} + \cdots\right) + 4\left(\frac{1}{10^2} + \frac{1}{10^8} + \cdots\right) + \\ &\quad + 2\left(\frac{1}{10^3} + \frac{1}{10^9} + \cdots\right) + 8\left(\frac{1}{10^4} + \frac{1}{10^{10}} + \cdots\right) + \\ &\quad + 5\left(\frac{1}{10^5} + \frac{1}{10^{11}} + \cdots\right) + 7\left(\frac{1}{10^6} + \frac{1}{10^{12}} + \cdots\right) + \cdots \\ &= \frac{1}{10}\left(1 + \frac{1}{10^6} + \frac{1}{10^{12}} + \cdots\right) + \frac{4}{10^2}\left(1 + \frac{1}{10^6} + \frac{1}{10^{12}}\right) + \cdots \\ &= \left(1 + \frac{1}{10^6} + \frac{1}{10^{12}} + \cdots\right)\left[\frac{1}{10} + \frac{4}{10^2} + \frac{2}{10^3} + \frac{8}{10^4} + \frac{5}{10^5} + \frac{7}{10^6}\right] \\ &= \frac{1}{1 - \frac{1}{10^6}}\left(\frac{10^5 + 4 \times 10^4 + 2 \times 10^3 + 8 \times 10^2 + 5 \times 10 + 7}{10^6}\right), \end{aligned}$$

Hence,

$$\begin{aligned}
0.142857142857\cdots &= \frac{10^6}{10^6-1}\left(\frac{10^5+5\times 10^4+2\times 10^3+8\times 10^2+5\times 10+7}{10^6}\right),\\
&= \frac{10^5+4\times 10^4+2\times 10^3+8\times 10^2+5\times 10+7}{10^6-1},\\
&= \frac{142857}{999999},\\
&= \frac{142857}{7\times 142857},\\
&= \tfrac{1}{7}.
\end{aligned}$$

In a similar way, all real numbers can be represented as a decimal string. The decimal string may terminate or be periodic (rational numbers), or may be infinite with no repeating pattern (irrational numbers). For example, a real number $y \in [0,1)$, with

$$y = \sum_{n=1}^{\infty} \frac{x_n}{10^n} = 0.x_1 x_2 \cdots$$

where $x_i \in \{0, 1, \cdots, 9\}$. This number does not as yet have a meaning. However, consider the sequence $\{y_N\}$ of rational numbers, where

$$y_N = \sum_{n=1}^{N} \frac{x_n}{10^n}. \qquad (2.1)$$

This is a sequence that is bounded above and monotone increasing. By the **completeness axiom**, the sequence has a limit, hence

$$y = \lim_{N\to\infty} y_N.$$

The completeness axiom is therefore equivalent to the construction of the real numbers: any real number can be obtained as the limit of a rational sequence such as Equation (2.1).

Now that we understand how numbers are represented in base 10 using positional notation, we now examine other bases. Consider for example the string

$$x = 1010110,$$

in base 2. Using positional notation and base 2, we understand x to be the number

$$\begin{aligned}
x &= (1\times 2^6)+(0\times 2^5)+(1\times 2^4)+(0\times 2^3)+(1\times 2^2)+(1\times 2)+(0\times 2^0),\\
&= 64+16+4+2,\\
&= 86, \text{ base } 10.
\end{aligned}$$

Numbers with absolute value less than unity can be represented in a similar way. For example, let

$$x = 0.01101 \text{ base } 2.$$

Using positional notation, this is understood as

$$\begin{aligned} x &= \frac{0}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{0}{2^4} + \frac{1}{2^5}, \\ &= \tfrac{1}{4} + \tfrac{1}{8} + \tfrac{1}{32}, \\ &= \tfrac{8}{32} + \tfrac{4}{32} + \tfrac{1}{32}, \\ &= \tfrac{13}{32}, \\ &= 0.40625 \text{ base } 10. \end{aligned}$$

Two binary strings can be added by 'carrying twos'. For example,

$$\begin{array}{r} 0.01101 \\ +\quad 1.11100 \\ \hline 10.01001 \end{array}$$

Let's check our calculation using base 10:

$$\begin{aligned} x_1 &= 0.01101 = \frac{0}{2} + \frac{1}{4} + \frac{1}{8} + \frac{0}{16} + \frac{1}{32} = \frac{13}{32}, \\ x_2 &= 1.111 = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} = \frac{15}{8} = \frac{60}{32}. \end{aligned}$$

Hence,

$$x_1+x_2 = \frac{73}{32} = 2+\frac{9}{32} = 2+\frac{1}{32}+\frac{8}{32} = 2+\frac{1}{32}+\frac{1}{4} = (1\times 2^1)+(0\times 2)+\frac{1}{2^2}+\frac{1}{2^5} = 10.01001 \text{ base } 2.$$

Because computers (at least notionally) consist of lots of switches that can be on or off, it makes sense to store numbers in binary, as a collection of switches in 'on' or 'off' states can be put into a one-to-one correspondence with a set of binary numbers. Of course, a computer will always contain only a **finite** number of switches, and can therefore only store the following kinds of numbers:

1. Numbers with absolute value less than unity that can be represented as a binary expansion with a finite number of non-zero digits;
2. Integers less than some certain maximum value;
3. Combinations of the above.

An irrational real number (e.g. $\sqrt{2}$) will be represented on a computer by a truncation of the true value. This introduces a potential source of error into numerical calculations – so-called **rounding error**.

2.3 Floating-point representation

Rounding error is the original sin of computational mathematics. A partial atonement for this sin is the idea of **floating-point arithmetic**. A base-10 floating-point number x consists of a fraction F containing the significant figures of the number, and an exponent E:

$$x = F \times 10^E,$$

where

$$\tfrac{1}{10} \leq F < 1.$$

Representing floating-point numbers on a computer comes with two kinds of limitations:

1. The range of the exponent is limited, $E_{\min} \leq E \leq E_{\max}$, where $E_{\min}$ is negative and $E_{\max}$ is positive; both have large absolute values. Calculations leading to exponents $E > E_{\max}$ are said to lead to **overflow**; calculations leading to exponents $E < E_{\min}$ are said to have **underflowed**.

2. The number of digits of the fraction F that can be represented by on and off switches on a computer is finite. This results in rounding error.

The idea of working with rounded floating-point numbers is that the number of significant figures ('precision') with which an arbitrary real number is represented is independent of the magnitude of the number. For example,

$$x_1 = 0.0000001234 = 0.1234 \times 10^{-6}, \qquad x_2 = 0.5323 \times 10^6$$

are both represented to a precision of four significant figures. However, let us add these numbers, keeping only four significant figures:

$$\begin{aligned}
x_1 + x_2 &= 0.0000001234 + 532,300, \\
&= 532,300.0000001234, \\
&= 0.5323000000001234 \times 10^6, \\
&= 0.5323 \times 10^6 \qquad \text{four sig. figs.}, \\
&= x_1.
\end{aligned}$$

Rounding has completely negated the effect of adding x_1 and x_2.

When starting with a real number x with a possibly indefinite decimal expansion, and representing it floating-point form with a finite number of digits in the fraction F, the rounding can be implemented in two ways:

1. Rounding up, e.g.
$$0.12345 = 0.1235, \qquad \text{four sig. figs.},$$
and $0.12344 = 0.1234$ and $0.12346 = 0.1235$, again to four significant figures;

2. 'Chopping', e.g.
$$0.12345 = 0.12344 = 0.12346 = 0.1234, \qquad \text{truncated to four sig. figs.}$$

The choice between these two procedures appears arbitrary. However, consider
$$x = a.aaaaB,$$
which is rounded up to
$$\tilde{x} = a.aaaC,$$
If $B < 5$, then $C = a$, hence
$$x - \tilde{x} = 0.0000B = B \times 10^{-5} < 5 \times 10^{-5}.$$
On the other hand, if $B \geq 5$, then $C = a + 1$ (the digit is incremented by one). In a worst-case scenario, $B = 5$, and
$$\tilde{x} - x = a.aaaC - a.aaaaaB = (C - a) \times 10^{-4} - B \times 10^{-5} = 10^{-4} - 5 \times 10^{-5} = 5 \times 10^{-5}.$$
In either case therefore,
$$|\tilde{x} - x| \leq 5 \times 10^{-5}.$$
Assuming $a \neq 0$, we have $|x| > 1$, hence $1/|x| < 1$, and
$$\boxed{\frac{|\tilde{x} - x|}{|x|} \leq 5 \times 10^{-5} = \tfrac{1}{2} \times 10^{-4}.}$$

More generally, rounding x to N decimal digits gives a relative error

$$\frac{|\tilde{x}-x|}{|x|} \leq \tfrac{1}{2} \times 10^{-N+1}.$$

See if you can show by similar arguments that for chopping, the relative error is twice as large than that for rounding:

$$\frac{|\tilde{\tilde{x}}-x|}{|x|} \leq 10^{-N+1}.$$

A more convenient way of summarizing these results is as follows: Let

$$\tilde{x} = \mathrm{fl}(x)$$

be the result of rounding the real number x using either rounding up or chopping. Define the signed relative error

$$\epsilon = \frac{\mathrm{fl}(x)-x}{x}. \tag{2.2}$$

We know,

$$|\epsilon| \leq \epsilon_N = \begin{cases} \tfrac{1}{2}10^{-N+1}, & \text{rounding up,} \\ 10^{-N+1}, & \text{chopping.} \end{cases} \tag{2.3}$$

Thus, by definition,

$$|\epsilon| \leq \epsilon_N$$

Re-arranging Equation (2.2), we have

$$\mathrm{fl}(x) = x(1+\epsilon), \qquad |\epsilon| \leq \epsilon_N.$$

The value ϵ_N is called **machine epsilon** and depends on the floating-point arithmetic of the machine in question. We can also think of machine epsilon as the largest number x for which the computed value of $1+x$ is 1. It can be computed as follows in Matlab:

```
x=1;
while( 1+x~=1)
  x=x/2;
end
x=2*x;
display(x)
```

However, Matlab will display machine epsilon if you simply enter ‘eps’ at the command prompt.

Common Programming Error:

Thinking that machine epsilon is 'the smallest number (in absolute value) the computer'. This is wrong. Machine epsilon refers to the maximum relative error between a number and its representation on the computer. Equivalently, you can think of it as follows: let x be the smallest number strictly greater 1 representable by the computer. Then $\epsilon_N = x - 1$. If you are still not convinced, we shall see soon when we study the double-precision format that the smallest and largest numbers in absolute value terms are quite distinct from machine epsilon.

2.4 Error accumulation

Most computing standards will have the following property:

$$\mathrm{fl}(a \circ b) = (a \circ b)(1 + \epsilon), \qquad |\epsilon| \leq \epsilon_N, \tag{2.4}$$

where ϵ_N is the machine epsilon and $\circ$ represents an arithmetic operation such as $\times$, $+$, $-$, or $\div$. This is a good property to have: if the error in representing the numbers a and b is small, then the error in representing their sum is also small. Because machine epsilon is very small, the compound error obtained in a long sequence of arithmetic operations (where each component operation has the property (2.4)) is very small. Errors induced by compounding individual errors such as Equation (2.4) are therefore almost always negligible. However, error accumulation can still occur in two other ways:

1. The numbers entered into the computer code lack the precision required for a long calculation, and 'cancellation errors' occur;

2. Certain iterative algorithms contain stable and unstable solutions. The unstable solution is not accessed if the 'initial condition' is zero. However, if the initial condition is ϵ_N, then the unstable solution can grow over time until it swamps the other, desired solution.

These sources of error will become more apparent in the examples in the homework.

2.5 Double precision and other formats

The gold standard for approximating an arbitrary real number in rounded floating-point form

$$x = F \times 2^E \tag{2.5}$$

is the so-called **IEEE double precision**. A double-precision number on a computer can be thought of as a 64 contiguous pieces of memory (64 bits). One bit is reserved for the sign of the number, eleven bits are reserved for the exponent (naturally stored in base 2), and the remaining fifty-two bits are reserved for the significand. Thus, in IEEE double precision, a real number is approximated

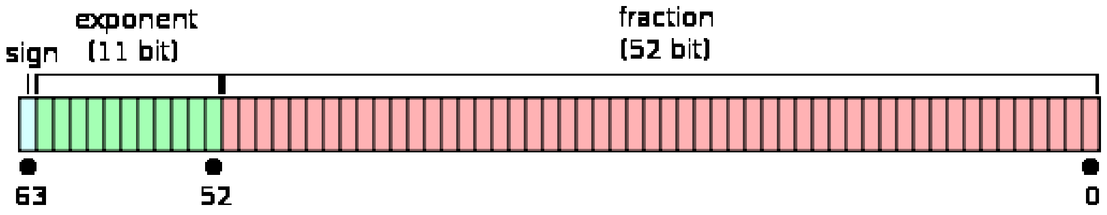


Figure 2.1: 64 contiguous bits in memory make up an IEEE floating-point number, with bits reserved for the sign, the exponent, and the fraction. From http://en.wikipedia.org/wiki/Double-precision_floating-point_format (20/11/2012).

and then stored as follows:

$$x \approx \mathrm{fl}(x) = (-1)^{\mathrm{sign}} \left(1 + \sum_{i=1}^{52} \frac{b_{-i}}{2^i}\right) \times 2^{E_{\mathrm{s}}-1023}.$$

Here, the exponent E_{s} is stored using a contiguous eleven-bit binary string, meaing that E_{s} can in principle range from $E_{\mathrm{s}} = 0$ to $E_{\mathrm{s}} = 2047$. However, $E_{\mathrm{s}} = 0$ is reserved for underflow to zero, and $E_{\mathrm{s}} = 2047$ is reserved for overflow to infinity, meaning that the maximum possible finite exponent is $E_{\mathrm{s}} = 2046$. Accounting for offset, the maximum true exponent is

$$E = E_{\mathrm{s,max}} - 1023 = 2046 - 1023 = 1023.$$

Hence, $x_{\max} \approx 2^{1023}$. Similarly,

$$x_{\min} = 2^{1-1023} = 2^{-1022}.$$

Now, recall the formula

$$\frac{|x - \mathrm{fl}(x)|}{|x|} := \epsilon \le \epsilon_N = \begin{cases} \frac{1}{2} 10^{-N+1}, & \text{rounding up,} \\ 10^{-N+1}, & \text{chopping,} \end{cases}$$

which gave the truncation error in base 10 for truncation after N figures of significance. Going over to base two and chopping, we have

$$\frac{|x - \mathrm{fl}(x)|}{|x|} := \epsilon \le \epsilon_N = 2^{-N+1}.$$

In IEEE double precision, the precision is $N = 52 + 1$ (the extra 1 comes from the digit stored implicitly), hence

$$\epsilon_N = 2^{-53+1} = 2^{-52}.$$

Equivalently, the smallest positive number strictly greather than 1 detectable in this standard is

$$1 + \frac{0}{2} + \frac{0}{2^2} + \cdots + \frac{1}{2^{52}},$$

and again,

$$\epsilon_N = 2^{-52} \approx 2.220456 \times 10^{-16}$$

gives machine precision.

The IEEE standard also supports extensions to the real numbers, including the symbols Inf (which will appear when a code has overflowed), and NaN. The symbol NaN will appear as a code's output if you do something stupid. Examples in Matlab sytanx include the following particularly egregious one:

```
x=0/0;
display(x)
```

Another datatype is the **integer**, which is stored in a contiguous chunk of memory like a double, typically of length 8, 16, 32, or 64 bits. Typically, the integers are defined with respect to an offset (**two's complement**), so that no explicit storage of the sign is required.

Common Programming Error:

Mixing up integers and doubles. For example, suppose in a computer-programming language such as C, that x has been declared to be a double-precision number. Then, assigning x the value 1, i.e.

```
x=1;
```

confuses the compiler, as it now thinks that x is an integer! In order not to confuse the compiler, one would have to write

```
x=1.0;
```

Happily, the distinction between integers and doubles is not enforced in Matlab, and ambiguity about variable types is allowed. However, you should remember this lesson if you do more advanced programming in high-level languages such as C or Fortran.

As hinted at previously, Matlab implements the IEEE double precision standard, albeit implicitly. For example, if you type

```
display(pi)
```

at the command line, you will only see the answer

```
3.1416
```

However, you can rest assured that the built-in working precision of the machine is 53 bits. For example, typing

```
display(eps)
```

yields

```
2.2204e-016
```

Also, typing

```
x=2;
while(x~=Inf)
  x_old=x;
  x=2*x;
end
display(x_old)
```

yields

```
8.9885e+307,
```

the same as $2^{1023} = 8.9885e + 307$.

Chapter 3

Computer architecture and Compilers

Overview

Computer architecture means the relationship between the different components of hardware in a computer. In this chapter, this idea is discussed under the following headings: the memory/processor model, memory organization, processor organization, simple assembly language.

3.1 The memory/processor or von Neumann model

Computer architecture means the relationship between the different components of hardware in a computer. On a very high level of abstraction, many architectures can be described as **von Neumann architectures**. This is a basic design for a computer with two components:

1. An undivided memory that stores both program and data;
2. A processing unit that executes the instructions of the program and operates on the data (CPU).

This design is different from the earliest computers in which the program was hard-wired. It is also very clever, as the line between 'data' and 'program' can become blurred – to our advantage. When we write a program in a given language, we work with a computer that has other, more basic programs installed – including a **text editor** and a **compiler**. The von Neumann architecture enables the computer to treat the code we write in the text editor as data, and the compiler is in this context a 'super-program' that operates on these data and converts our **high-level code** into instructions that can be read by the machine. Having said this, in this module, we understand 'data' to be the collection of numbers to be operated on, and the code is the set of instructions detailing the operations to be performed.

In conventional computers, the machine instructions generated by the compiled version of our code do not communicate directly with the memory. Instead, information about the location of data in the computer memory, and information about where in memory the results of data processing should go, are stored directly in a part of the CPU called the **register**. Rather counter-intuitively, the existence of this 'middle-man' register speeds up execution times for the code. Many computer programs possess **locality of reference**: the same data are often accessed repeatedly. Rather than moving these frequently-used data to and from memory, it is best to store them locally on the CPU, where they can be manipulated at will.

The main statistic that is quoted about CPUs is their Gigahertz rating, implying that the speed of the processor is the main determining factor of a computer's performance. While speed certainly influences performance, memory-related factors are important too. To understand these factors, we need to describe how computer memory is organized.

3.2 Memory organization

Practically, a pure von Neumann architecture is unrealistic because of the so-called *memory wall*. In a modern computer, the CPU performs operations on data on timescales much shorter than the time required to move data from memory to the CPU. To understand why this is the case, we need to study how the CPU and the computer memory communicate.

In essence, the CPU and the computer memory communicate via a load of wires called the **bus**. The **front-side bus** (FSB) or 'North bridge' connects the computer main memory (or 'RAM') directly to the CPU. The bus is typically much slower than the processor, and operates with clock frequencies of $\sim 1\mathrm{GHz}$, a fraction of the CPU clock frequency. A processor can therefore consume many items of data fed from the bus in one clock tick – this is the reason for the memory wall.

The memory wall can be broken up further in two parts. Associated with the movement of data are two limitations: the **bandwidth** and the **latency**. During the execution of a process, the CPU will request data from memory. Stripping out the time required for the actual data to be transferred, the time required to process this request is called latency. Bandwidth refers to the amount of data that can be transferred per unit time. Bandwidth is measured in $\mathrm{bytes/second}$, where a byte (to be discussed below) is a unit of data. In this way, the total time required to for the CPU to request and receive n bytes from memory is

$$T(n) = \alpha + \beta n,$$

where α is the latency and β is the inverse of the bandwidth ($\mathrm{second/byte}$). Thus, even with infinite bandwidth ($\beta = 0$), the time required for this process to be fulfilled is non-zero.

Typically, if the chunk of memory of interest physically lies far away from the CPU, then the latency is high and the bandwidth is low. It is for this reason that a computer architecture tries to maximize the amount of memory near the CPU as possible. For that reason, a second chunk of memory close the CPU is introduced, called the **cache**. This is shown schematically in Figure 3.1. Data needed in

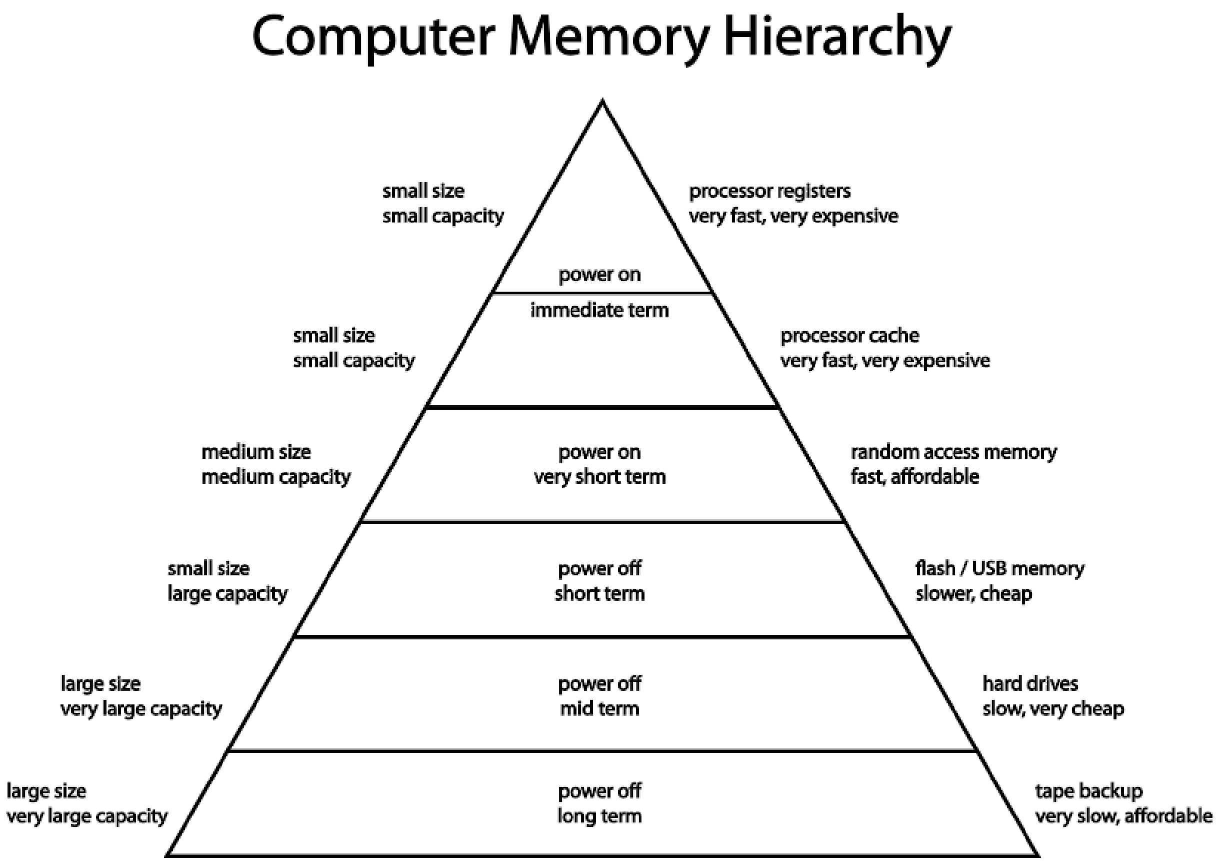


Figure 3.1: The different levels of memory shown in a hierarchy

some operation gets copied into the cache on its way to the processor. If, some instructions later, a data item is needed again, it is searched for in the cache. If it is not found there, it is loaded from the main memory. Finding data in cache is called a **cache hit**, and not finding it is called a **cache miss**. Again, the cache is a part of the computer's memory that is located *on the die*, that is, on the processor chip. Because this part of the memory is close the CPU, it is relatively quick to transfer data to and from the CPU and the cache. For the same reason, the cache is limited in size. Typically, during the execution of a programme, data will be brought from slower parts of the computer's memory to the cache, where it is moved on and off the register, where in turn, operations are performed on the data. There is a sharp distinction between the register and the cache. The instructions in machine language that have been generated by our compiled code are instructions to the CPU and hence, to the register. It is therefore possible in some circumstances to control movement of data on and off the register. On the other hand, the move from the main memory to the cache is done purely by the hardware, and is outside of direct programmer control.

3.3 The rest of the memory

The rest of the memory is referred to as ‘RAM’, and is neither built into the CPU (like the registers), nor collocated with the CPU (like the cache). It is therefore relatively slow but has the redeeming feature that it is large. The most-commonly known feature of RAM is that the data it contains are removed when the computer powers off. This is why you must save your work to the hard drive!

RAM itself is broken up into two parts – the **stack** and the **heap**.

Stacks are regions of memory where data is added or removed on a last-in-first-out basis. The stack really does resemble a stack of plates. You can only take a plate on or off the top of a stack – this is also true of data stored in the stack. Another silly analogy is to imagine a series of postboxes attached one on top of the other to a vertical pole. Initially, all the postboxes are empty. Then, the bottommost postbox is filled and a postit note is placed on it, indicating that the location of the next available postbox. As letters are put into and removed from postboxes, the postit note moves up and down the stack of postboxes accordingly. It is therefore very simple to know how many postboxes are full and how many are empty – a single label suffices. The system for addressing memory slots in the stack is equally simple and for that reason, accessing the stack is faster than accessing other kinds of memory.

On the other hand, there is the **heap**, which is a region of memory where data can be added or removed at will. The system for addressing memory slots in the heap is therefore much more detailed, and accessing the heap is therefore much slower than accessing the stack. However, the size of the stack is fixed at runtime and is usually quite small. Many codes require lots of memory. Trying to fit lots of data into the relatively small amount of stack that exists can lead to **stack overflow** and **segmentation faults**. Stack overflow is a specific error where the exectuting program requests more stack resources than those that exist; segmentation faults are generic errors that occur when a code tries to access addresses in memory that either do not exist, or are not available to the code. So ubiquitous and terrifying are these errors to computer codes a popular web forum for coders and computer scientists is called http://stackoverflow.com/.

If you ever do beginner’s coding in C or Fortran remember the following lesson:

Common Programming Error:

Never allocate arrays on the stack (Possibly Fatal)!

In this module, these issues will never arise; however, this is a salutary lesson, and one not often referred to in beginner’s courses on real coding!

All of the different levels of memory and their dependencies are summarized in the diagram at the end of this chapter (Figure 3.2).

3.4 Multicore architectures

If you open the task manager on a modern machine running Windows, the chances are you will see two panels by first going to 'performance' and then 'CPU Usage History' . It would appear that the machine has two CPUs. In fact, modern computers contain **multiple cores**. We still consider the machine to have a single CPU, but two smaller processing units (or cores) are placed on the same chip. The two cores share some cache ('L2 cache'), while some other cache is private to each core ('L1 cache'). This enables computer to break up a computational task into two parts, work on each task separately, via the private cache, and communicate necessary shared data via the shared cache. This architecture therefore facilitates **parallel computing**, thereby speeding up computation times. High-level programs such as MATLAB take advantage of multiple-core computing without any direction from the user. On the other hand, lower-level programming standards (e.g. C, Fortran) require explicit direction from the user in order to implement multiple-core processing. This is done using the OpenMP standard.

Unfortunately, the idea of having several cores on a single chip makes the description of this architecture ambiguous. We reserve the word **processor** for the entire chip, which will consist of multiple sub-units called **cores**. Sometimes the cores are referred to as **threads** and this kind of computing is called **multi-threaded**.

3.5 Compilers

As mentioned in Section 3.1, a standard procedure for writing code is the following:

1. Write the code in a **high-level** computer language such as C or Fortran. You will do this in a text editor. Computer code on this level has a definite syntax that is very similar to ordinary English.

2. Convert this high-level code to **machine-readable** code using a **compiler**. You can think of this as a translator that takes the high-level code (readable to us, and similar in its syntax to English) into lots of gobbledegook that only the computer can understand.

3. Compilation takes in a text file and outputs a machine-readable **executable** file. The executable can then be run from the **command line**.

MATLAB sits one level higher than a high-level computer language, with a friendly syntax and all sorts of clever procedures for allocating memory so that we don't need to worry about technical issues. It also has a user-friendly interface so that our high-level Matlab files can be run and the output interpreted and plotted in a user-friendly fashion. Incidently, Matlab is written in C, so it as though two translations happen before the computer executes our code: Matlab$\rightarrow$ C $\rightarrow$ (Machine-readable code).

In this course, issues of precision, truncation error, and computer architecture are moot. Now that we have tentatively (and metaphorically) opened the lid of our computer and seen its architecture, we will close it firmly, learn Matlab, and compute things. That said, these questions are important a number of reasons:

1. Learning stuff is always good!
2. We should never treat something as a 'black box' to be intereacted with only by mindlessly pressing a few buttons. Knowledge is good (point 1 again).
3. Sometimes, things go wrong with our codes (e.g. truncation error). Then, we need to understand properly how numbers are represented on a computer.
4. Suppose that our calculations become large (requiring long runtimes and large amounts of memory). Then, knowledge of the computer's architecture helps us to understand the limitations of the calculations, and extend those limits (e.g. virtual memory, multi-threading / shared memory, distributed memory). These last topics would be studied typically in an MSc in High-Performace Computing.

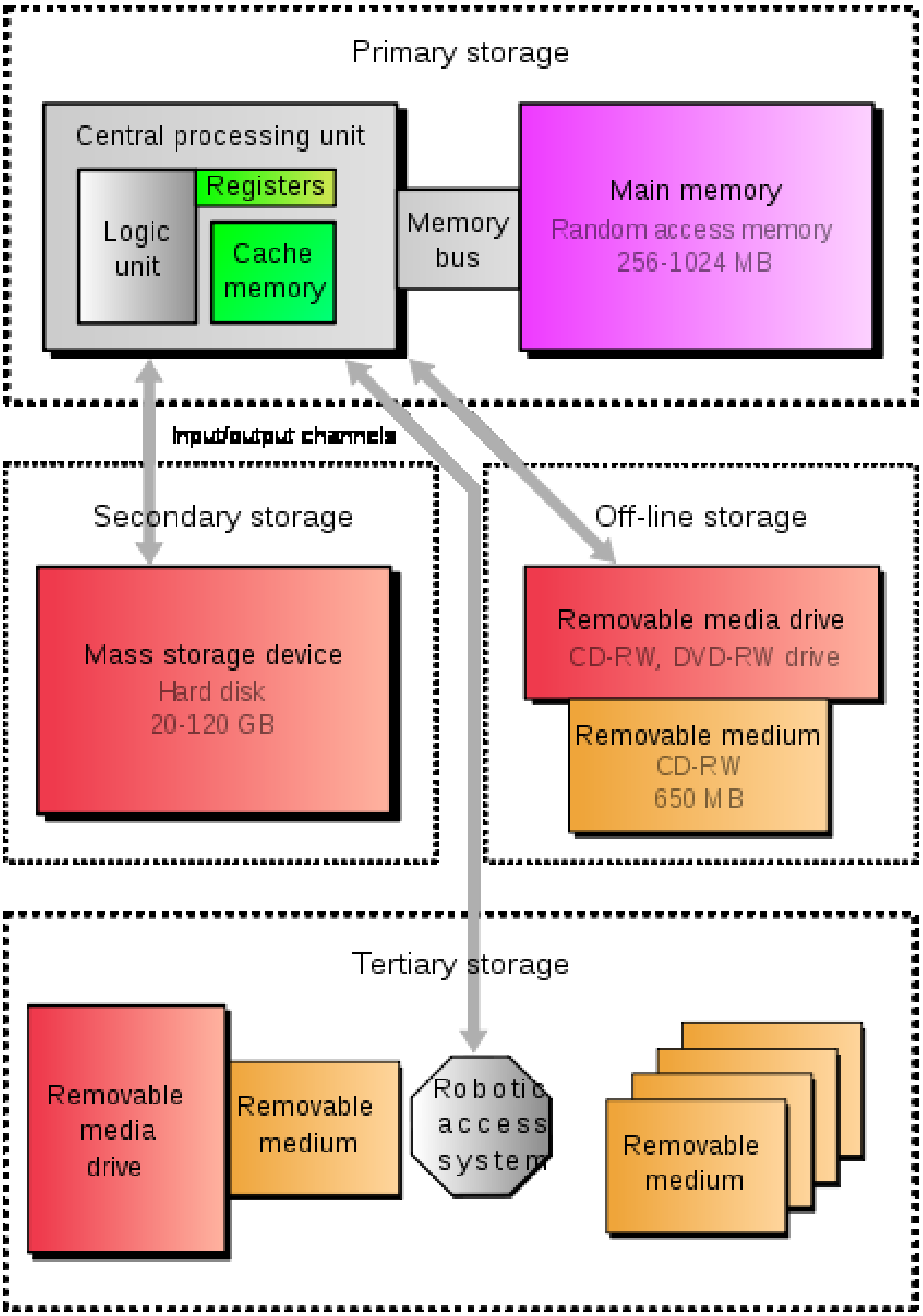


Figure 3.2: (From Wikipedia) Computer architecture showing the interaction between the different levels of memory.

Chapter 4

Our very first Matlab function

Open the Matlab text editor and type the following:

```
function x=addnumbers(a,b)
x=a+b;
end
```

Save this as a file called "addnumbers.m" We have thus created a Matlab **function** "addnumbers" with filename "addnumbers.m". We call a, b, and x **variables**. These are placeholders for a real number. There are rich analogies between computer syntax and mathematical syntax. Given a function like $f(x) = 2x^2 + x + 1$, $f(x)$ and x are placeholders for real numbers, and the real number $f(x)$ is got by setting x equal to a definite value and then evaluating the function. Again, just like in mathematical functions, we have the notion of **inputs** and **outputs**:

1. The inputs to the Matlab function are a and b, which can be any real numbers.
2. The output is $x = a + b$.

Common Matlab Programming Error:

- Not giving the Matlab function and its filename the same name.
- Matlab is CaSE SensItiVE: a and A are not the same variable. ['Little-a' and 'big-a' are not the same variable.]

Now, at the command line, type

```
x=addnumbers(1,2);
display(x)
```

The result should be $x = 3$. You could get the same result by typing

```
x=addnumbers(1,2)
```

Common Matlab Programming Error:

Not using the semicolon to suppress output. This is not fatal, but can lead to lots of unnecessary numbers being displayed on the GUI.

Matlab functions can have more than one output. For example, consider the following:

```
function [x,y]=add_and_multiply(a,b)
x=a+b;
y=a*b;
end
```

After saving this function, one would type at the command line:

```
[x,y]=add_and_multiply(1,2)
```

Chapter 5

Vectors, Arrays, and Loops in Matlab

Overview

At its heart Matlab is nothing more than a glorified Linear Algebra package. It is a giant calculator for doing linear-algebra calculations very efficiently. A main aim of this module is therefore to understand Matlab's syntax for handling vectors and matrices (and more generally, arrays).

5.1 Vectors and For Loops

Supposing we have an ordinary three-dimensional vector

$$\boldsymbol{v} = (1, 2, 4)$$

This can be stored in Matlab (for example, in RAM, on the command line) by typing

```
v=[1,2,3];
```

We can check that the individual components of the vector have been stored properly by typing

```
display(v(1))
display(v(2))
display(v(3))
```

Thus, $v(i)$ is the i^{th} component of the vector, in the Matlab syntax. We call i the index. Here, obviously, $i = 1, 2, 3$.

The for loop

Accessing the different components of a vector is straightforward for a three-dimensional vector. However, supposing we had the following vector:

```
v=rand(100,1);
```

which is a 100-wide row vector with entries that are random numbers between 0 and 1[1]. We might like to print all of the elements to the screen. Typing

```
display(v(1))
display(v(2))
display(v(3))
```

&c &c all the way down to the 100^{th} index would be tiresome and very silly. Happily, we can tell Matlab to cycle through each of the elements in the vector in a sequential manner, and print the elements to the screen as Matlab cycles through the vector. This is done with a **for loop**:

```
for i=1:100
  display(v(i))
end
```

Granted, the same result could be accomplished by typing

```
v
```

but that would be less instructive.

[1]The notion of random numbers on a computer are treated in Chapter 25.

The mean of the components

Suppose now that we want to compute the mean of the components of the vector. Mathematically, we have

$$\boldsymbol{v} = (v_1, \cdots, v_{100}), \qquad \overline{v} := \frac{1}{100}\sum_{i=1}^{100} v_i.$$

This can be accomplished with a for loop as follows:

```
sum_val=0;
for i=1:100
  sum_val=sum_val+v(i);
end
sum_val=sum_val/100;
display(sum_val)
```

I can't really explain this to you; you will just have to go away and look at it, and play with the associated Matlab function. After worrying about this for long enough, I promise it will make sense.

Common Matlab Programming Error:

Not initializing `sum_val` to be zero (Fatal).

Moving on, a keynote of this module is the following principle:

Good Programming Practice:

Operations on vectors can be performed component-wise or equivalently, using inbuilt vector functions.

In other words, for every for loop that we construct, there is a specialized Matlab command that does the same thing. For example, typing

```
sum_val=sum(v)/100
```

will also give the mean of the random vector; here 'sum' is the built-in Matlab function.

Exercise 5.1 *Let*

$$v\texttt{=rand(1,200)}, \qquad w\texttt{=rand(1,200)}$$

be two **distinct** *random vectors. Compute the dot product of* $\boldsymbol{v}$ *and* $\boldsymbol{w}$*,*

$$\boldsymbol{v}\cdot\boldsymbol{w} = \sum_{i=1}^{200} v_i w_i$$

(i) using a for loop; (ii) using a built-in function to be found by looking at the Matlab Help pages.

The dot-star operation

Following on from this exercise, we introduce a very useful operation in matlab called **dot-star**. This is pointwise multiplication. Given vectors

$$\boldsymbol{v} = (v_1, \cdots, v_n), \qquad \boldsymbol{w} = (w_1, \cdots, w_n),$$

a new vector $\boldsymbol{v}\cdot *\boldsymbol{w}$ is defined such that

$$\boldsymbol{v}\cdot *\boldsymbol{w} = (v_1 w_1, \cdots, v_n w_n).$$

Thus, an alternative way of doing Exercise 5.1 is to type

```
newvec=v.*w;
dotprod=sum(newvec);
```

Common Matlab Programming Error:

Typing $v * w$ when $v \cdot *w$ is meant. The ordinary $*$ operation in Matlab means the multiplication of two scalars, or two matrices (see below).

5.2 Nested for-loops and matrices

Let $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times p}$ be matrices. We can take the product of these matrices: the matrix AB has ij^{th} component

$$(AB)_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj}.$$

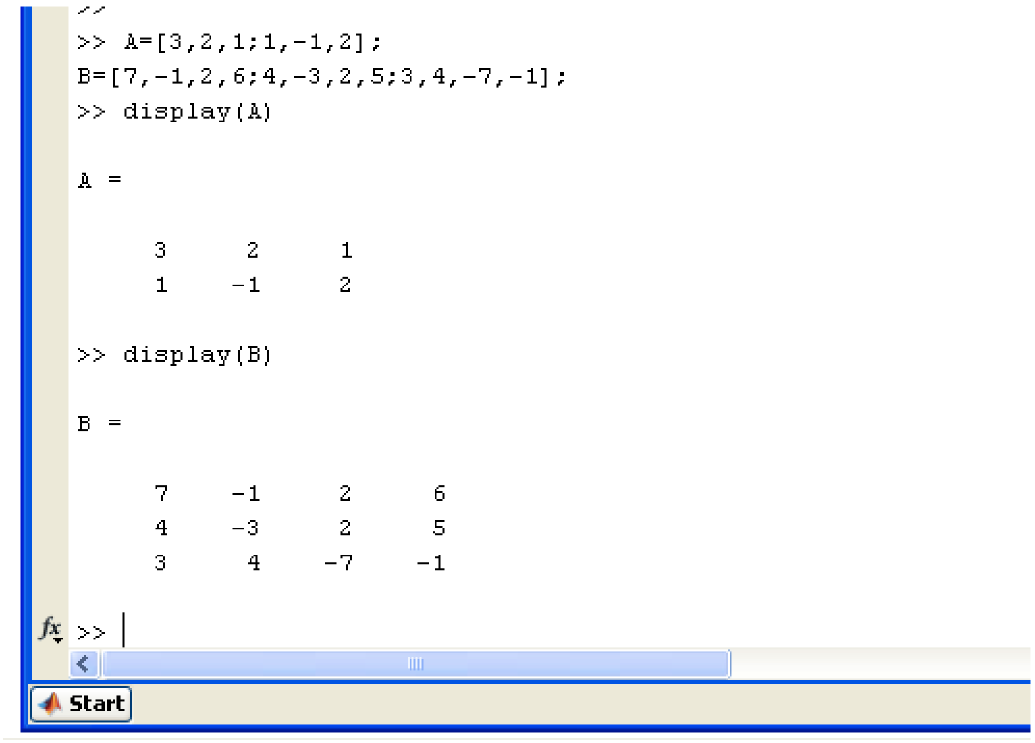
Thus, the ij^{th} component is obtained by taking the i^{th} row of A and dotting it with the j^{th} column of B. For that reason, to do matrix multiplication, the number of elements in a column of A should be the same as the number of elements in a row of B. This can be remembered in a mnemonic:

$$\text{(Matrix product) } (m\times n)(n\times p) = \text{(new matrix) } (m\times p).$$

It is as if we do a 'cross multiplication' whereby 'the n in the middle cancels'. Using dot products, we can now multiply two matrices, as in the following example:

```
A=[3,2,1;1,-1,2];
B=[7,-1,2,6;4,-3,2,5;3,4,-7,-1];
```

It might be nice to visualize these matrices before we go any further:



```
>> A=[3,2,1;1,-1,2];
B=[7,-1,2,6;4,-3,2,5;3,4,-7,-1];
>> display(A)

A =

     3     2     1
     1    -1     2

>> display(B)

B =

     7    -1     2     6
     4    -3     2     5
     3     4    -7    -1

>> 
```

The matrix A is a 2×3 matrix; B is 3×4. Their matrix product AB will be 2×4. We now **allocate** a matrix to hold the result of our calculation:

```
ABprod=zeros(2,4);
```

Good Programming Practice:

Always initialize or 'allocate' any arrays which are to be accessed using 'for' loops. In some cases, this can speed up the code's execution times by factors of 10 or 100.

Now, we take the i^{th} row of A and we dot it with the j^{th} row of B. But we have now hit a problem! There are two labels (or 'indices') to 'loop' over – and we are only familiar with 'for loops' over one index. The answer is a **nested for loop**:

```
for i=1:2
  for j=1:4
    tempa=A(i,:);
    tempb=B(:,j);
    ABprod(i,j)=dot(tempa,tempb);
  end
end
```

Now, by now, you should be starting to realise that a main goal of this course is to open up the 'black box' made up by Matlab's built-in functions. For that reason, we can check the results of our calculation with Matlab's own built-in method for multiplying matrices:

```
display(ABprod)
display(A*B)
```

Chapter 6

Operations using for-loops and their built-in Matlab analogues

Exercise 6.1 *Write a Matlab function to do the following tasks. If possible, verify your answer using the appropriate built-in functions which can be found in the Matlab 'help' documents.*

1. Compute the factorial of a non-negative integer.
2. Compute the cross product of two three-dimensional vectors.
3. Compute the square of a $n \times n$ matrix. The input must be a square matrix – A, say. The size of A can be obtained from the command

   ```
   [nx,ny]=size(A);
   ```

   Because the matrix is square, nx and ny should be the same. Later on we will write code to check if conditions like this one are true.
4. Using the formula

   $$\tfrac{1}{90}\pi^4 = \sum_{n=1}^{\infty} \frac{1}{n^4}, \tag{6.1}$$

   compute π valid to 10 significant figures.

   Hints:

   - The apparent (i.e. displayed) precision of Matlab can be lengthened by first of all typing

     ```
     format long
     ```

at the Matlab command line, before the function is executed.

- In this exercise, you should write a function that takes in N_{approx} – a finite truncation order of the sum (6.1). It should return a value π_{approx}. You should experiment by executing the function for different (increasing) values of N_{approx} until there is no change in the first 10 digits of π_{approx}.
- You should write two versions of the function. The first version will use a four loop; the second will use only built-in Matlab functions $.*$, $./$, and $\mathrm{sum}()$. A vector $(1, 2, \cdots, N)$ can be defined in Matlab with the command

  ```
  vecN=1:1:N;
  ```

  Here, 1 is the starting value of the vector, N is the final value, and the 1 sandwiched between the colons is the increment.

Chapter 7

While loops, logical operations, precedence, subfunctions

Overview

We introduce some additional operations in Matlab that will be indispensable throughout this module.

7.1 The ‘while’ loop

We have seen how the ‘for’ loop provides a means of accessing the elements of a vector or an array in a sequential fashion, e.g.

```
v=1:1:10;
for i=1:length(v)
  temp_val=v(i);
  display(temp_val)
end
```

The ‘for’ loop passes the counter `i` through the loop. During each pass through the loop, the counter is **incremented** by one. The passes continue through the loop provided the statement

$$i \leq 10$$

is true. When this statement becomes false, the passes through the loop stop. Thus, a sequence of logical operations (true/false) is carred out automatically, until certains statements become false.

Another way of doing this is with a **while** loop, as follows:

```
v=1:1:10;
i=1;
while(i<=10)
  temp_val=v(i);
  display(temp_val)
  i=i+1;
end
```

Indeed, this is completely equivalent to the for loop. The counter starts at $i = 1$. The counter enters into a loop, an operation is performed, and the counter is incremented by one. The passes through the loop continue until a condition becomes false.

Since these methods are completely equivalent, one might ask why we bother with the 'while' loop. The answer is that the logical condition can be much more general than

```
imax=...
while(i<=imax)
```

For example, suppose that in Chapter 6 we called our function to approximate π 'sum_nfour.m'. This code would contain the following elements:

```
function [pi_approx]=sum_nfour(N)

pi_approx=...

end
```

sample_matlab_codes/sum_nfour_missing_info.m

We might like to continue implementing this code, continuously increasing the truncation order N until a decent approximation of π is obtained. We would do the following **at the command line:**

```
format long
tol=1e-6;
N=10;
pi_approx=sum_nfour(N);
while(abs(pi-pi_approx)>tol)
  N=N+10;
  pi_approx=sum_nfour(N);
end
display(pi_approx);
display(N)
```

The ‘while’ loop is therefore more general than a ‘for’ loop. With this extra freedom comes a requirement for extra caution:

Common Programming Error:

- Forgetting to initialize the counter in the ‘while loop’
- Forgetting to increment the counter in the ‘while loop’
- Performing an operation on the incremented counter $(i+1)$ instead of using i.

7.2 Logical operations

We have already mentioned that the counter in ‘for’ and ‘while loops’ are incremented until some logical condition becomes false. This suggests that Matlab has a way of checking for the truth or falseness. This is indeed correct. Such checks are often encountered in ‘if’ statements.

‘If’ statements

Suppose that in Chapter 6 had a Matlab code to compute A^2, where A is a square matrix. This code would contain the following elements:

```
function Asq=square_A(A)

[nx,ny]=size(A);

...

end
```

sample_matlab_codes/square_A_missing_info.m

If $n_x \neq n_y$ there is not really much point in going any further with this calculation, as it will return nonsense. It might be good to have in the code a check to see if $n_x = n_y$, and to know what to do in case $n_x \neq n_y$. The following flowchart indicates what we need:

- If $n_x = n_y$ we need to get on with the calculation!
- If $n_x \neq n_y$ we should exit the code.

This can be implemented in Matlab with an ‘if-else statement’:

```
 1 function Asq=square_A_missing_info1(A)
 2 
 3 [nx,ny]=size(A);
 4 
 5 if(nx==ny)
 6     % The code to square A goes here.
 7     ...
 8 else
 9     % We should exit the code and return a value.
10     Asq=0*A;
11     display('Error: A is not a square matrix')
12     display('Returning A^2=0 and exiting code')
13     return
14 end
15 
16 end
```

sample_matlab_codes/square_A_missing_info1.m

Some notes:

- The condtion $n_x = n_y$ is checked in Line 5, with the piece of code `if(nx==ny)`. The double equals sign is not a typo: this is a **logical equals** sign, which is an operation to check the truth of the statement $n_x = n_y$.

  On the other hand, the piece of code `nx=ny` is called an **assignment equals** sign: it is an operation whereby the variable nx is assigned the value ny.

Common Matlab Programming Error:

Using an assignment equals sign in a logical check.

- On line 8, Matlab is instructed what to do if A is not a square matrix. Because we have written a function, we have in a sense painted ourselves into a corner: we must return some output to the command line, even if a correct calculation is impossible. We elect to return a zero matrix of size $n_x \times n_y$, and alert the user using the warnings on lines 11 and 12 that a mistake has been made.

As a further example of an ‘if-else statement’, consider a homemade Matlab function to compute the absolute value of a number:

$$|x| = \begin{cases} +x, & \text{if } x \geq 0, \\ -x, & \text{if } x \leq 0. \end{cases}$$

This is implemented as follows:

```
function [abs_x]=abs_x_homemade(x)

if(x>=0)
    abs_x=x;
else
    abs_x=-x;
end

end
```

sample_matlab_codes/abs_x_homemade.m

Of course, as with many other things in Matlab, there is a built-in function for computing absolute values:

```
abs_x=abs(x);
```

If built-in functions exist, they should always be preferred over their home-made alternatives: armies of Ph.D. computational scientists are paid lots of money by Matlab to devise clever algorithms; unfortunately, we are rarely likely to beat them at their own game.

Common Matlab Programming Error:

- Using a homemade Matlab function instead of the built-in alternative.
- Calling a homemade function by a name reserved for a built-in function.

Other logical operations are possible. For example, it is possible to check a condition without having an alternative ('if without the else'). Further possibilities:

- A series of independent 'if' statements, e.g.

```
if(i<2)
...
end

if(i<10)
...
end
```

- A series of **dependent** ‘if’ statements, e.g.

```
if(i<2)
...
elseif(i<10)
...
elseif(i<20)
...
else
...
end
```

Sequences of independent ‘if’ statements can lead to errors. For example, consider the following piece of code:

```
function x=sample_if_statements1(i)

if(i<10)
  x=5;
end

if(i<2)
    x=2;
end

end
```

sample_matlab_codes/sample_if_statements1.m

This is an attempt at writing a code to implement the function

$$f(i) = \begin{cases} 2, & \text{if } i < 2, \\ 5, & \text{if } 2 < i < 10. \end{cases}$$

Here, the list of alternatives for the input variable i is not exhaustive (e.g. what about $i \geq 10$). An input value $i \geq 10$ gives rise to a situation where none of the ‘if’ statements are true, and the output variable x is therefore not assigned a value. This causes the code to crash.

Common Matlab Programming Error:

Writing a list of ‘if’ statements that is not exhaustive, causing an output variable to be unassigned.

A better idea is the following:

```
function x=sample_if_statements2(i)

if(i<2)
    x=2;
elseif(i<10)
    x=5;
else
    display('input variable out of range')
    display('returning x=-1')
    x=-1;
end

end
```

sample_matlab_codes/sample_if_statements2.m

Now, the list is exhaustive! If a value $i \geq 10$ is entered (out of range of the funcion $f(i)$), an error message is printed and the value $x = -1$ is returned, thereby avoiding errors where output variables are unassigned.

Logical 'And' and 'Or' operations

Suppose a complex piece of code we are writing relies on a function $f(x)$ being positive at **two** values. How could we check that? The answer is using a **logical 'and' operation**. For example, consider

```
function check=check_sign_f1()

% We are going to check the sign of f(a) and f(b), for
%
% f(x) = sin(x)+x*cos(x)+exp(x)/(1+x^2).

a=1;
b=2;

fa=sin(a)+a*cos(a)+exp(a)/(1+a^2);
fb=sin(b)+b*cos(b)+exp(b)/(1+b^2);

if( (fa>0) && (fb>0)  )
    check=1;
    display('both function evaluations have positive sign')
else
    check=0;
```

```
end

end
```

sample_matlab_codes/check_sign_f1.m

On the other hand, suppose that our code relies on $f(x)$ being positive at $x = a$ OR $x = b$ (or both). We check this using a **logical 'or' operation**:

```
function check=check_sign_f2()

% We are going to check the sign of f(a) and f(b), for
%
% f(x) = sin(x)+x*cos(x)+exp(x)/(1+x^2).

a=1;
b=2;

fa=sin(a)+a*cos(a)+exp(a)/(1+a^2);
fb=sin(b)+b*cos(b)+exp(b)/(1+b^2);

if( (fa>0) || (fb>0)  )
    check=1;
    display('at least one of the function evaluations has positive sign')
else
    check=0;
end

end
```

sample_matlab_codes/check_sign_f2.m

Logical negation

Often it is useful to check if a variable x is NOT equal to some singular value. For example, suppose we want to compute $f(x) = \sin(x)/x$. Obviously, $\sin(0)/0$ is not defined, but by l'Hôpital's rule, we know that it is sensible to define $f(0) = 1$. We would write the following piece of code:

```
if(x==0)
  fx=1;
else
  fx=sin(x)/x;
end
```

However, the same operation can be achieved using a logical negation:

- If $x \neq 0$, then $f(x) = \sin(x)/x$;
- Otherwise, we have $x = 0$ and we set $f(x) = 1$.

This is implemented in Matlab as follows:

```
if(x~=0)
  fx=sin(x)/x;
else
  fx=1;
end
```

‘Isnan’ and ‘Isinf’ statements

Finally, there are other checks that one can perform. We might like to see if a varible has overflowed to become ‘numerical infinity’:

```
x=1/0;
isinf(x)
```

Typing `isinf(x)` in this instance returns the value 1. In logical operations, ‘1’ corresponds to ‘true; and ‘0’ to ‘false’. Thus, when `isinf(x)`$=1$, we know that x has overflowed to become numerical infinity.

Similarly, we can check to see if a number has been badly defined to become ‘Not a number’:

```
x=0/0;
isnan(x)
```

Typing `isnan(x)` returns the value 1, meaning that it is **true** that x is not a (double precision) number. On the other hand, typing

```
y=1;
isnan(y)
```

returns 0, meaning that y is well-defined as a double-precision number.

7.3 Precedence

As in ordinary arithmetic, the precedence of operations (i.e. which comes first in a composition of operations) is BOMDAS. Sensibly, compositions of operations that ordinarily have the same level or precedence are performed starting with the leftmost operation and then reading to the right. However, Matlab admits more operations than primary-school arithmetic, so the list is longer. The following list is not exhaustive, but includes all of the operations you will encounter in this module:

1. Brackets ()
2. Matrix transpose (.'), pointwise power (.^), Matrix complex-conjugate-transpose (') and scalar complex conjugate ('), matrix power (^)
3. Unary plus (+), unary minus ($-$), logical negation ($\sim$)

   Unary operators (operators involving only one argument) do not really have an independent existence in Matlab; here $+A$ just means A, and $-A$ means $(-1) \times A$, where A is an array.
4. Pointwise operations: multiplication (.$*$), right division (./), left division (.\); Matrix operations: matrix multiplication ($*$), matrix right division (/), matrix left division (\)
5. Addition (+), subtraction ($-$)
6. Logical operators: less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=), equal to (==), not equal to (=)
7. Short-circuit AND (&&)
8. Short-circuit OR (||)

   Short-circuit AND and OR means that the second argument of the operation is not evaluated unless it is needed.

7.4 Subfunctions

It is quite common in Matlab to write a function in Matlab (a '.m' file) and to find that within that file, you need to call other functions. This idea of a 'function within a function' can be easily accommodated in Matlab and is called 'nesting'.

We re-visit the example in Section 7.2 (check_sign_f1.m), with a small twist: we check the sign of the (mathematical) function

$$f(x) = \sin x + x \cos x + \frac{\mathrm{e}^x}{k_0^2 + x^2},$$

at locations $x = a$ and $x = b$. Here k_0 is a user-defined constant that entered at the command line when the (Matlab) function is called. Instead of having two near-identical function evaluations at $x = a$ and $x = b$, we make a one-off definition of $f(x)$ and reuse it as follows:

```
function check=check_sign_f3(k0)

% We are going to check the sign of f(a) and f(b), for
%
% f(x) = sin(x)+x*cos(x)+exp(x)/(k0^2+x^2).

a=1;
b=2;

fa=evalf(a);
fb=evalf(b);

if( (fa>0) && (fb>0)  )
    check=1;
    display('both function evaluations have positive sign')
else
    check=0;
end

% ************************************************************************
% Definition of f(x) here.

    function y=evalf(x)
        y=sin(x)+x*cos(x)+exp(x)/(k0^2+x^2);
    end

end
```

sample_matlab_codes/check_sign_f3.m

The advantage of this is approach is economy. While this economy is not very clear here, one can imagine that such ‘recycling’ is extremely important when (say) 100 sequential function evaluations are required.

Writing subfunctions has its pitfalls. In the example above (check_sign_f3.m) the subfunction where $f(x)$ is defined is **nested** – it appears between the beginning and the end of the main function. It is also possible to have a completely independent subfunction:

```
function check=check_sign_f4(k0)

% We are going to check the sign of f(a) and f(b), for
%
% f(x) = sin(x)+x*cos(x)+exp(x)/(k0^2+x^2).

a=1;
b=2;

fa=evalf(a,k0);
fb=evalf(b,k0);

if( (fa>0) && (fb>0)  )
    check=1;
    display('both function evaluations have positive sign')
else
    check=0;
end

end

% ************************************************************************

function y=evalf(x,k0_loc)
    y=sin(x)+x*cos(x)+exp(x)/(k0_loc^2+x^2);
end
```

sample_matlab_codes/check_sign_f4.m

However, in this case, none of the variables defined in the main part of the code is defined in the subfunction. A real programmer would say that the variables in the main function are **limited in scope**, or are only **locally defined**. For that reason, we pass two values to the subfunction $f(x)$ – the value of the variable x, and the value of the parameter k. For the avoidance of ambiguity, we give the parameter k a new variable name in the subfunction, calling it `k_loc` (for 'local', as it is locally defined in the subfunction).

Common Matlab Programming Error:

Hoping that local variables will be defined in an indpendent (non-nested) subfunction.

There is another way around the issue of passing variables limited in scope to independent (non-nested) subfunctions. One can declare a variable to be **globally defined**. However, to the uninitiated, these can be very dangerous, and are not discussed further in this module.

Chapter 8

Plotting in Matlab

Overview

We learn how to make simple one-dimensional curve plots in Matlab. We also learn how to prettify these plots in order to create production-level graphics.

8.1 The idea

As we have mentioned before, at its heart, Matlab is a tool for maniuplating vectors and matrices. For that reason, the way in which we plot functions is based on the maniuplation of vectors.

For example, suppose we wish to plot the function

$$f(x) = \sin x + x\cos x + \frac{\mathrm{e}^x}{1+x^2}$$

in the range $[0, 6]$.

We would create a vector of x-locations, spaced apart by a small distance:

```
x=0:0.01:6;
```

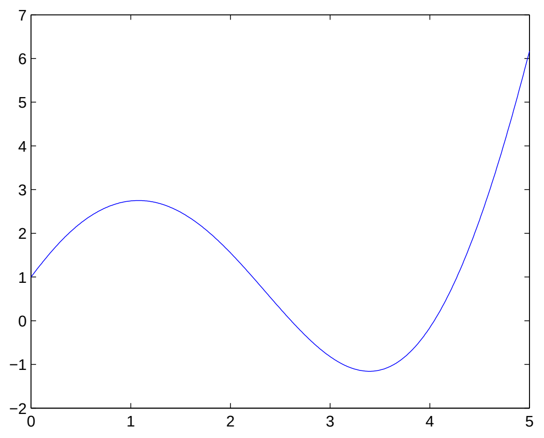
We would then create a second vector of points, corresponding to $f(x)$:

```
fx=sin(x)+x.*cos(x)+exp(x)./(1+x.^2)
```

(note the '.*' operation here). We would then plot the result as follows:

```
plot(x,fx)
```

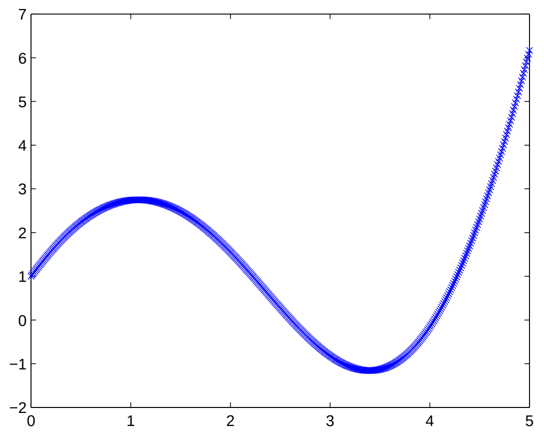
The result looks like the following figure:



Of course, we have not plotted a continuous curve, rather we have plotted the value of $f(x)$ at the discrete x-locations $x = 0, 0.01, 0.02, \cdots$. One way to see this explicitly is to put a big 'X' at each of these discrete locations:

```
plot(x,fx,'-x')
```

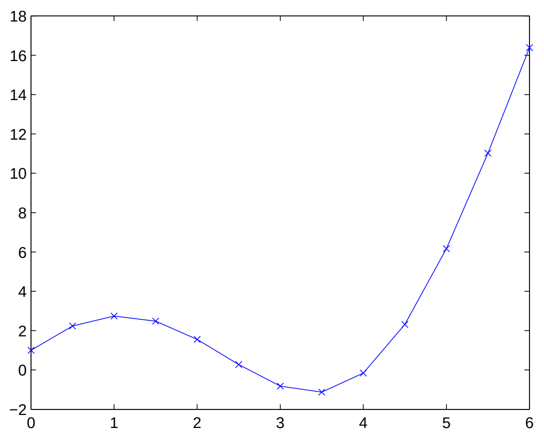
Clearly, there are lots of these dots, and our grid `x=0:0.01:6` is fine enough to give a good description of the continuous curve $(x, f(x))$.



To see the effects of having too coarse a grid, we de-refine the x-grid as follows:

```
x=0:0.1:6;
plot(x,fx,'-x')
```

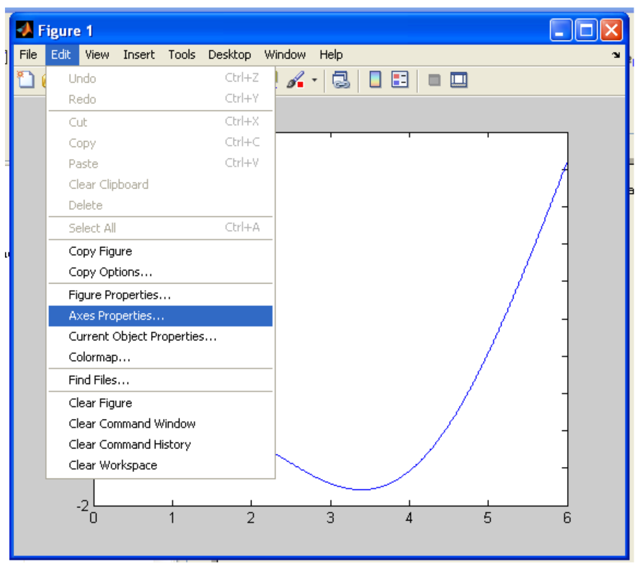
The result is terrible!



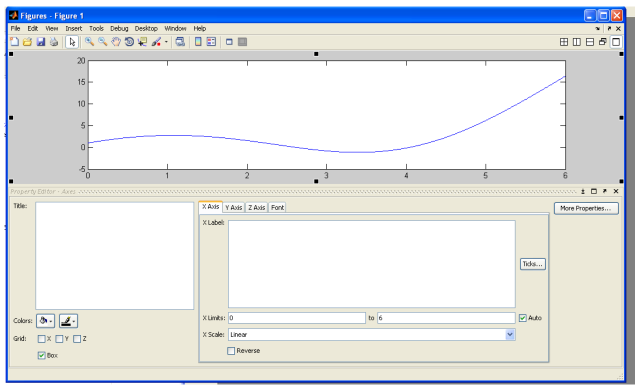
Clearly, the grid chosen must match the amount of variation in the function. This choice can be refined by trial-and-error.

8.2 Embellishments

Any Physics student who has survived the gruelling ordeal of lab sessions will know the importance of labelling graphs clearly. Matlab provides this facility:



(a)



(b)

However, I prefer to do this kind of thing on the command line (it gets quicker with practice, and it can be automated for batches of plots):

- To create production-quality axis labels:

```
set(gca,'fontsize',18,'fontname','times new roman')
```

Here, 'gca' is a handle to the current axes ('get current axes').

- To label the graph:

  ```
  xlabel('x')
  ylabel('y=f(x)')
  ```

  The order is important here – you must change the font before drawing the labels; otherwise the labels will be in the default font (small and plain).

- For production-quality graphics, the thickness of the curve ('linewidth') should be set to three. This can be done via the editor, or immediately on creation of the plot, using instead the modifed plot command

  ```
  plot(x,fx,'linewidth',3)
  ```

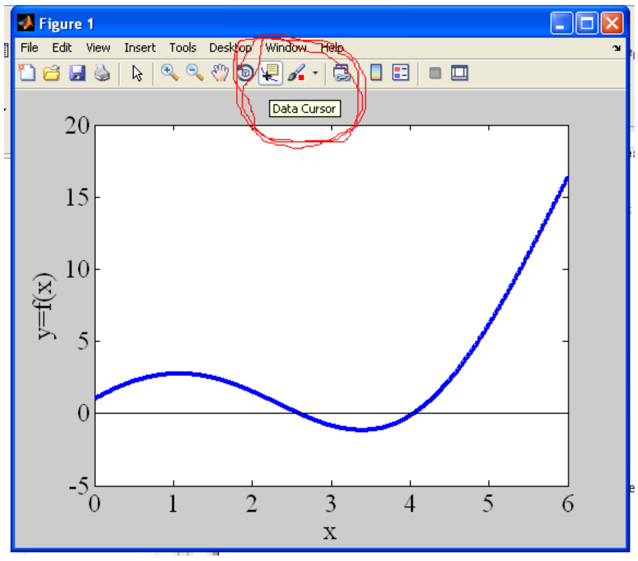
- Sometimes, the line $y = 0$ can be helpful in a plot to guide the eye. This can be included as follows:

  ```
  hold on
  plot(x,0*x,'linewidth',1,'color','black')
  hold off
  ```

  Here, the 'hold on' command holds the current figure in place so that another plot layer can be included. Without this 'hold on' command, the additional plot command would overwrite the first plot.

  The instruction `...,'color','black'` tells Matlab to plot the horizontal line in black. Matlab only takes American spellings!

- To pick out a particular point on the curve (e.g. a point where $y = f(x)$ hits zero, one can use the **data cursor**.



I think the final, embellished result is much nicer than our original attempts (Fig. 8.1)!

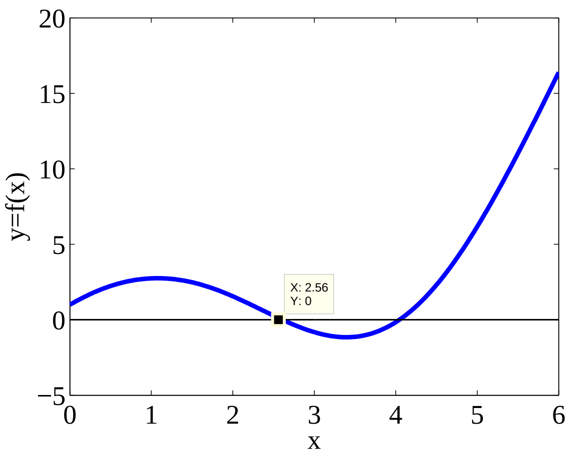


Figure 8.1: Final, embellished plot of $f(x) = \sin x + x \cos x + e^x/(1 + x^2)$ on the range $x \in [0, 6]$.

Chapter 9

Root-finding

Overview

In this chapter we study an elementary numerical method to compute roots of the problem

$$f(x) = 0,$$

where $f(x)$ is a continuous function.

9.1 Roots

Definition: Let $f : \mathbb{R} \to \mathbb{R}$ be a continuous function The value x_* is said to be a a **root** of f if

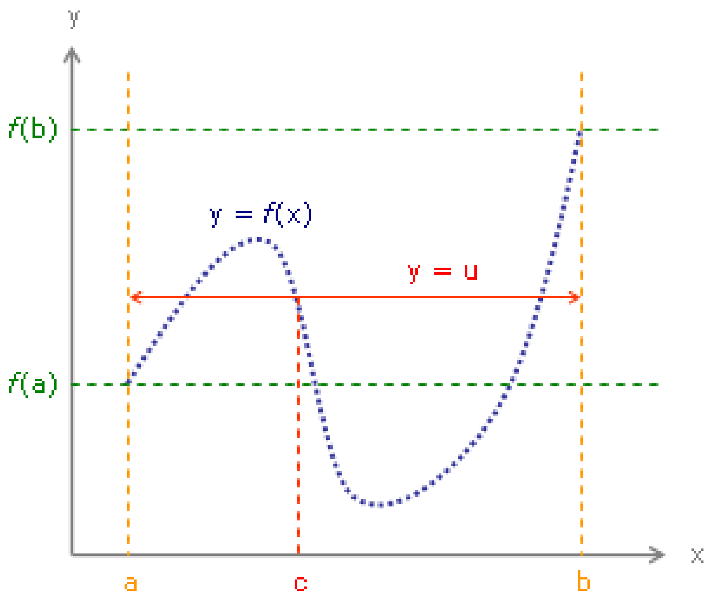
$$f(x_*) = 0.$$

Example: $x = 1$ is a root of $f(x) = x^2 - 3x + 2$ because $f(1) = 1 - 3 + 2 = 0$. There is no limit to the number or roots that a function may have. For example, the **quadratic function** just described has two roots, $x_* = 1, 2$. On the other hand, the function $f(x) = \sin x$ has infinitely many roots, $x_* = n\pi$, where $n \in \mathbb{Z}$. We do have some theorems however that tell us when **at least one root** should exist:

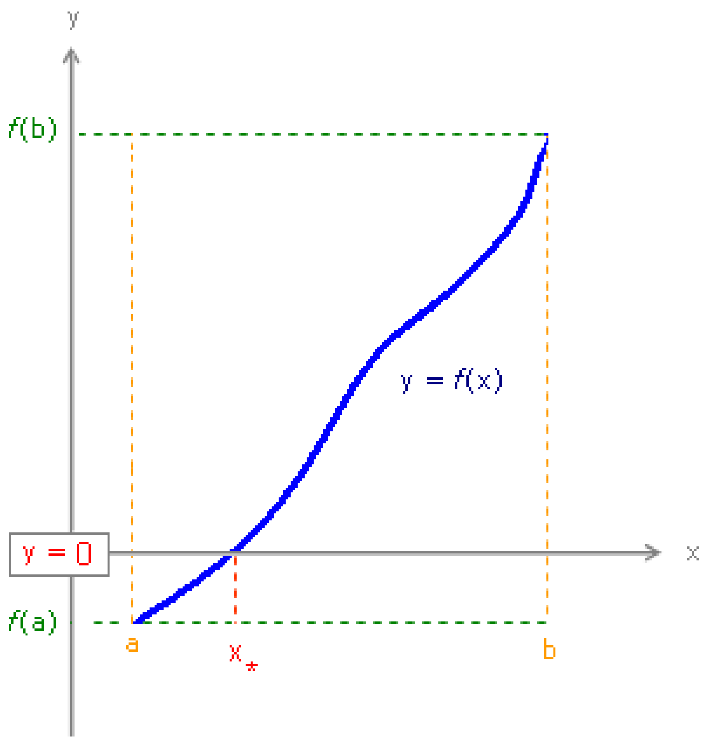
Theorem 9.1 (Intermediate Value Theorem) *Let $f : [a, b] \to \mathbb{R}$ be a continuous real-valued function, with $f(a) < f(b)$. Then for each real number u with $f(a) < u < f(b)$, there exists at least one value $c \in (a, b)$ such that $f(c) = u$.*

No proof is given here but see for example Beales (p. 105); see also Figure 9.1.

Corollary 9.1 *If $f : [a, b] \to \mathbb{R}$ is a continuous real-valued function with $f(a) < 0$ and $f(b) > 0$, then there exists at least one value $x_* \in (a, b)$ such that $f(x_*) = 0$, that is, f has a root strictly between a and b.*



(a)



(b)

Figure 9.1: Sketch for the Intermediate Value Theorem and its corollary.

9.2 Bracketing and Bisection

Let $f : [a,b] \rightarrow \mathbb{R}$ be a continuous function with $f(a) < 0$ and $f(b) > 0$. By the Intermediate Value Theorem, f has at least one root on (a,b). Bracketing and Bisection (B&B) is an algorithm for finding one of these roots:

1. Compute the midpoint $c_1 = (a+b)/2$.
2. Compute $f(c_1)$. If $f(c_1) < 0$ then focus on a new interval $[c_1, b]$. If $f(c_1) > 0$ then focus on a new interval $[a, c_1]$.
3. Compute the midpoint of the new interval, then repeat step 2.
4. Repeat indefinitely until convergence down to the required precision is obtained.

Steps (1)–(2) are shown schematically in Figure 9.2, and a sample MATLAB code is given here in what follows.

```
function xstar=do_bracketing_bisection(a,b)

% ************************************************************************
% Iterate until solution is root is converged to within the following
% tolerance.

tol=1e-16;

% ************************************************************************
%  Initial guess for the interval and for the root.

c1=a;
c2=b;

xstar_old=(c1+c2)/2;

% ************************************************************************
% Error checking: See if Bracketing and Bisection is possible.

if((f(a)*(f(b))>=0))
    display('bracketing and bisection not possible; exiting')
    xstar='rubbish';
    return
end

% ************************************************************************
% Error checking: See if initial guess is actually the root; if so,
% terminate program.

if(abs(f(xstar_old))<tol)
    display('initial guess hits root')
    xstar=xstar_old;
    return
end

% *************************************************************************
% First pass through the algorithm to find new value of xstar.
% There are two sub-algorithms:
% 1. One sub-algorithm if f(a)<0 and f(b)>0 -- the one described in the
% text
% 2.  Another sub-algorithm if f(a)>0 and f(b)<0.

cm=(c1+c2)/2;

if(f(a)<0)

    if( f(cm)<0)
        c1=cm;
        xstar=(c1+c2)/2;
    elseif (f(cm)>0)
        c2=cm;
        xstar=(c1+c2)/2;
    else
        xstar=    (c1+c2)/2;
    end

else

    if( f(cm)<0)
        c2=cm;
        xstar=(c1+c2)/2;
    elseif (f(cm)>0)
        c1=cm;
        xstar=(c1+c2)/2;
    else
        xstar=    (c1+c2)/2;
    end

end

% *************************************************************************
% Further passes through the algorithm using a WHILE loop.

% Structure for sub-algorithm 1:
%
% 1.  If f(cm)<0 then the new interval should be [cm,c2];
% 2.  If f(cm)>0 then the new interval should be [c1,cm];
% 3.  If f(cm)=0 then we have hit the root exactly and should exit the
% loop.

 if(f(a)<0)

     while(abs(xstar-xstar_old)>tol)
         cm=(c1+c2)/2;
          if( f(cm)<0)
              c1=cm;
              xstar_old=xstar;
              xstar=(c1+c2)/2;
          elseif (f(cm)>0)
              c2=cm;
              xstar_old=xstar;
              xstar=(c1+c2)/2;
          else
              xstar_old=(c1+c2)/2;
              xstar=     (c1+c2)/2;
          end
     end

 else
     while(abs(xstar-xstar_old)>tol)
         cm=(c1+c2)/2;
          if( f(cm)<0)
              c2=cm;
              xstar_old=xstar;
              xstar=(c1+c2)/2;
          elseif (f(cm)>0)
              c1=cm;
              xstar_old=xstar;
              xstar=(c1+c2)/2;
          else
              xstar_old=(c1+c2)/2;
              xstar=     (c1+c2)/2;
          end
     end

end

```

```
118
119
120 % ************************************************************************
121 % End of main program.
122
123
124 end
125
126 % ************************************************************************
127 % ************************************************************************
128 % Subfunction to evaluate y=f(x).
129
130 function y=f(x)
131     % y=x.^2-2;
132     % y=x.^3-2*x.^2+x-1;
133     % y=x.^3+10*x.^2+x-1;
134     y=sin(x);
135 end
```

sample_matlab_codes/do_bracketing_bisection.m

There is a lot to discuss in this code! Let's go through it line-by-line:

- Lines 12-15. Here I find the initial values for the interval, with $c_1 = a$ and $c_2 = b$. I make an initial guess for the root, namely $f[(c_1 + c_2)/2]$.

  Note that I am leaving the definition of $f(\cdot)$ in a subfunction. This is handy: the code can be easily recycled to compute the roots of many different continuous functions.

- Lines 20-24. Here I check to see if there really is a sign change, i.e. if $f(a)f(b) < 0$. If there is not a sign change, then bracketing and bisection will not work, and the code should be halted. Because the function must return a value, I set the variable `xstar` to equal a **string** called `rubbish`. A string is an array of characters.

- Lines 30-34. These lines are included in case we get very lucky. If we are very lucky, the starting-guess for the root will in fact be the root, to within machine precision. Then we should set $x_* = (c_1 + c_2)/2 = (a + b)/2$ and exit the code.

- Lines 43-69. A first pass through the algorithm (i.e. Steps 1 and 2). I have to split up the algorithm into two sub-algorithms:

  1. When $f(a) < 0$ and $f(b) > 0$;
  2. When $f(a) > 0$ and $f(b) < 0$,

since conceptually, there is no reason why B&B should not work in the second case. Let's focus on the first sub-algorithm. I compute the midpoint $c_m = (c_1 + c_2)/2$ and evaluate $f(c_m)$. Since $c_1 = a$ and $c_2 = b$, there are two possibilities:

Case 1	Case 2
$\boldsymbol{f(c_1) < 0}$	$\boldsymbol{f(c_1) < 0}$
	$f(c_m) > 0$
$f(c_m) < 0$	
$\boldsymbol{f(c_2) > 0}$	$\boldsymbol{f(c_2) > 0}$

In Case 1 I take my new interval to be $[c_m, c_2]$ and in Case 2 I take my new interval to be $[c_1, c_m]$. I compute my new estimate of the root using the new interval endpoints: $x_{*\text{new}} = (c_1 + c_2)/2$.

- Lines 81-116. I check the difference between the initial guess and the new guess $|x_* - x_{*\text{new}}|$. If this is too large, I repeat steps (1)–(2) of the algorithm. Again, two sub-algorithms are considered.

- Lines 85–96. The first sub-algorithm again with $f(a) < 0$. I repeat steps (1)–(2), very similar to Lines 43–69. An extra step is included in here, namely the possibility to break out of the `while` loop if the estimated value of the root is in fact the true root, i.e. if $f(c_m) = 0$. **Note the application of the very useful `elseif` statement here.**

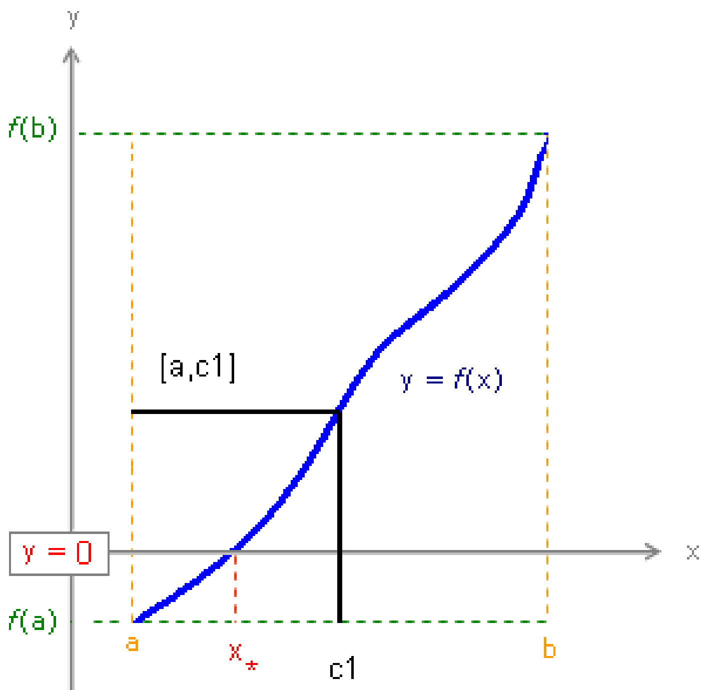


Figure 9.2: Sketch for Bracketing and Bisection

Convergence analysis

At each level n of iteration, the estimate of the root is

$$x_{*n} = \frac{c_{1n} + c_{2n}}{2},$$

and the maximum possible distance between the estimated value of the root and the true value is given by

$$\mathrm{Error}(n) = \max\left(|c_{2n} - x_{*n}|, |x_{*n} - c_{1n}|\right).$$

We have

$$\mathrm{Error}(n) = \max\left(|c_{2n} - x_{*n}|, |x_{*n} - c_{1n}|\right) \le \frac{|c_{2n} - c_{1n}|}{2} := \delta_n.$$

Thus, at the zeroth level of iteration, we have

$$\delta_0 = |b - a|.$$

At the first level, we have (case 1) $c_1 = a$ and $c_2 = (a+b)/2$ or (case 2) $c_1 = (a+b)/2$ and $c_2 = b$. In either case,

$$\delta_1 = \frac{|b-a|}{2}.$$

Guessing the pattern, or doing a proper proof by induction, we have

$$\mathrm{Error}(n) \le \delta_n = \frac{|b-a|}{2^n}.$$

Also,

$$\frac{\delta_{n+1}}{\delta_n} = \tfrac{1}{2}$$

is a constant, so the maximum possible error δ_n **converges linearly** as $n \to \infty$. As we shall see later, linear convergence is rather slow, and B&B is not normally used as the sole method by which a root is found.

Failure analysis

When applied to a continuous function on an interval where a sign change occurs, Bracketing and Bisection will never fail. It will converge (slowly) to a root. Ambiguity can occur however when the continuous function possesses multiple roots on the interval (e.g. $f(x) = \sin(x)$ on $x \in (-\pi/2, 5\pi/2)$, with roots at $0, \pi, 2\pi$, and $\sin(-\pi/2) = -1$ and $\sin(5\pi/2) = +1$. In this case, B&B will converge to **one** of the roots; however, it is not obvious in advance which root will be selected.

Brackecketing and Bisection is therefore robust but slow. In the next chapter we examine a method with the opposite properties. The goal is to combine these two methods to produce a hybrid scheme that is robust and fast.

Chapter 10

The Newton–Raphson method

Overview

In this chapter we study the Newton–Raphson method for solving

$$f(x) = 0,$$

where $f(x)$ is a **differentiable** function.

10.1 The idea

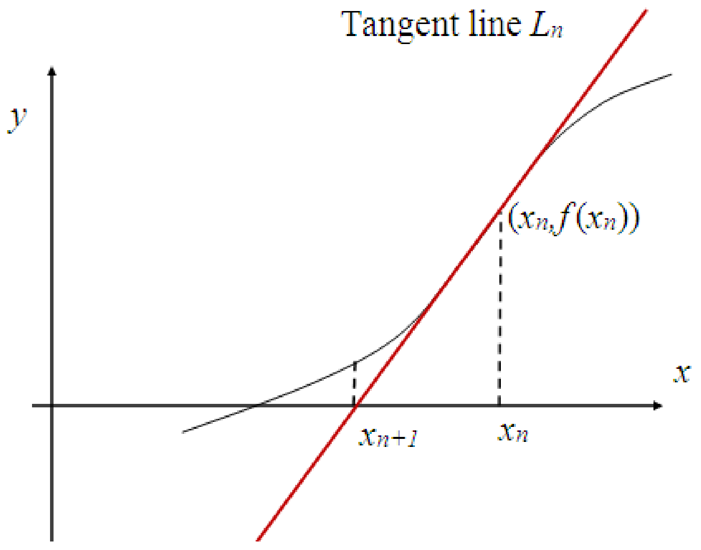


Figure 10.1: Sketch for the Newton–Raphson method

Let $f : [a, b] \to \mathbb{R}$ be a **differentiable** function on (a, b), with at least one root in the interval

(a,b). Start with a guess for the root x_n. We refine the guess as follows. Referring to Figure 10.1, construct the tangent line to $f(x_n)$, called L_n. The slope is $f'(x_n)$ and a point on the line is $(x_n, f(x_n))$. We have

$$L_n : y - f(x_n) = f'(x_n)[x - x_n]. \tag{10.1}$$

Our next level of refinement for the root – x_{n+1} is got by moving along the tangent line L_n until the x-axis is crossed. Using Equation (10.1), this is

$$0 - f(x_n) = f'(x_n)[x_{n+1} - x_n].$$

Re-arranging, this is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \tag{10.2}$$

provided of course the tangent line has finite slope. The method (10.2), supplemented with a starting value, is called the Newton–Raphson method for root-finding:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \qquad x_0 \text{ given.} \tag{10.3}$$

Error analysis

In this section, we require that f be C^2 on any interval of interest, and that $f'(x) \neq 0$ on the same interval. We let $\epsilon_n = x_* - x_n$ be the difference between the root and the n^{th} level of approximation. Then,

$$\begin{aligned}\epsilon_{n+1} &= x_* - x_{n+1}, \\ &= x_* - \left(x_n - \frac{f(x_n)}{f'(x_n)}\right), \\ &= \underbrace{(x_* - x_n)}_{=\epsilon_n} + \frac{f(x_n)}{f'(x_n)}.\end{aligned} \tag{10.4}$$

Also, by definition

$$f(x_*) = f(\epsilon_n + x_n) = 0.$$

Hence, by Taylor's remainder theorem, we have the **exact** expression

$$f(x_n) + f'(x_n)\epsilon_n + \tfrac{1}{2}f''(\eta)\epsilon_n^2 = 0, \qquad \eta \in [x_n, x_n + \epsilon_n].$$

Re-arrange:

$$\frac{f(x_n)}{f'(x_n)} = -\epsilon_n \left[1 + \tfrac{1}{2}\frac{f''(\eta)}{f'(x_n)}\epsilon_n\right]. \tag{10.5}$$

Combine Equations (10.4) and (10.5):

$$\begin{aligned}\epsilon_{n+1} &= \epsilon_n - \epsilon_n\left[1 + \tfrac{1}{2}\frac{f''(\eta)}{f'(x_n)}\epsilon_n\right],\\ &= -\left[\tfrac{1}{2}\frac{f''(\eta)}{f'(x_n)}\epsilon_n^2\right]\end{aligned}$$

Thus,

$$\epsilon_{n+1} = \tfrac{1}{2}\frac{f''(\eta)}{f'(x_n)}\epsilon_n^2.$$

Taking absolute values with $\delta_n := |\epsilon_n|$ &c., this becomes

$$\delta_{n+1} = \left|\tfrac{1}{2}\frac{f''(\eta)}{f'(x_n)}\right|\delta_n^2.$$

An upper limit on the error is

$$\delta_{n+1} = M\delta_n^2, \tag{10.6}$$

where

$$M = \sup_{\substack{x\in(a,b)\\ y\in(a,b)}}\left|\tfrac{1}{2}\frac{f''(x)}{f'(y)}\right|.$$

The convergence in the Newton–Raphson method is called **quadratic** because, by Equation (10.6), $\delta_{n+1} \propto \delta_n^2$.

It would now seem that we have a rather awesome numerical method for root finding, with excellent convergence properties. However, the result (10.6) should be regarded only as 'local': it guarantees fast convergence **only if** δ_0 is small. In other words, if an initial guess is a small distance away from a root, then the guess will converge quadratically fast to the true root. However, the method is very sensitive, and in the next chapters we investigate what happens if the initial guess is not close to the root.

Chapter 11

Interlude: One-dimensional maps

Overview

The failure analysis for the Newton–Raphson method is linked intimately to the study of one-dimensional maps. For that reason, we make a brief interlude and study such maps: their definition, the notion of fixed points, stability, and periodic orbits.

11.1 Definitions

Definition 11.1 *A sequence x is a map from non-negative integers to the real numbers:*

$$\begin{aligned} x : \{0\} \cup \mathbb{N} &\rightarrow \mathbb{R}, \\ n &\mapsto x_n. \end{aligned}$$

Example:

$$\{0\} \cup \mathbb{N} \rightarrow \left\{ 0, 1, \frac{1}{2^2}, \frac{1}{3^2}, \frac{1}{4^2}, \cdots \right\}$$

is a sequence.

Definition 11.2 *An autonomous discrete map F is a sequence where the $(n+1)^{th}$ element depends on the n^{th} element through a definite functional form:*

$$x_{n+1} = F(x_n),$$

and where starting value x_0 is also specified.

Example:

$$x_{n+1} = \lambda x_n + \sin(2\pi x_n), \qquad \lambda \in \mathbb{R}$$

is a discrete autonomous map.

Another example is the root-finding procedure in the Newton–Raphson method:

$$x_{n+1} = F(x_n), \qquad F(x) = x - \frac{f(x)}{f'(x)}.$$

There are more general discrete maps, such as

$$x_{n+1} = F(x_n, x_{n-1}).$$

Such maps, involving more than two levels, are often called **difference equations**. We do not discuss these any further.

11.2 Fixed points and stability

Definition 11.3 *Let*

$$x_{n+1} = F(x_n)$$

be a discrete autonomous map. The **fixed points** *of the map are those values x_* for which*

$$F(x_*) = x_*.$$

Theorem 11.1 (Fixed points of the Newton–Raphson map) *Let*

$$x_{n+1} = F(x_n), \qquad F(x) = x - \frac{f(x)}{f'(x)}$$

be the Newton–Raphson dynamical system. Then the fixed points of the dynamical system are the roots of $f(x)$.

Proof: Set $x_* = F(x_*)$, i.e.

$$x_* = F(x_*) = x_* - \frac{f(x_*)}{f'(x_*)}$$

Cancellation yields

$$\frac{f(x_*)}{f'(x_*)} = 0,$$

hence $f(x_*)$=0.

Definition 11.4 *Let*

$$x_{n+1} = F(x_n)$$

be a discrete autonomous map with a fixed point at x_*.

- *The fixed point is called* **stable** *if $|F'(x_*)| < 1$;*
- *The fixed point is called* **unstable** *if $|F'(x_*)| > 1$.*

The reason for this definition is the following. Suppose the initial condition for the map $x_{n+1} = F(x_n)$ is near the fixed point:

$$x_{n=0} = x_* + \delta_0, \qquad \delta_0 \ll 1.$$

We want to know what the next value of x will be:

$$x_{n=1} = F(x_{n=0}) = F(x_* + \delta_0).$$

Now δ_0 is small, so we can do a **Taylor expansion**:

$$F(x_* + \delta_0) = F(x_*) + F'(x_*)\delta_0 + \tfrac{1}{2}F''(x_*)\delta_0^2 + \cdots .$$

However, δ_0 is so small that we are going to ignore the quadratic terms:

$$F(x_* + \delta_0) \approx F(x_*) + F'(x_*)\delta_0 = x_* + F'(x_*)\delta_0$$

since $F(x_*) = x_*$. Hence,

$$x_{n=1} = x_* + F'(x_*)\delta_0.$$

Let us introduce δ_1 such that $x_{n=1} = x_* + \delta_1$. Thus,

$$\delta_1 = F'(x_*)\delta_0.$$

Imagine repeating the map n times, such that

$$\delta_{n+1} = F'(x_*)\delta_n.$$

This equation is linear and has solution

$$\delta_n = \delta_0 \left[F'(x_*)\right]^n .$$

- If $|F'(x_*)| < 0$, then $\lim_{n\to\infty} \delta_n = 0$, or $\lim_{n\to\infty} x_n = x_*$;

- If $|F'(x_*)| > 0$, then $\lim_{n\to\infty} \delta_n = \infty$, and $\lim_{n\to\infty} x_n$ is undetermined from the linearized analysis.

- In the first case, if the system (the map and the x-values) starts near the fixed point, it stays near the fixed point – the fixed point is stable;

- In the second case, if the system starts near the fixed point, it moves away from the fixed point **exponentially fast** – the fixed point is unstable.

Exercise 11.1 *Let x_* be a fixed point of the Newton–Raphson map. Analyse the behaviour of the map near a fixed point by showing that $F'(x_*) = 0$. Such a fixed point is called* **superstable**.

Chapter 12

Newton–Raphson method: Failure analysis

Overview

We classify the different ways in which the Newton–Raphson method can fail. We apply the theory of one-dimensional maps to analysing these failures. Finally, we examine Matlab's own built-in method for root finding.

12.1 One-dimensional maps and failure analysis for the Newton–Raphson method

Let $f(x)$ be a differentiable function with at least one real root. Consider the associated Newton–Raphson root-finding method:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \tag{12.1}$$

for a given starting-point x_0. Calling

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

and

$$x_k = \underbrace{F(F(\cdots(x_0)))}_{k\text{times}}$$

there are at least five ways in which a poor choice for the starting value can lead to a catastrophic failure of the Newton–Raphson algorithm:

1. x_0 such that $F(x_0)$ is undefined.

2. Predecessors of type-1 points: x_0 with $F(x_n)$ undefined for some positive-integer value of n.
3. Periodic orbits: x_0 such that $x_0 = F(x_{0+n})$ for some positive-integer value of n.
4. Predecessors of periodic orbits: points x_0 such that $\lim_{k\to\infty} x_k$ is a periodic orbit.
5. Divergence to infinity: Points x_0 such that $\lim_{k\to\infty} |x_k| = \infty$.

As an example of failure analysis using this list, we examine the Newton–Raphson method applied to the problem of finding the roots of $f(x) = x^3 - x$. The analysis is simplified in this instance because the roots are obvious: $x = \pm 1$ and $x = 0$. Of course, the Newton–Raphson method can also be used to find these roots:

$$x_{n+1} = F(x_n), \qquad F(x) = x - \frac{x^3 - x}{3x^2 - 1}, \tag{12.2}$$

with some starting value x_0. We shall examine how the choice of starting value x_0 affects the performance of the algorithm (12.2).

Type 1 points

With the starting value

$$x_0 = \pm\tfrac{1}{\sqrt{3}},$$

we obtain $x_1 = \infty$, since the derivative of $f(x)$ vanishes at $x = \pm 1/\sqrt{3}$. These are then **type 1 points**

It is of interest to know if there are any predecessors of type-1 points. For that reason, we construct **cobweb diagrams** for the map $x_{n+1} = F(x_n)$. Given a generic map of this form, it is possible to determine graphically where a given starting condition x_0 will end up, that is, there is a graphical way of determining

$$\lim_{n\to\infty} x_n, \qquad x_{n+1} = F(x_n), \qquad x(n=0) = x_0.$$

This is called a **cobweb**, and is computed according to the following recipe:

- Plot $y = F(x)$ on the allowed range. For the NR map, $x \in \mathbb{R}$, with singular points at $x = \pm 1/\sqrt{3}$.
- Plot $y = x$.
- Take the starting point x_0 and match it up with a point on $y = f(x)$, giving x_1.
- Draw a horizontal line from x_1 until it hits the line $y = x$.

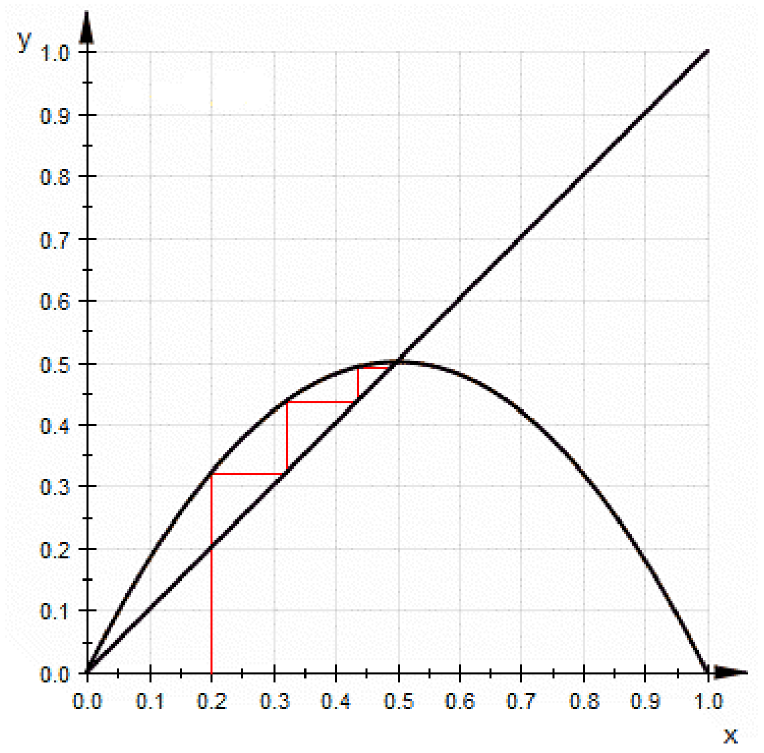


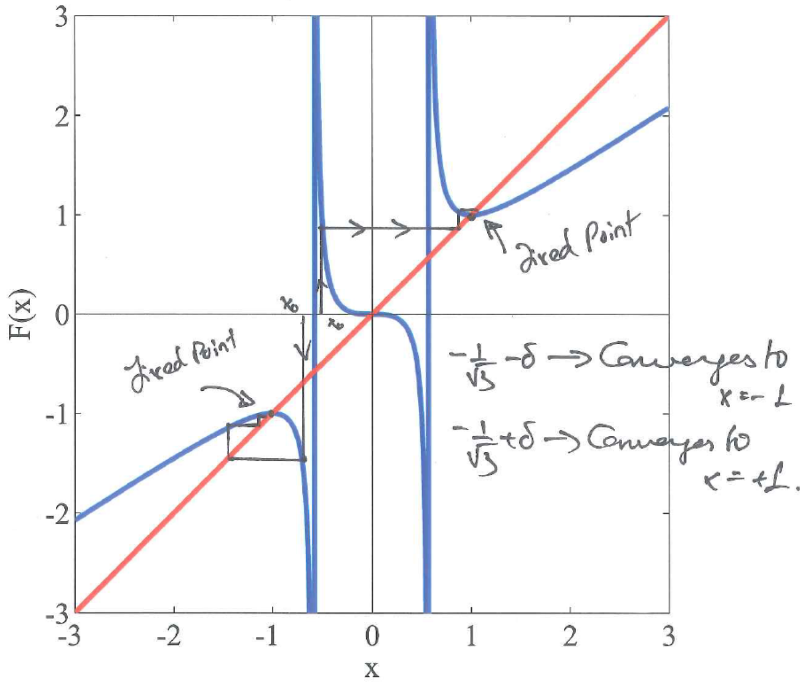
Figure 12.1: Cobweb diagram for the map $x_{n+1} = 2x_n(1 - x_n)$, with $x_0 \geq 0$.

- From this point, draw a vertical line until such that the curve $y = f(x)$ is intersected again. This is x_2.
- Repeat the procedure.

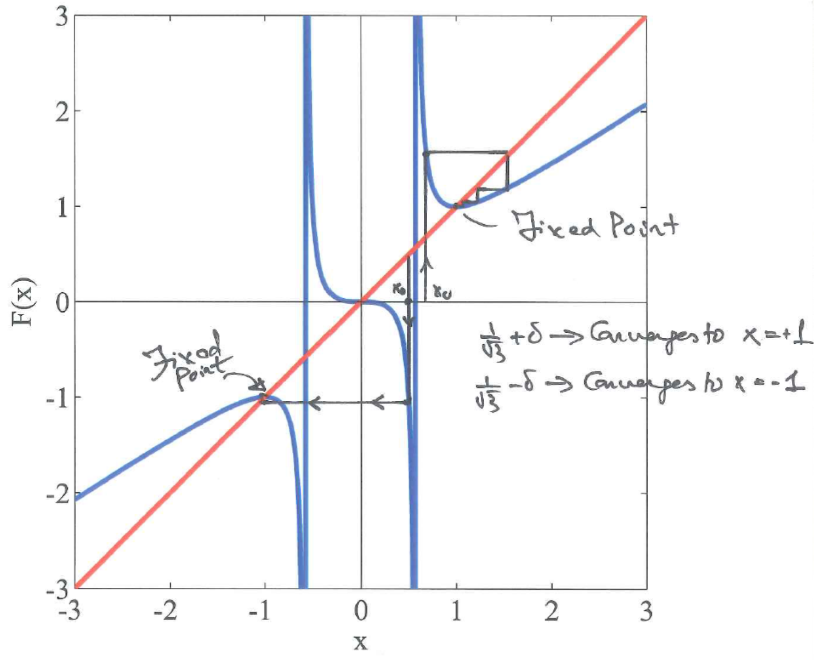
Simple example: Use a cobweb diagram to find the fixed points of the map $x_{n+1} = 2x_n(1 - x_n)$, with $x_0 \geq 0$.

First, the fixed points are found using analytic methods by setting $x_* = 2x_*(1 - x_*)$, hence $x_* = 0$ or $x_* = 1/2$. The corresponding cobweb diagram is shown in Figure 12.1.

Next, cobweb diagrams for the more involved Newton–Raphson map (12.2) are shown here:



(a)



(b)

Figure 12.2: Cobweb procedure for the Newton–Raphson map $x_{n+1} = x_n - [(x_n^3 - x)/(3x_n^2 - 1)]$

In the figure, we have implemented the cobweb procedure for starting values $x_0 = \pm 1/\sqrt{3} \pm \delta$, with δ small and positive. For $|\delta|$ small, after a few iterations, we arrive at one of the fixed points $x_* = \pm 1$.

Period-2 orbits

Definition 12.1 *Let $x_p = F(x_n)$ be a discrete map. A* **period-two orbit** *is a point x_p such that*

$$x_p = F(F(x_p)).$$

We apply this definition to Equation (12.2) by supposing a starting-value x_0 can be found such that $x_0 = F(F(x_0))$. Then, the iterative method (12.2) will not converge on the root, instead, it will cycle through the sequence $\{x_0, F(x_0), x_0, F(x_0), \cdots\}$ indefinitely. Indeed, such starting values do in fact exist for the particular map (12.2); we find them now by solving

$$x_p = F(F(x_p)).$$

To simplify, we rewrite

$$F(x) = x - \frac{x^3 - x}{3x^2 - 1} = \frac{x(3x^2-1) - x^3 + x}{3x^2 - 1} = \frac{2x^3}{3x^2 - 1}.$$

We are to solve

$$x_p = F(y_p), \qquad y_p = F(x_p).$$

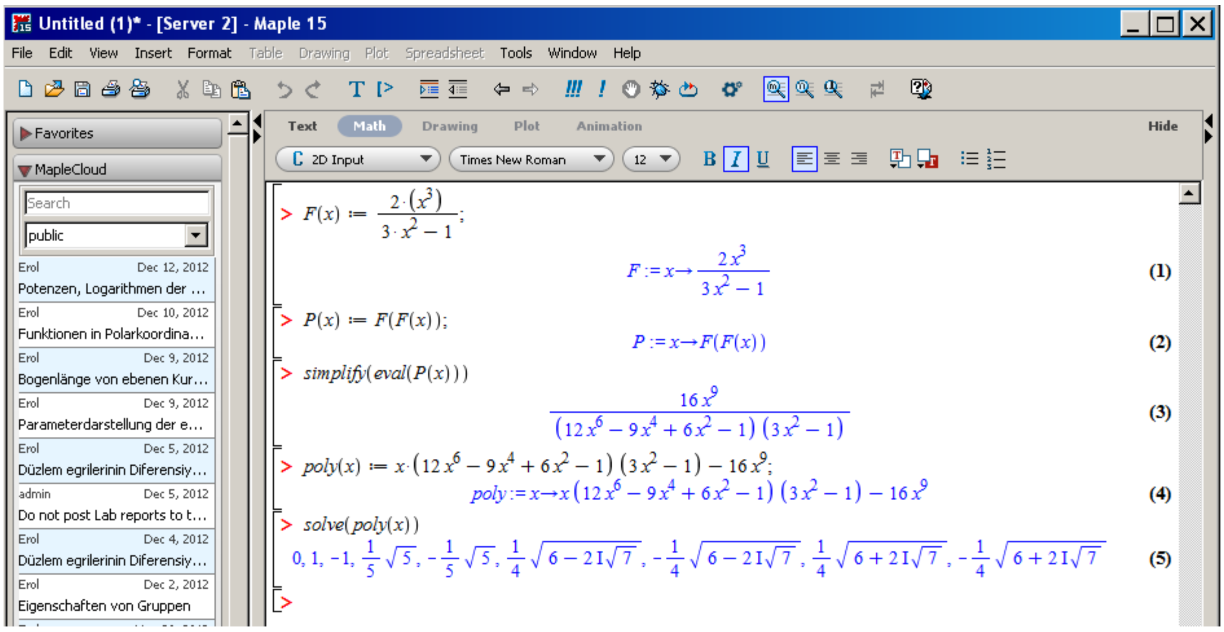
We have,

$$\begin{aligned} x_p &= \frac{2y_p^3}{3y_p^2 - 1}, \\ &= \frac{2\left(\frac{2x_p^3}{3x_p^2-1}\right)^3}{3\left(\frac{2x_p^3}{3x_p^2-1}\right)^2 - 1}, \\ &= \frac{16x_p^9}{(3x_p^2 - 1)\left[12x_p^6 - (3x_p^2-1)^2\right]}. \end{aligned}$$

Hence, the periodic orbits are obtained as roots of the equation

$$16x_p^9 - x_p(3x_p^2 - 1)\left[12x_p^6 - (3x_p^2 - 1)^2\right] = 0. \tag{12.3}$$

The leading-order term here is $-20x_p^9$ – this is a ninth-order polynomial. In fact, the roots $x = 0$, $x = \pm 1$ are (trivially) period-two orbits, so $x(x-1)(x+1)$ are factors of Equation (12.3). By dividing out by these factors, we would be left with a sixth-order polynomial whose real roots would give possible period-two orbits. However, this division, and the resulting sixth-order polynomial root-finding problem would be very difficult to solve – especially in this case where a precise analytical answer is necessary. We therefore resort to a different (analytic) method to extract the roots – the symbolic software tool Maple. This is done in Figure 12.3. The real roots are $x_p = \pm 1/\sqrt{5}$. These


Untitled (1)* - [Server 2] - Maple 15
File Edit View Insert Format Table Drawing Plot Spreadsheet Tools Window Help
Favorites
MapleCloud
Search
public
Erol Dec 12, 2012 Potenzen, Logarithmen der ...
Erol Dec 10, 2012 Funktionen in Polarkoordina...
Erol Dec 9, 2012 Bogenlänge von ebenen Kur...
Erol Dec 9, 2012 Parameterdarstellung der e...
Erol Dec 5, 2012 Düzlem egrilerinin Diferensiy...
admin Dec 5, 2012 Do not post Lab reports to t...
Erol Dec 4, 2012 Düzlem egrilerinin Diferensiy...
Erol Dec 2, 2012 Eigenschaften von Gruppen
Text Math Drawing Plot Animation Hide
2D Input Times New Roman 12

> $F(x) := \dfrac{2\cdot(x^3)}{3\cdot x^2-1};$

$$F := x \to \frac{2x^3}{3x^2-1} \qquad (1)$$

> $P(x) := F(F(x));$

$$P := x \to F(F(x)) \qquad (2)$$

> $simplify(eval(P(x)))$

$$\frac{16x^9}{(12x^6-9x^4+6x^2-1)(3x^2-1)} \qquad (3)$$

> $poly(x) := x\cdot(12x^6-9x^4+6x^2-1)(3x^2-1)-16x^9;$

$$poly := x \to x\,(12x^6-9x^4+6x^2-1)(3x^2-1)-16x^9 \qquad (4)$$

> $solve(poly(x))$

$$0, 1, -1, \frac{1}{5}\sqrt{5}, -\frac{1}{5}\sqrt{5}, \frac{1}{4}\sqrt{6-2\mathrm{I}\sqrt{7}}, -\frac{1}{4}\sqrt{6-2\mathrm{I}\sqrt{7}}, \frac{1}{4}\sqrt{6+2\mathrm{I}\sqrt{7}}, -\frac{1}{4}\sqrt{6+2\mathrm{I}\sqrt{7}} \qquad (5)$$

>

Figure 12.3: Maple procedure for finding the periodic orbits of the Newton–Raphson map $x_{n+1} = x_n - [(x_n^3 - x)/(3x_n^2 - 1)]$

are therefore the periodic orbits. Thus, starting at an initial guess $x_0 = 1/\sqrt{5}$, the Newton–Raphson method would give a sequence $\{1/\sqrt{5}, -1/\sqrt{5}, 1/\sqrt{5}, -1/\sqrt{5}, \cdots\}$, never converging on any of the roots $x = 0, \pm 1$. **This is a disaster!**

The good news is that these periodic orbits are unstable: starting-values x_0 near $\pm 1/\sqrt{5}$ rapidly move away from $\pm 1/\sqrt{5}$. This is readily shown because the because the stability parameter for a period-2 orbit is

$$\lambda_2 = \left| \frac{d}{dx} F(F(x))\Big|_{x_p} \right|.$$

We compute:

$$\begin{aligned}
\lambda_2 &= \left| F'(F(x_p))F'(x_p) \right|, \\
&= \left| F'(x_{p1})F'(x_{p2}) \right|, \\
&= \left| \left[\frac{6x_{p1}^2(x_{p1}^2-1)}{(3x_{p1}^2-1)^2} \right] \left[\frac{6x_{p2}^2(x_{p2}^2-1)}{(3x_{p2}^2-1)^2} \right] \right|, \\
&= 36
\end{aligned}$$

Hence, $\lambda_2 > 1$ and the periodic orbits are unstable. Thus, although disastrous, the starting-value $x_0 = \pm 1/\sqrt{5}$ is not 'too disastrous', as any starting-value $x_0 = \pm 1/\sqrt{5} \pm \delta$ with $\delta > 0$ small will lead to a non-periodic orbit and (perhaps) convergence to the roots.

The following list is a summary of our findings:

1. For $x_0 = \pm 1/\sqrt{3}$ a type-1 singularity is obtained;

2. For x_0 in the neighbourhood of $\pm 1/\sqrt{3}$ the Newton–Raphson iterates tend to ± 1.

3. For $x_0 = \pm 1/\sqrt{5}$ a periodic orbit is obtained;

4. For x_0 in the neighbourhood of $1/\sqrt{5}$ the Newton–Raphson iterates tend either to ± 1 or 0.

In short, the eventual outcome depends sensitively on the choice of starting-value x_0 – the Newton–Raphson map is **chaotic**.

Other possible failures

For a continuous function, a sign change in an interval indicates the existence of a root in that interval. However, naively applying this result to functions with a singular point can lead to a completely erroneous answer. For example, the function

$$f(x) = \frac{1}{1-x} \tag{12.4}$$

has a singularity at $x = 1$ and asymptotes to $x = -\infty$ as $x \downarrow 1$ and asymptotes to $x = +\infty$ as $x \uparrow 1$. In this case, it is easy to derive the following asymptotic behaviours of the Newton–Raphson map associated with Equation (12.4)

$$\begin{aligned} x_0 &= 1: \ x_n = 1, \ \ n = 1, 2, \cdots, & \text{(12.5a)} \\ x_0 &> 1: \ \lim_{n\to\infty} x_n = +\infty, & \text{(12.5b)} \\ x_0 &< 1: \ \lim_{n\to\infty} x_n = -\infty. & \text{(12.5c)} \end{aligned}$$

Thus, $x_0 = 1$ is an unstable fixed point and all other starting values are type-5 divergences.

Exercise 12.1 *Prove the result* (12.5).

12.2 Conclusions from failure analysis

We have seen that the Newton–Raphson method has rather odd (almost contradictory) properties:

1. Quadratic convergence: For x_0 sufficiently close to a root, the deviation of the estimated root from the true root decreases as the square of the deviation itself (i.e. 'very fast').

2. Sensitive dependence on initial conditions ('chaos'): For x_0 not sufficiently close to a root, the Newton–Raphson method can exhibit wild behaviour, including periodic orbits, singularities, and asymptotic divergences.

Clearly then, in order for the Newton–Raphson method to succeed, the initial guess must be a judicious one. In practice, Bracketing and Bisection is used to 'zoom in' on a region of interest containing a root and the Newton–Raphson method (or some variant thereof) is used to compute the numerical root with more precision.

12.3 Matlab's built-in root-finding function

Matlab has its own built-in root-finding function. It starts by doing Bracketing and Bisection until it is fairly confident that it has found a small region where the desired root exists. It then homes in on the root by using the secant method – which is a variant on the NR method. This switch in methods gives the best of both worlds – at the beginning of the calculation, the method is robust and does not do anything crazy (such as entering into a periodic orbit). At the end of the calculation, the method is fast and accurate, because it takes advantage of the quadratic convergence of the NR-type method. However, one must still provide a starting-guess for the calculation.

Suppose we are interested in the roots of the function

$$f(x) = \sin x + x\cos x + \frac{\mathrm{e}^x}{1+x^2}.$$

A sensible thing is first of all to plot this function and to get an idea of where the roots lie (in practice this is not always possible):

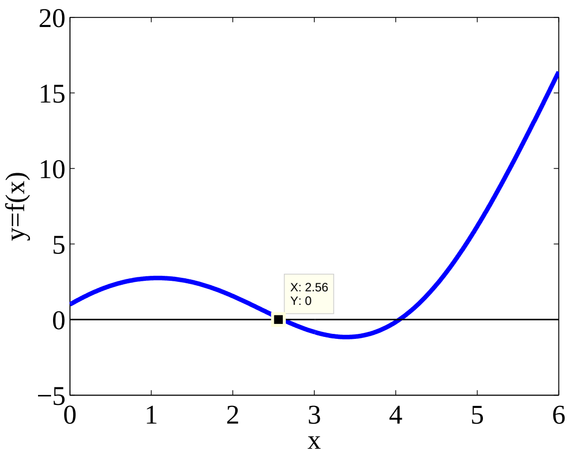


Figure 12.4: Plot of $f(x) = \sin x + x\cos x + \mathrm{e}^x/(1+x^2)$ on the range $x \in [0,6]$.

We see that a the point $x = 2.6$ is very close to a root. This is therefore a good starting-value for our root-finding calculation. We would proceed as follows and write a small Matlab function:

```
function xstar=get_roots()

% Matlab function to find roots of f(x)=0.
% The starting-value must be provided by the user.

% Starting value:
x0=2.6;
xstar = fzero(@myfun,x0);


function fx=myfun(x)
    % User-defined function - put whatever you like here, but watch
    % out for ./ and .*
    fx=sin(x)+x.*cos(x)+exp(x)./(1+x.^2);
end

end
```

sample_matlab_codes/get_roots.m

Exercise 12.2 *Write a code that starts with a continuous function $f : [a, b] \to \mathbb{R}$ such that $f(a) < 0$ and $f(b) > 0$. Use Bracketing and Bisection to compute an estimate of the root, down to a tolerance of 10^{-3}. Then, use this value as a starting-point for a Newton–Raphson iteration and iterate to find the root down to machine precision. Execute your built-in function 10,000 times inside a loop and do the same thing for the Matlab version. Find the average execution time in each case.*

Chapter 13

Numerical Quadrature – Introduction

Overview

In this chapter we study numerical methods to compute the definite integral

$$I = \int_a^b f(x)\mathrm{d}x,$$

where $f(x)$ is a continuous function.

13.1 Riemann Sums

We first of all recall the definition of the **Riemann integral**. We consider a generic continuous function

$$\begin{aligned} f : [a,b] &\rightarrow \mathbb{R}, \\ x &\mapsto f(x), \end{aligned}$$

with the positivity property

$$f(x) > 0 \,\text{on}\, [a,b].$$

We consider the problem of finding the area under the curve $y = f(x)$.

We form a uniform partition of $[a,b]$,

$$[a,b] = [x_0,x_1] \cup [x_1,x_2] \cup \cdots \cup [x_{N-1},x_N], \qquad a = x_0, \qquad b = x_N,$$
$$x_i = a + i\left(\frac{b-a}{N}\right), \; i = 0,1,\cdots,N.$$

We form the following **upper and lower sums**:

$$\text{Upper}: \qquad \mathcal{U}_N := \sum_{i=0}^{N-1} \left[\sup_{x\in[x_i,x_{i+1}]} f(x) \right] (x_{i+1}-x_i)\,,$$
$$\text{Lower}: \qquad \mathcal{L}_N := \sum_{i=0}^{N-1} \left[\inf_{x\in[x_i,x_{i+1}]} f(x) \right] (x_{i+1}-x_i)\,.$$

The function f is called **Riemann-integrable** with Riemann integral I if

$$\lim_{N\to\infty} \mathcal{U}_N = \lim_{N\to\infty} \mathcal{L}_N := I. \tag{13.1}$$

See Figure 13.1 for the accompanying sketch.

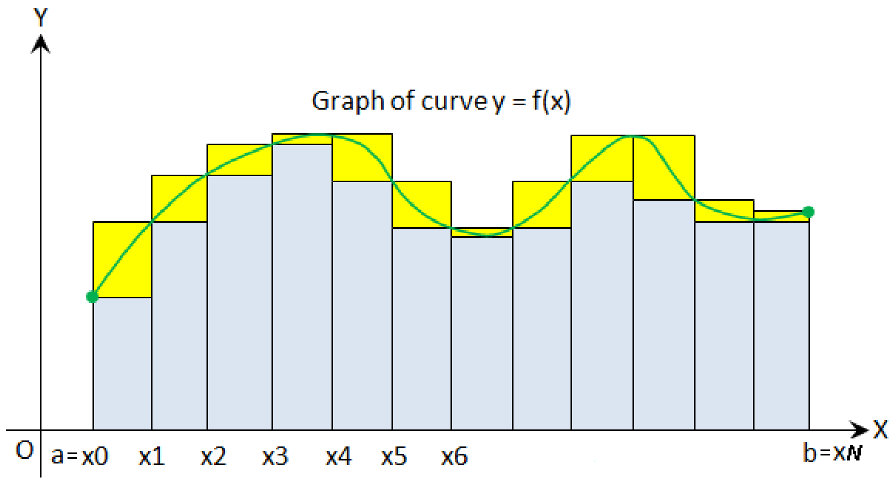


Figure 13.1: Upper and lower sums for Riemann integration

We have the following important result:

Theorem 13.1 (Riemann integrability for continuous functions) *Let $f:[a,b]\to\mathbb{R}$ be continuous on $[a,b]$ with the positivity property $f(x)>0$. Then f is Riemann integrable.*

No proof is provided here – this is not an analysis class, but see Beals (p. 107) A further result enables an obvious generalization of Riemann integration to non-positive functions:

Theorem 13.2 *Let $f:[a,b]\to\mathbb{R}$ be continuous on $[a,b]$ with the positivity property $f(x)>0$, and let $c\in(a,b)$. Then*

$$\int_a^b f(x)\,\mathrm{d}x = \int_a^c f(x)\,\mathrm{d}x + \int_c^b f(x)\,\mathrm{d}x.$$

Hence, we have the following definition:

Definition 13.1 *Let $f:[a,b]\to\mathbb{R}$ be continuous on $[a,b]$ with $c\in(a,b)$ such that $f(x)\le 0$ on $[a,c]$ and $f(x)\ge 0$ on $[c,b]$. Then*

$$\int_a^b f(x)\,\mathrm{d}x := -\int_a^c [-f(x)]\,\mathrm{d}x + \int_c^b f(x)\,\mathrm{d}x. \tag{13.2}$$

Definition (13.1) is extended in an obvious way to continuous functions with many zeros on a given interval of integration. Moreover, the Riemann-sum definition (13.1) can be applied to non-positive functions immediately without the intermediate step in Equation (13.1). These concepts will now be helpful in developing a theory of numerical quadratures.

13.2 Approximation by Riemann sums – Midpoint Rule

For the duration of this chapter, we regard $f:[a,b]\to\mathbb{R}$ to be a continuous function with arbitrarily many roots on $[a,b]$. We are interested as before in numerical approximations to $I=\int_a^b f(x)\,\mathrm{d}x$. Consider therefore the following sum:

$$\mathcal{S}_N=\sum_{i=0}^{N=1} f\left(\frac{x_i+x_{i+1}}{2}\right)(x_{i+1}-x_i)\,.$$

By definition,

$$\mathcal{L}_N\le\mathcal{S}_N\le\mathcal{U}_N,$$

hence, by the sandwich theorem,

$$\lim_{N\to\infty}\mathcal{S}_N=I.$$

A numerical approximation to the Riemann integral I is thus

$$I\approx\sum_{i=0}^{N=1} f\left(\frac{x_i+x_{i+1}}{2}\right)(x_{i+1}-x_i)\,.$$

See Figure 13.2 for the accompanying sketch. We are interested in this section in the error involved

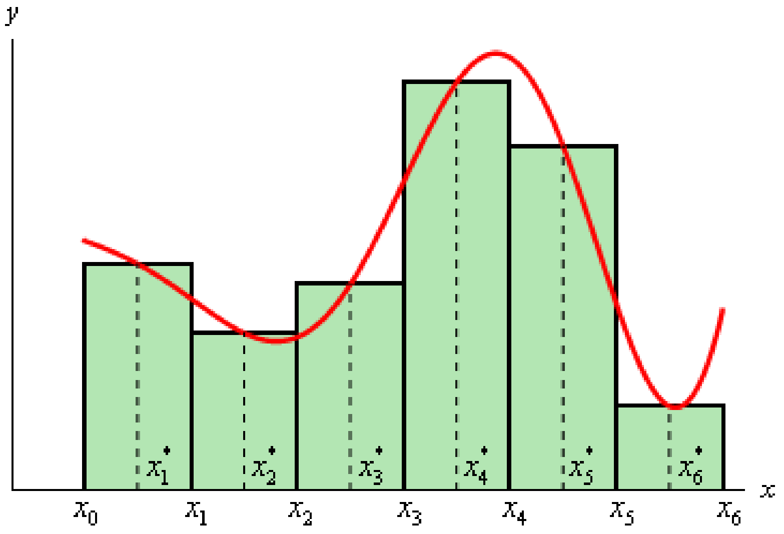


Figure 13.2: The midpoint rule

in using this approximation instead of the limit. We compute:

$$\begin{aligned}
\delta_N &:= \left| \int_a^b f(x)\,\mathrm{d}x - \mathcal{S}_N \right|, \\
&= \left| \int_a^b f(x)\,\mathrm{d}x - \sum_{i=0}^{N-1} f\left(\frac{x_i + x_{i+1}}{2}\right)(x_{i+1} - x_i) \right|, \\
&= \left| \int_a^b f(x)\,\mathrm{d}x - \Delta x \sum_{i=0}^{N-1} f\left(x_i + \tfrac{1}{2}\Delta x\right) \right|, \qquad \Delta x = \frac{b-a}{N}, \\
&= \left| \sum_{i=0}^{N-1} \int_{x_i}^{x_i+\Delta x} f(x)\,\mathrm{d}x - \Delta x \sum_{i=0}^{N-1} f\left(x_i + \tfrac{1}{2}\Delta x\right) \right|, \\
&= \left| \sum_{i=0}^{N-1} \int_{x_i}^{x_i+\Delta x} \left[f(x) - f(x_i + \tfrac{1}{2}\Delta x)\right] \mathrm{d}x \right|, \\
&= \left| \Delta x \sum_{i=0}^{N-1} \int_0^1 \left[f(x_i + \eta\Delta x) - f\left(x_i + \tfrac{1}{2}\Delta x\right)\right] \mathrm{d}\eta \right|.
\end{aligned}$$

We use Taylor's remainder theorem to write

$$f(x_i+\eta\Delta x) = f\left(x_i + \tfrac{1}{2}\Delta x\right) + f'(x_i+\tfrac{1}{2}\Delta x)\left(\eta - \tfrac{1}{2}\right)\Delta x + \tfrac{1}{2}f''(c_\eta)\left(\eta - \tfrac{1}{2}\right)^2 \Delta x^2, \qquad c_\eta \in [x_i, x_i + \Delta x].$$

Hence,

$$f(x_i + \eta) - f\left(x_i + \tfrac{1}{2}\Delta x\right) = f'(x_i + \tfrac{1}{2}\Delta x)\left(\eta - \tfrac{1}{2}\right)\Delta x + \tfrac{1}{2}f''(c_\eta)\left(\eta - \tfrac{1}{2}\right)^2 \Delta x^2.$$

But

$$\int_0^1 \left(\eta - \tfrac{1}{2}\right) \mathrm{d}\eta = 0,$$

hence

$$\begin{aligned}
\delta_N &= \Delta x^3 \left| \sum_{i=0}^{N-1} \int_0^1 \tfrac{1}{2} f''(c_\eta) \left(\eta - \tfrac{1}{2}\right)^2 \mathrm{d}\eta \right|, \\
&\leq \Delta x^3 \sum_{i=0}^{N-1} \left| \int_0^1 \tfrac{1}{2} f''(c_\eta) \left(\eta - \tfrac{1}{2}\right)^2 \mathrm{d}\eta \right|, \\
&\leq \tfrac{1}{2} \Delta x^3 \sum_{i=0}^{N-1} \int_0^1 |f''(c_\eta)| \left(\eta - \tfrac{1}{2}\right)^2 \mathrm{d}\eta, \\
&\leq \tfrac{1}{2} \Delta x^3 N \sup_{x \in [a,b]} |f''(x)| \int_0^1 \left(\eta - \tfrac{1}{2}\right)^2 \mathrm{d}\eta, \\
&\leq \tfrac{1}{2} \frac{(b-a)^3}{N^2} \left(\sup_{x \in [a,b]} |f''(x)| \right) \left(\tfrac{1}{12}\right),
\end{aligned}$$

Finally,

$$\delta_N = \frac{(b-a)^3}{24N^2} \left(\sup_{x \in [a,b]} |f''(x)| \right). \tag{13.3}$$

Of interest as well is the so-called **asymptotic error**, wheren the error is expressed in terms of a leading-order term in a Taylor expansion. As before, we start with

$$\begin{aligned}
\delta_N &:= \left| \int_a^b f(x)\, \mathrm{d}x - \mathcal{S}_N \right|, \\
&= \left| \Delta x \sum_{i=0}^{N-1} \int_0^1 \left[f(x_i + \eta) - f\left(x_i + \tfrac{1}{2}\Delta x\right) \right] \mathrm{d}\eta \right|.
\end{aligned}$$

We use Taylor's remainder theorem again to write

$$f(x_i + \eta) = f\left(x_i + \tfrac{1}{2}\Delta x\right) + f'(x_i + \tfrac{1}{2}\Delta x) \left(\eta - \tfrac{1}{2}\right) \Delta x + \tfrac{1}{2} f''(x_i + \tfrac{1}{2}\Delta x) \left(\eta - \tfrac{1}{2}\right)^2 \Delta x^2 + O(\Delta x^3),$$

where the omitted term can be written **exactly** as

$$\tfrac{1}{3!} f'''(d_\eta) \left(\eta - \tfrac{1}{2}\right)^3 \Delta x^3, \qquad d_\eta \in [x_i, x_i + \Delta x].$$

As before then,

$$\delta_N = \Delta x \left| \sum_{i=0}^{N-1} \int_0^1 \left[\tfrac{1}{2} f''(x_i + \tfrac{1}{2}\Delta x) \left(\eta - \tfrac{1}{2}\right)^2 \Delta x^2 + O(\Delta x^3) \right] \mathrm{d}\eta \right|,$$

$$\begin{aligned}
\delta_N &\le \Delta x\left[\sum_{i=0}^{N-1}\left|\int_0^1 \tfrac{1}{2}f''(x_i+\tfrac{1}{2}\Delta x)\left(\eta-\tfrac{1}{2}\right)^2\Delta x^2 \mathrm{d}\eta\right| + O(\Delta x^3)\right],\\
&\le \left[\tfrac{1}{2}\Delta x^3\sum_{i=0}^{N-1}\left|f''(x_i+\tfrac{1}{2}\Delta x)\right|\int_0^1\left(\eta-\tfrac{1}{2}\right)^2\mathrm{d}\eta\right] + N\Delta x\, O(\Delta x^3),\\
&\le \tfrac{1}{2}\Delta x^3 N\left(\sup_{x\in[a,b]}|f''(x)|\right)\left(\tfrac{1}{12}\right) + N\Delta x\, O(\Delta x^3),\\
&\le \tfrac{1}{24}\frac{(b-a)^3}{N^2}\left(\sup_{x\in[a,b]}|f''(x)|\right) + \left(\frac{b-a}{\Delta x}\right)\Delta x\, O(\Delta x^3).
\end{aligned}$$

since $N=(b-a)/\Delta x$. Using this fact again, we get

$$\delta_N = \tfrac{1}{24}\frac{(b-a)^3}{N^2}\left(\sup_{x\in[a,b]}|f''(x)|\right) + O\left(\frac{1}{N^3}\right). \tag{13.4}$$

Technical notes

- The exact form of the error estimate (Equation (13.3)) holds whenever the function to be integrated is C^2 on (a,b). This is a rather weak condition.

- On the other hand, the asymptotic error estimate (13.4) requires that $f(x)$ be at least C^3 on (a,b) – a slightly stronger condition.

- Both estimates will fail if the function is less smooth. However, the midpoint rule will still work. This is because the midpoint rule is a simple implementation of the Riemann-sum integration rule, whose success requires only that $f(x)$ be continuous on $[a,b]$.

- The basic theorems assumed here are listed for reference in Appendix A.

Exercise 13.1 *Write a Matlab code to integrate a continuous function $f:[a,b]\to\mathbb{R}$ using the Midpoint rule.*

13.3 The trapezoidal rule

The idea of the trapezoidal rule is similar in spirit to the midpoint rule. As before, the interval $[a, b]$ of integration is broken up into subintervals (the uniform partition), as

$$[a,b] = [x_0, x_1] \cup [x_1, x_2] \cup \cdots \cup [x_{N-1}, x_N], \qquad a = x_0, \qquad b = x_N,$$
$$x_i = a + i\left(\frac{b-a}{N}\right), \quad i = 0, 1, \cdots, N.$$

The area A_i under the curve in a typical subinterval (the i^{th}) is examined, and is approximated using a 'trapezoid', as in Figure 13.3 In other words, the curve is regarded as behaving 'almost' like a

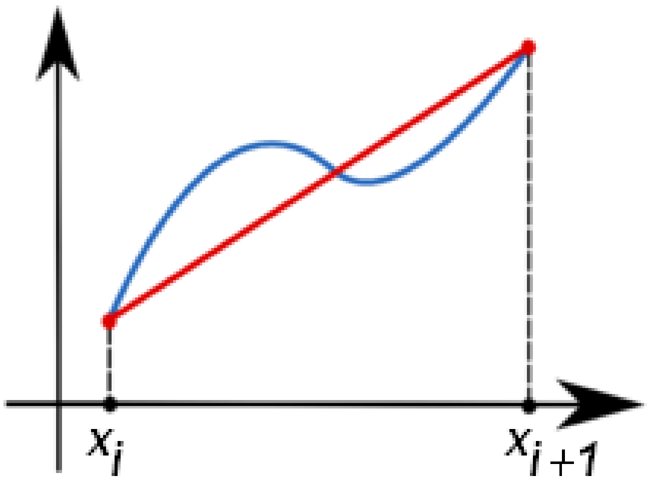


Figure 13.3: Trapezoidal rule for a subinterval

linear function,

$$f(x) \approx f_L(x) = f(x_i) + \left[\frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}\right](x - x_i),$$

hence

$$f(x) \approx f(x_i) + \left[\frac{f(x_{i+1}) - f(x_i)}{\Delta x}\right](x - x_i) \qquad \text{on } [x_i, x_i + \Delta x].$$

The approximate area can be computed exactly by integration:

$$A_i \approx \int_{x_i}^{x_i+\Delta x} f_L(x)\mathrm{d}x = \tfrac{1}{2}\Delta x\left[f(x_{i+1}) + f(x_i)\right].$$

Summing over all such typical intervals, the approximate area under the curve is

$$I \approx \tfrac{1}{2}\Delta x \sum_{i=0}^{N-1}\left[f(x_{i+1}) + f(x_i)\right].$$

Exercise 13.2 *Repeat the asymptotic error analysis of Section 13.3 and show that the error associated with the Trapezoidal Rule is*

$$\delta_N \leq \frac{(b-a)^3}{12N^2}\left(\sup_{x\in[a,b]}|f''(x)|\right) + O\left(\frac{1}{N^3}\right).$$

The Trapezoidal rule is therefore twice as poor as the easier-to-develop midpoint rule. This is an illustration of a general point:

Common Programming Error:

The addition of extra complexity does not necessarily improve an algorithm.

This loss-of-accuracy can be explained in loose terms as follows: the midpoint rule involves a function evaluation at the midpoint of an interval. For a typical continuous function, this will lead, on average, to an overestimate of the area half the time, and an underestimate for the other half of the time. Many of these errors will therefore cancel out, by symmetry, leading to a relatively small error. On the other hand, the trapezoidal rule does not possess this symmetry, leading to a larger error.

A further bizarre property of the midpoint rule is that the error is exactly zero for linear functions (wheren $f''(x) = 0$). However, looking at the method, we would expect it to be exact only for piecewise constant functions! The special symmetry inherent in the method gives us an extra layer of accuracy that we have no right to expect, *a priori*.

One advantage of the trapezoidal rule is that it belongs to a general class of methods called the **Newton–Coates formulas**, whose accuracy can be increased simply by increasing the so-called order of the formula. The trapezoidal rule is the first-order Newton–Coates formula, because it fits a line to a curve on each subinterval. At second order, a parabola is fitted to the curve on each subinterval. There are higher-order methods too. Increasing the order increases the accuracy of the approximation. We shall demonstrate this now with the second-order Newton–Coates method, also called **Simpson's rule**.

Chapter 14

Numerical Quadrature – Simpson's rule

Overview

In this chapter we use a more advanced method (Simpson's rule) to evaluate the definite integral

$$I = \int_a^b f(x)\mathrm{d}x,$$

where $f(x)$ is a continuous function.

14.1 Simpson's rule

The idea of Simpson's rule should by now be familiar. The interval $[a, b]$ of integration is broken up into subintervals (the uniform partition), and the area A_i under the curve in **two** neighbouring subintervals is examined, and is approximated using a parabola, as in Figure 14.1 In other words, the curve is regarded as behaving 'almost' like a quadratic function (parabola) which intersects the true curve at x_i, x_{i+1}, and x_{i+2} . We therefore write

$$f(x) \approx f_P(x) = A + Bx + Cx^2,$$

where the unknowns A, B, and C are determined via

$$f_P(x_i) = f(x_i), \;\; f_P(x_{i+1}) = f(x_{i+1}), \;\; f_P(x_{i+2}) = f(x_{i+2}).$$

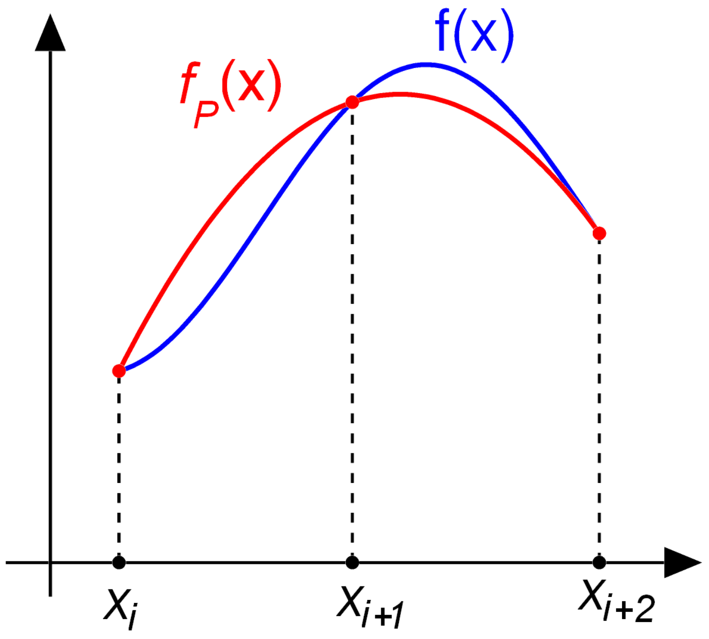


Figure 14.1: Simpson's rule for two neighbouring subintervals

In other words,

$$A + Bx_i + Cx_i^2 = f(x_i), \tag{14.1}$$
$$A + Bx_{i+1} + Cx_{i+1}^2 = f(x_{i+1}), \tag{14.2}$$
$$A + Bx_{i+2} + Cx_{i+2}^2 = f(x_{i+2}). \tag{14.3}$$

These are three linearly-independent equations in three unknowns, A, B, and C. We take (14.2)−(14.1) and (14.3)−(14.2) to obtain

$$B(x_{i+1} - x_i) + C(x_{i+1}^2 - x_i^2) = f(x_{i+1}) - f(x_i),$$
$$B(x_{i+2} - x_{i+1}) + C(x_{i+2}^2 - x_{i+1}^2) = f(x_{i+2}) - f(x_{i+1}).$$

In other words,

$$B\Delta x + C\left(2x_i\Delta x + \Delta x^2\right) = f(x_{i+1}) - f(x_i), \tag{14.4}$$
$$B\Delta x + C\left(2x_i\Delta x + 3\Delta x^2\right) = f(x_{i+2}) - f(x_{i+1}). \tag{14.5}$$

We subtract these equations ((14.5)−(14.4)) to obtain

$$C\left(2\Delta x^2\right) = f(x_{i+2}) - 2f(x_{i+1}) + f(x_i),$$

hence

$$C = \frac{1}{2\Delta x^2}\left[f(x_{i+2}) - 2f(x_{i+1}) + f(x_i)\right]. \tag{14.6}$$

Back-substitution into Equation (14.4) yields

$$B\Delta x = \left[f(x_{i+1}) - f(x_i)\right] - \Delta x\left(2x_i + \Delta x\right)C.$$

Final substitution into Equation (14.1) yields

$$A = f(x_i) - Bx_i - Cx_i^2.$$

The approximate area can be computed exactly by integration:

$$\begin{aligned}
A_i \approx \int_{x_i}^{x_i+2\Delta x} f_P(x)\,\mathrm{d}x &= 2A\Delta x + \tfrac{1}{2}B\left[(x_i+2\Delta x)^2 - x_i^2\right] + \tfrac{1}{3}C\left[(x_i+2\Delta x)^3 - x_i^3\right], \\
&= 2A\Delta x + 2B\Delta x\left(x_i + \Delta x\right) + \tfrac{2}{3}C\Delta x\left(3x_i^2 + 3x_i 2\Delta x + 4\Delta x^2\right) \\
&= 2A\Delta x + 2B\Delta x\left(x_i + \Delta x\right) + 2C\Delta x\left(x_i^2 + x_i 2\Delta x + \tfrac{4}{3}\Delta x^2\right), \\
&= 2\Delta x\left[f(x_i) - Bx_i - Cx_i^2\right] + 2B\Delta x\left(x_i + \Delta x\right) + 2C\Delta x\left(x_i^2 + 2x_i\Delta x + \tfrac{4}{3}\Delta x^2\right).
\end{aligned}$$

Some cancellation occurs, giving

$$A_i \approx 2\Delta x f(x_i) + 2B\Delta x^2 + 2C\Delta x\left(2x_i\Delta x + \tfrac{4}{3}\Delta x^2\right).$$

Use $\Delta x B = \left[f(x_{i+1}) - f(x_i)\right] - \Delta x\left(2x_i + \Delta x\right)C$ again to obtain

$$\begin{aligned}
A_i &\approx 2\Delta x f(x_i) + 2\Delta x\left\{\left[f(x_{i+1}) - f(x_i)\right] - \Delta x\left(2x_i + \Delta x\right)C\right\} + 2C\Delta x\left(2x_i\Delta x + \tfrac{4}{3}\Delta x^2\right), \\
&= 2\Delta x f(x_i) + 2\Delta x\left[f(x_{i+1}) - f(x_i)\right] - 4\Delta x^2 x_i C - 2\Delta x^3 C + 4\Delta x^2 x_i C + \tfrac{8}{3}\Delta x^3 C, \\
&= 2\Delta x f(x_{i+1}) + \tfrac{2}{3}\Delta x^3 C.
\end{aligned}$$

Using the definition of C in Equation (14.6), this is

$$\begin{aligned}
A_i &\approx 2\Delta x f(x_{i+1}) + \tfrac{2}{3}\Delta x^3\left\{\frac{1}{2\Delta x^2}\left[f(x_{i+2}) - 2f(x_{i+1}) + f(x_i)\right]\right\}, \\
&= \tfrac{1}{3}\Delta x\left[f(x_{i+2}) + f(x_i) + 4f(x_{i+1})\right].
\end{aligned}$$

We sum over all such **pairs** of intervals, such that

$$\begin{aligned}
I \approx{}& \tfrac{1}{3}\Delta x\left[f(x_2) + f(x_0) + 4f(x_1)\right] + \tfrac{1}{3}\Delta x\left[f(x_4) + f(x_2) + 4f(x_3)\right] + \\
& \tfrac{1}{3}\Delta x\left[f(x_6) + f(x_4) + 4f(x_5)\right] + + \cdots \tfrac{1}{3}\Delta x\left[f(x_N) + f(x_{N-2}) + 4f(x_{N-1})\right].
\end{aligned}$$

This can be re-arranged so the pattern becomes clearer:

$$I \approx \tfrac{1}{3}\Delta x\left[f(x_0)+f(x_N)\right]+\tfrac{2}{3}\Delta x\left[f(x_2)+f(x_4)+f(x_6)+\cdots+f(x_{N-2})\right]+$$
$$\tfrac{4}{3}\Delta x\left[f(x_1)+f(x_3)+f(x_5)+\cdots+f(x_{N-1})\right].$$

Because we consider the partition to be made up of pairs of subintervals, N must be even.

Common Programming Error:

Not taking N to be even in an implementation of Simpson's rule.

We may therefore re-write the approximate formula as follows:

$$I \approx \tfrac{1}{3}\Delta x\left[f(x_0)+f(x_N)+2\sum_{\substack{1\le j<N\\ j\text{ even}}} f(x_j)+4\sum_{\substack{1\le j<N\\ j\text{ odd}}} f(x_j)\right]. \tag{14.7}$$

Exercise 14.1 *Write a Matlab code to integrate a continuous function $f:[a,b]\to\mathbb{R}$ using Simpson's rule.*

Common Programming Error:

The x-index in the Simpson formula (14.7) starts with $j=0$. Array indices in Matlab start at 1. A code to implement Simpson's rule has to recognize this shift.

14.2 Error analysis for Simpson's rule

We perform an asymptotic error analysis. We start with the error associated with integration over a typical subinterval pair:

$$\delta_i=\left|\int_{x_i}^{x_i+2\Delta x} f(x)\mathrm{d}x-\tfrac{1}{3}\Delta x\left[f(x_{i+2})+f(x_i)+4f(x_{i+1})\right]\right|.$$

We Taylor-expand to **fifth order**, with Taylor series centred at x_i. For simplicity, we call the signed error ϵ_i, with $\delta_i = |\epsilon_i|$. We have,

$$\begin{aligned}
&\epsilon_i = \\
&\int_{x_i}^{x_i+2\Delta x} \big[f(x_i)+f'(x_i)(x-x_i)+\tfrac{1}{2}f''(x_i)(x-x_i)^2+\tfrac{1}{3!}f'''(x_i)(x-x_i)^3+\tfrac{1}{4!}f^{(4)}(x-x_i)^4+\tfrac{1}{5!}f^{(5)}(c_x)(x-x_i)^5\big]\mathrm{d}x \\
&\quad -\tfrac{1}{3}\Delta x\big[f(x_i)+f'(x_i)2\Delta x+\tfrac{1}{2}f''(x_i)4\Delta x^2+\tfrac{1}{3!}f'''(x_i)8\Delta x^3+\tfrac{1}{4!}f^{(4)}16\Delta x^4+\tfrac{1}{5!}f^{(5)}(c_0)32\Delta x^5\big] \\
&\qquad -\tfrac{1}{3}\Delta x f(x_i) \\
&\qquad -\tfrac{4}{3}\Delta x\big[f(x_i)+f'(x_i)\Delta x+\tfrac{1}{2}f''(x_i)\Delta x^2+\tfrac{1}{3!}f'''(x_i)\Delta x^3+\tfrac{1}{4!}f^{(4)}\Delta x^4+\tfrac{1}{5!}f^{(5)}(c_1)\Delta x^5\big].
\end{aligned}$$

We compute the contributions, order-by-order in the Taylor series:

Zeroth order: We have

$$\int_{x_i}^{x_i+2\Delta x} f(x_i)\mathrm{d}x - \Delta x\left(\tfrac{1}{3}+\tfrac{1}{3}+\tfrac{4}{3}\right)f(x_i) = 0.$$

First order: We have

$$f'(x_i)\int_{x_i}^{x_i+2\Delta x}(x-x_i)\,\mathrm{d}x - \Delta x^2 f'(x_i)\left(\tfrac{2}{3}+\tfrac{4}{3}\right) = f'(x_i)\left[\tfrac{1}{2}(x-x_i)^2\Big|_{x_i}^{x_i+2\Delta x} - 2\Delta x^2\right] = 0.$$

Second order: We have

$$\tfrac{1}{2}f''(x_i)\int_{x_i}^{x_i+2\Delta x}(x-x_i)^2\,\mathrm{d}x - \tfrac{1}{2}\Delta x^3 f''(x_i)\left(\tfrac{4}{3}+\tfrac{4}{3}\right) = \tfrac{1}{2}f''(x_i)\left[\tfrac{1}{3}(x-x_i)^3\Big|_{x_i}^{x_i+2\Delta x} - \tfrac{8}{3}\Delta x^3\right] = 0.$$

Third order: We have

$$\tfrac{1}{3!}f'''(x_i)\int_{x_i}^{x_i+2\Delta x}(x-x_i)^3\,\mathrm{d}x - \tfrac{1}{3!}\Delta x^4 f'''(x_i)\left(\tfrac{8}{3}+\tfrac{4}{3}\right) = \tfrac{1}{3!}f'''(x_i)\left[\tfrac{1}{4}(x-x_i)^4\Big|_{x_i}^{x_i+2\Delta x} - \tfrac{12}{3}\Delta x^4\right] = 0.$$

Fourth order: We have

$$\begin{aligned}
&\tfrac{1}{4!}f^{(4)}(x_i)\int_{x_i}^{x_i+2\Delta x}(x-x_i)^4\,\mathrm{d}x - \tfrac{1}{4!}\Delta x^5 f^{(4)}(x_i)\left(\tfrac{16}{3}+\tfrac{4}{3}\right) \\
&\qquad = \tfrac{1}{4!}f^{(4)}(x_i)\left[\tfrac{1}{5}(x-x_i)^5\Big|_{x_i}^{x_i+2\Delta x} - \tfrac{20}{3}\Delta x^4\right] = -\tfrac{4}{15\cdot 24}\Delta x^5 f^{(4)}(x_i) = -\tfrac{1}{90}\Delta x^5 f^{(4)}(x_i).
\end{aligned}$$

Remainder terms: We have

$$\tfrac{1}{5!}\int_{x_i}^{x_i+2\Delta x} f^{(5)}(c_x)(x-x_i)^5\mathrm{d}x - \tfrac{1}{5!}\Delta x^6\left[\tfrac{64}{3}f^{(5)}(c_0)+\tfrac{4}{3}f^{(5)}(c_0)\right].$$

Let's put it all together now:

$$\begin{aligned}
\epsilon_i &= -\tfrac{1}{90}\Delta x^5 f^{(4)}(x_i) + \tfrac{1}{5!}\int_{x_i}^{x_i+2\Delta x} f^{(5)}(c_x)(x-x_i)^5\mathrm{d}x - \tfrac{1}{5!}\Delta x^6\left[\tfrac{64}{3}f^{(5)}(c_0)+\tfrac{4}{3}f^{(5)}(c_0)\right],\\
\delta_i &= \left|-\tfrac{1}{90}\Delta x^5 f^{(4)}(x_i) + \tfrac{1}{5!}\int_{x_i}^{x_i+2\Delta x} f^{(5)}(c_x)(x-x_i)^5\mathrm{d}x - \tfrac{1}{5!}\Delta x^6\left[\tfrac{64}{3}f^{(5)}(c_0)+\tfrac{4}{3}f^{(5)}(c_0)\right]\right|,\\
\delta_i &\le \tfrac{1}{90}\Delta x^5|f^{(4)}(x_i)| + \left|\tfrac{1}{5!}\int_{x_i}^{x_i+2\Delta x} f^{(5)}(c_x)(x-x_i)^5\mathrm{d}x - \tfrac{1}{5!}\Delta x^6\left[\tfrac{64}{3}f^{(5)}(c_0)+\tfrac{4}{3}f^{(5)}(c_0)\right]\right|,\\
&\le \tfrac{1}{90}\Delta x^5|f^{(4)}(x_i)| + \tfrac{1}{5!}\left(\sup_{[x_i,x_i+2\Delta x]}|f^{(5)}(x)|\right)\int_{x_i}^{x_i+2\Delta x}(x-x_i)^5\,\mathrm{d}x\\
&\qquad + \Delta x^6\left(\sup_{[x_i,x_i+2\Delta x]}|f^{(5)}(x)|\right)\left(\tfrac{64}{3}+\tfrac{4}{3}\right).
\end{aligned}$$

Hence,

$$\delta_i \le \tfrac{1}{90}\Delta x^5|f^{(4)}(x_i)| + O(\Delta x^6). \tag{14.8}$$

We sum over all interval pairs as follows:

$$\delta_N = \left|\sum_{\substack{j\ge 0\\ \text{interval pairs}}}\epsilon_j\right| \le \sum_{\substack{j\ge 0\\ \text{interval pairs}}}|\epsilon_j| = \sum_{\substack{j\ge 0\\ \text{interval pairs}}}\delta_j.$$

From Equation (14.8)

$$\begin{aligned}
\delta_N &\le \sum_{\substack{j\ge 0\\ \text{interval pairs}}}\left[\tfrac{1}{90}\Delta x^5|f^{(4)}(x_i)| + O(\Delta x^6)\right],\\
&\le \tfrac{1}{2}N\tfrac{1}{90}\Delta x^5\left(\sup_{[a,b]}|f^{(4)}(x)|\right) + \tfrac{1}{2}N\,O(\Delta x^6),
\end{aligned}$$

Finally,

$$\delta_N \le \tfrac{1}{180}\left(\frac{(b-a)^5}{N^4}\right)\left(\sup_{[a,b]}|f^{(4)}(x)|\right) + O\left(\frac{1}{N^5}\right).$$

Thus, **the error in Simpson's rule decreases as** N^{-4}. Contast this with the Trapezoidal and midpoint rules, where the error decreased as N^{-2}.

14.3 Quadrature with Matlab's built-in functions

Matlab has a built-in quadrature function that implements Simpson's rule. Technically, it uses an **adaptive** Simpson's rule, which means that the partition is non-uniform: the partition is refined locally so that the error never exceeds a prescribed tolerance. A first implementation is shown below, where the default tolerance is assumed.

```
function [Q]=do_num_quad(a,b)

% Function "myfun" to be integrated; this is defined in a Matlab
% subfunction below.

Q = quad(@myfun,a,b);

function y = myfun(x)
    % User-defined function - put whatever you like here, but watch
    % out for ./ and .*
    y = 1./(1+sin(x).*sin(x));
end

end
```

sample_matlab_codes/do_num_quad.m

On the other hand, the following version shows a user-defined tolerance. This is a limit on the **absolute error** of the total integral I.

```
function [Q]=do_num_quad_withtol(a,b)

% Function "myfun" to be integrated; this is defined in a Matlab
% subfunction below.

% User-defined tolerance (default is 1e-6)
tol=1e-8;

Q = quad(@myfun,a,b,tol);

function y = myfun(x)
    % User-defined function - put whatever you like here, but watch
    % out for ./ and .*
    y = 1./(1+sin(x).*sin(x));
end

end
```

sample_matlab_codes/do_num_quad_withtol.m

Finally, from the Matlab help pages:

- The quadl function may be more efficient than quad at higher accuracies with smooth integrands.
- The quad function may be most efficient for low accuracies with nonsmooth integrands.

Here 'quadl' is a built-in function that implements **Gauss–Lobtatto** quadrature. A discussion of this method is beyond the scope of this module. However, it can easily be implemented in Matlab simply by replacing the command 'quad' with 'quadl'. Since we have already spent a great deal of time in studying numerical quadrature, we should not feel too guilty in resorting to a black box in this instance.

Chapter 15

Ordinary Differential Equations – Euler's method

Overview

We examine the model ordinary-differential equation (ODE)

$$\frac{dx}{dt} = F(x,t), \qquad x(0) = x_0.$$

We investigate the conditions that guarantee that the ODE has a solution ('existence theorem'). We examine a numerical method to approximate the solution (if it exists), reducing the problem to the numerical calculation of a simple difference equtaion. This is the so-called **Euler method**.

15.1 The definition

An n-dimensional ordinary-differential equation (ODE) is a relation of the following form:

$$\frac{d\boldsymbol{x}}{dt} = \boldsymbol{F}(\boldsymbol{x},t), \tag{15.1a}$$

where $\boldsymbol{x} = (x_1, ..., x_n)^T$ are variables that depend on t and $\boldsymbol{F}$ is a function of the form

$$\boldsymbol{F}(\boldsymbol{x},t) = \begin{pmatrix} F_1(x_1, ..., x_n, t) \\ F_2(x_1, ..., x_n, t) \\ \vdots \\ F_1(x_1, ..., x_n, t) \end{pmatrix} \tag{15.1b}$$

The equation can be solved either as a **boundary-value problem** or as an **initial-value problem** (IVP). In this module, we are interested in IVPs. Equation (15.1a) is solved as an IVP if the following extra condition is given:

$$\boldsymbol{x}(t=0)=\boldsymbol{x}_0, \tag{15.1c}$$

where $\boldsymbol{x}_0$ is some constant vector called the **initial condition**. .

Furthermore, an IVP is called **autonomous** if the function $\boldsymbol{F}$ does not depend **explicitly** on t, $\boldsymbol{F}=\boldsymbol{F}(\boldsymbol{x})$ only.

15.2 Examples

The simplest possible ODE is a one-dimensional autonomouos ODE:

$$\frac{dx}{dt}=f(x).$$

We solve this by separation of variables:

$$\int\frac{dx}{f(x)}=t+C,$$

where C is a contant of integration. More concretely, consider the example

$$\frac{dx}{dt}=ax,$$

where a is a constant real number. We separate the variables:

$$\frac{\mathrm{d}x}{x}=a\,\mathrm{d}t.$$

Integrate:

$$\int\frac{\mathrm{d}x}{x}=a\int\mathrm{d}t+C.$$

Hence,

$$\log x=at+C.$$

Exponentiate both sides ('taking inverses'):

$$x=\mathrm{e}^C\mathrm{e}^{at}.$$

Let $D=\mathrm{e}^C$ (a constant):

$$x=D\mathrm{e}^{at}.$$

The constant D is fixed by initial conditions:

$$x(t=0)=x_0 \implies D=x_0.$$

Finally, the solution to the IVP is

$$x=x_0\mathrm{e}^{at}.$$

So far our discussion of ODEs has not involved the notion of 'order'. This is for a good reason. The order of a differential equation refers to how many derivatives appear. For example, an ODE involving d/dt only is called first order, while an ODE involving d^2/dt^2 and d/dt only is called second order, and so on. However, a single second-order ODE can always be converted into a system of first-order ODEs. For example, consider

$$m\frac{d^2x}{dt^2}=f(x,t),$$

where m is a constant. Calling $v(t)=dx/dt$, we can re-write this as

$$\frac{d}{dt}\begin{pmatrix} x \\ v \end{pmatrix}=\begin{pmatrix} v \\ \frac{1}{m}f(x,t) \end{pmatrix}.$$

Thus, the notion of 'order' is really subsumed into the notion of a system of ODEs.

Concetely, consider the following physical example:

$$m\frac{d^2x}{dt^2}+\gamma m\frac{dx}{dt}+m\omega_0^2x^2=f_0\cos(\omega t), \tag{15.2}$$

which is the equation of a damped, driven (linear) pendulum. Here, m, γ, ω_0, f_0, and ω are all positive constants. The force on the pendlum is

$$f(x,t)=-\gamma m\frac{dx}{dt}-m\omega_0^2+f_0\cos(\omega t),$$

and the equation of motion (15.2) can be re-written as

$$m\frac{d^2x}{dt^2}=f(x,t).$$

Viewed as a system of first-order equations, this is

$$\frac{d}{dt}\begin{pmatrix} x \\ v \end{pmatrix}=\begin{pmatrix} v \\ -\gamma v-\omega_0^2x+(f_0/m)\cos(\omega t) \end{pmatrix}.$$

To solve these equations, the following initial data are needed:

$$x(t=0)=x_0, \qquad v(t=0)=v_0,$$

where x_0 and v_0 are constants corresponding to the initial position and velocity of the pendulum.

15.3 Existence of solutions

An IVP is not guaranteed to have a solution at all. Furthermore, a solution can exist for a finite time and then become infinite ('blow up'). We therefore need an **existence theorem** which tells us when an ODE has a solution. For definiteness, we consider the one-dimensional case.

Theorem 15.1 *Let $F : [0,T] \times \mathbb{R} \to \mathbb{R}$ be a real function of two variables. Assume that F is continuous and that there exists a real number K with*

$$|F(t,x_2)-F(t,x_1)| \le K|x_2-x_1|, \tag{15.3}$$

for all $t \in [0,T]$, and for all $x_1, x_2 \in \mathbb{R}$. Then the ODE

$$\frac{dx}{dt}=F(x,t) \tag{15.4}$$

with initial condition

$$x(t=0)=x_0$$

has a unique solution in some finite interval $[0,T_0]$, with $0<T_0\le T$.

Notes:

- The proof of this theorem requires advanced topics and is not covered in this module.
- Even so, it is a relatively weak statement: it is guaranteed that a unique solution exists only in some interval $[0,T_0]$ – the solution does not even necessarily exist out to $t=T$. For this reason, the theorem is called a **local-existence theorem**.
- Also, the condition (15.3) is rather restrictive: this is a stronger condition than simple continuity.
- Finally, note the precise wording of the theorem: the K-number ('Lipschitz constant') has to be independent of x!

Examples

1. The equation $dx/dt = ax$ has a unique (local) solution because

$$|f(x_2) - f(x_1)| = |ax_2 - ax_1| = a\,|x_2 - x_1|\,,$$

hence a is the Lipschitz constant. Also, we know that the solution $x(t) = x_0 \mathrm{e}^{at}$ is smooth and globally defined.

2. Consider

$$\frac{dx}{dt} = x^3, \qquad x(0) = x_0 > 0$$

Solving this using separation of variables, we get

$$-\tfrac{1}{2}\frac{1}{x^2}\bigg|_{x_0}^{x} = t,$$

hence

$$x = \operatorname{sign}(x_0)\left(\frac{1}{\frac{1}{x_0^2} - 2t}\right)^{1/2},$$

and the solution ceases to exist at $t = 1/(2x_0^2)$ (‘blow-up’).

Local existence theory fails here because

$$\begin{aligned}
|F(x_2) - F(x_1)| &= \left|x_2^3 - x_1^3\right|,\\
&= \left|(x_2 - x_1)\left(x_2^2 + x_1x_2 + x_1^2\right)\right|,\\
&= \left|x_2^2 + x_1x_2 + x_1^2\right|\,|x_2 - x_1|\,,\\
&:= K(x_1, x_2)|\,|x_2 - x_1|\,.
\end{aligned}$$

It is as if the Lifshitz ‘constant’ depends on x_1 and x_2. Although Theorem (15.1) can be modified to include such case, it is not surprising that blowup occurs here.

3. Consider

$$\frac{dx}{dt} = \sqrt{|x|}, \qquad x(0) = 0.$$

Solving this using separation of variables, we get $x(t) = (1/4)t^2$. However, $x(t) = 0$ is also a good solution. In other words, the ODE fails to have a unique solution.

We can inspect what has gone wrong by consideration of the following. We take $x_2, x_1 \geq 0$.

Then, the difference $|F(x_2) - F(x_1)|$ can be written as follows:

$$\begin{aligned} |F(x_2) - F(x_1)| &= |x_2 - x_1| \left| \frac{1}{\sqrt{x_2}} - \frac{1}{\sqrt{x_2 x_1}} \frac{x_2 + \sqrt{x_2 x_1} + x_1}{\sqrt{x_2} + \sqrt{x_1}} + \frac{1}{\sqrt{x_1}} \right|, \\ &:= K(x_2, x_1) \left| x_2 - x_1 \right|. \end{aligned} \tag{15.5}$$

However, $K(x_2 = 0, x_1 = 0) = \infty$, and the Lifshitz coefficient does not exist – even when allowing for it to vary. For this reason, the local existence theory in Theorem (15.1) fails even at $t = 0$, leading to two non-unique solutions emanating from $t = 0$.

Exercise 15.1 *Prove Equation* (15.5).

These examples serve up a very cautionary tale:

Common Programming Error:

Just because you know some fancy numerical method to solve $dx/dt = F(x, t)$, it does not mean that the method will work – you need to understand the properties of the F-function.

15.4 Euler's method for one-dimensional systems – the idea

Suppose now that our F-function is well-behaved, and possesses as many partial derivatives with respect to x as we see fit. We are interested in solving

$$\frac{dx}{dt} = F(x, t), \qquad x(0) = x_0, \qquad t \in [0, T]. \tag{15.6}$$

We discretize the time variable, by taking

$$t \in \{t_0, t_1, \cdots t_N\},$$

where

$$t_i = i \left(\frac{T}{N} \right) := i \Delta t, \qquad i = 1, 2, \cdots, N$$

(we call Δt the **timestep**). We consider the variable x sampled at the discrete time points:

$$x_i := x(t_i).$$

We integrate $dx/dt = F(x,t)$ with respect to time over a **single timestep**. This yields the following **exact** result:

$$x_{i+1} = x_i + \Delta t \int_{t_i}^{t_{i+1}} F(x(t),t)\,\mathrm{d}t. \tag{15.7}$$

Now here is the magic: we approximate the integral in Equation (15.7) by the Riemann sum $\Delta t F(x_n, t_n)$. Hence, an approximation of Equation (15.7) is

$$x_{i+1} = x_i + \Delta t F(x_i, t_i), \qquad x_{i=0} = x_0. \tag{15.8}$$

This is a simple iterative map that can be easily implemented on a computer. In the next chapter we put this method through its paces by examining the truncation error associated with the step (15.8).

Chapter 16

Euler's method – Accuracy and Stability

Overview

In the last chapter we approximated solutions of the ODE

$$\frac{dx}{dt} = F(x,t), \qquad x(0) = x_0, \tag{16.1}$$

by the discrete-time iterative equation

$$x_{i+1} = x_i + \Delta t F(x_i, t_i), \qquad x_{i=0} = x_0,$$

where Δt is the timestep. We now examine the error associated with this approach. More importantly, we investigate whether this method is **stable**.

16.1 Local truncation errors

Theorem 16.1 *The true solution of Equation* (16.1) *can always be written formally (and implicitly) as an anti-derivative:*

$$X(t) = X_0 + \int_0^t F(X(t'), t')\mathrm{d}t', \tag{16.2}$$

Proof:

$$\begin{aligned}
\frac{d}{dt}\,(\text{R.H.S.}) &= \frac{d}{dt}\int_0^t F(X(t'),t')\mathrm{d}t', \\
&= F(X(t),t), \\
&\overset{\text{ODE}}{=} \frac{dX}{dt}, \\
&= \frac{d}{dt}\,(\text{L.H.S}) \,.
\end{aligned}$$

Now, the result (16.2) can also be written as

$$X(t_{n+1}) = X(t_n) + \int_{t_n}^{t_{n+1}} F(X(t),t)\mathrm{d}t.$$

Compare this with the Euler method:

$$x_{n+1} = x_n + \Delta t F(x_n, t_n).$$

It is as if we have approximated the integral

$$\int_{t_n}^{t_{n+1}} F(X(t),t)\mathrm{d}t \tag{16.3}$$

with the Riemann sum $F(x_n,t_n)\Delta t$! We now investigate the error incurred in this approximation – the so-called **local truncation error**.

Let $X(t)$ denote the true solution. We have

$$X(t_{n+1}) = X(t_n) + \int_{t_n}^{t_{n+1}} F(X(t),t)\mathrm{d}t.$$

Evaluate the integral by doing a Taylor-series expansion of $F(\cdot,\cdot)$ around t_n:

$$\begin{aligned} X(t_{n+1}) &= X(t_n) + \int_{t_n}^{t_n+\Delta t}\left[F(X_n,t_n) + \left(\frac{\partial F}{\partial x}\frac{dX}{dt} + \frac{\partial F}{\partial t}\right)_{(X_n,t_n)}(t-t_n) + \cdots\right]\mathrm{d}t, \\ &= X(t_n) + F(X_n,t_n)\Delta t + \tfrac{1}{2}\left(\frac{\partial F}{\partial x}\frac{dX}{dt} + \frac{\partial F}{\partial t}\right)_{(X_n,t_n)}\Delta t^2 + \cdots, \\ &= X(t_n) + F(X_n,t_n)\Delta t + \tfrac{1}{2}\left(\frac{\partial F}{\partial x}F(x,t) + \frac{\partial F}{\partial t}\right)_{(X_n,t_n)}\Delta t^2 + \cdots, \end{aligned}$$

Compare the exact solution with the Euler-iterated solution:

$$x_{n+1} = x_n + \Delta t F(x_n,t_n), \tag{16.4a}$$

$$X(t_{n+1}) = X(t_n) + \Delta t F(X_n,t_n) + \tfrac{1}{2}\left(\frac{\partial F}{\partial x}F(x,t) + \frac{\partial F}{\partial t}\right)_{(X_n,t_n)}\Delta t^2 + O(\Delta t^3). \tag{16.4b}$$

Again, we define the **local truncation error** as the error incurred in approximating the integral (16.3) by a finite one-point sum. **The local truncation error does not therefore depend on errors incurred at previous timesteps.** For that reason, we examine Equations (16.4) again and identify the local truncation error δ_n^{LTE} as

$$\begin{aligned}
\delta_n^{\mathrm{LTE}} &:= \left|\tfrac{1}{2}\left(\frac{\partial F}{\partial x}F(x,t)+\frac{\partial F}{\partial t}\right)_{(X_n,t_n)}\Delta t^2+O(\Delta t^3)\right|, \\
&\le \tfrac{1}{2}\left(\sup_{\substack{-\infty<x<\infty\\ t>0}}\left|\frac{\partial F}{\partial x}F(x,t)+\frac{\partial F}{\partial t}\right|\right)\Delta t^2+O(\Delta t^3), \\
&:= \tfrac{1}{2}M\Delta t^2+O(\Delta t^3).
\end{aligned}$$

We sum up all the local truncation errors:

$$\delta_N^{\mathrm{LTE}} \le \sum_{i=1}^{N}\delta_i^{\mathrm{LTE}}$$

(triangle inequality), hence

$$\delta_N^{\mathrm{LTE}} \le N\left[\tfrac{1}{2}M\Delta t^2+O(\Delta t^3)\right]=\left(\frac{T}{\Delta t}\right)\left[\tfrac{1}{2}M\Delta t^2+O(\Delta t^3)\right],$$

or

$$\delta_N^{\mathrm{LTE}} \le \tfrac{1}{2}TM\Delta t+O(\Delta t^2).$$

The **truncation error** is thus linear in the stepsize Δt. This is a rather poor result. However, by increasing the computational effort, this result does guarantee that the truncation error can be reduced (Herculean though the task may be). For this reason, the Euler method is called **first-order accurate** with respect to truncation errors.

More worrying is the following: analysis of the truncation error does not yield any insight into how the error is compounded at each timestep. A truncation error refers to **accuracy** – how well does the approximation

$$\Delta t F(x_n,t_n)\approx\int_{t_n}^{t_{n+1}}F(x(t),t)\,\mathrm{d}t$$

represent the true integral in the formula

$$x_{n+1}=x_n+\int_{t_n}^{t_{n+1}}F(x(t),t)\,\mathrm{d}t.$$

On the other hand, studying how errors are compounded is a matter of **stability** – do the errors accumulate to such an extent that the numerical method becomes unreliable? Accuracy and stability are in fact two completely independent matters. We consider the latter in more detail now.

16.2 Stability analysis

Theorem 16.2 *The Euler method with stepsize* Δt *applied to Equation* (16.1) *is* **numerically stable** *if the following condition holds:*

$$-2 \leq \Delta t F_x(x,t) < 0, \;\; \forall x \in \mathbb{R}, t > 0.$$

Proof: Let $X_n = X(t_n)$ denote the **true** analytic, solution of the equation (16.1), let x_n denote the Euler time-integrated solution, and let

$$\epsilon_n = X_n - x_n$$

be the difference between these two quantities. From Section 16.1 and Equations (16.4) we have

$$\begin{aligned} x_{n+1} &= x_n + \Delta t F(x_n, t_n), \\ X(t_{n+1}) &= X(t_n) + \Delta t F(X_n, t_n) + \left[\frac{\partial F}{\partial x} F(x,t) + \frac{\partial F}{\partial t}\right]_{(X_n,t_n)} \Delta t^2 + O(\Delta t^3). \end{aligned}$$

In other words,

$$\begin{aligned} x_{n+1} &= x_n + \Delta t F(x_n, t_n), \\ X(t_{n+1}) &= X(t_n) + \Delta t F(x_n + \epsilon_n, t_n) + \tfrac{1}{2}\left[\frac{\partial F}{\partial x} F(x,t) + \frac{\partial F}{\partial t}\right]_{(x_n+\epsilon_n,t_n)} \Delta t^2 + O(\Delta t^3). \end{aligned}$$

We subtract to obtain

$$\begin{aligned} \epsilon_{n+1} &= \epsilon_n + \Delta t \left[F(x_n+\epsilon_n, t_n) - F(x_n,t_n)\right] + \tfrac{1}{2}\left[\frac{\partial F}{\partial x} F(x,t) + \frac{\partial F}{\partial t}\right]_{(x_n+\epsilon_n,t_n)} \Delta t^2 + O(\Delta t^3), \\ &= \epsilon_n + \Delta t \left.\frac{\partial F}{\partial x}\right|_{(x_n,t_n)} \epsilon_n \\ &\quad + \tfrac{1}{2}\Delta t^2 \Bigg\{ \left[\frac{\partial F}{\partial x} F(x,t) + \frac{\partial F}{\partial t}\right]_{(x_n,t_n)} + \left[\frac{\partial^2 F}{\partial x^2} F(x,t) + \frac{\partial^2 F}{\partial x \partial t}\right]_{(x_n,t_n)} \epsilon_n + O(\epsilon_n^2) \Bigg\} + O(\Delta t^3), \\ &= \epsilon_n \left(1 + \Delta t \left.\frac{\partial F}{\partial x}\right|_{(x_n,t_n)}\right) + O(\Delta t^2, \epsilon_n \Delta t^2), \\ &:= \epsilon_n \left[1 + \Delta t F_x(x_n,t_n)\right] + O(\Delta t^2, \epsilon_n \Delta t^2). \end{aligned}$$

From now on we omit the higher-order terms without loss of generality and write

$$\epsilon_{n+1} = \epsilon_n \left[1 + \Delta t F_x(x_n, t_n)\right].$$

We telescope this result as follows:

$$
\begin{aligned}
\epsilon_{n+1} &= [1+\Delta t F_x(x_n,t_n)]\,\epsilon_n,\\
&= [1+\Delta t F_x(x_n,t_n)]\,[1+\Delta t F_x(x_{n-1},t_{n-1})]\,\epsilon_{n-1},\\
&= [1+\Delta t F_x(x_n)]\,[1+\Delta t F_x(x_{n-1},t_{n-1})]\cdots[1+\Delta t F_x(x_1,t_1)]\,\epsilon_1.
\end{aligned}
$$

Take absolute values now and consider only unsigned errors:

$$\delta_{n+1} = |1+\Delta t F_x(x_n)|\,|1+\Delta t F_x(x_{n-1},t_{n-1})|\cdots|1+\Delta t F_x(x_1,t_1)|\,\delta_1.$$

Let $\max_i|1+\Delta t F_x(x_i,t_i)| = K$ Then,

$$\delta_{n+1} \le K^{n-1}\delta_1,$$

If $K>1$, then ϵ_{n+1} will diverge to infinity, implying runaway growth in the error. Thus, to keep δ_n small for all steps $n \in \mathbb{N}$, we need the exponent $|1+\Delta t F_x(x_i,t_i)|$ to have norm less than unity:

$$|1+\Delta t F_x(x_i,t_i)| \le 1.$$

This is a quadratic inequality with solution

$$-2 \le \Delta t F_x(x_i,t_i) < 0.$$

In order to be a general result, we must take a the worst-case scenario over all possible trajectories:

$$-2 \le \Delta t F_x(x,t) < 0,\ \forall x \in \mathbb{R}, t>0.$$

Example

Consider the ODE $dx/dt = ax$, with $x(0)=x_0$ and (i) $a<0$, and (ii) $a>0$.

Case 1, a negative: We have $F(x,t)=ax$, hence $F_x(x,t)=a$ (Constant). We re-write this as $F_x(x,t)=-|a|$. The stability criterion is thus

$$-2 \le -|a|\Delta t \le 0.$$

The rightmost part of this string is always satisfied, as $-|a|\Delta t$ is definitely negative. The leftmost part is equivalent to

$$\Delta t \le 2/|a|.$$

So, provided Δt is smaller than $2/a$, the algorithm will not become unstable.

Case 2, a positive: We have $F(x,t) = ax$, hence $F_x(x,t) = a$ (Constant). The stability criterion is thus

$$-2 \leq a\Delta t \leq 0.$$

The rightmost branch of this inequality is $a\Delta t \leq 0$ – this can never be satisfied for positive a and the Euler method is unstable in this case – errors are compounded at each iteration.

We can put the two cases together and write down a general condition for stability for the equation $dx/dt = ax$, viz.

$$-2 \leq a\Delta t \leq 0.$$

Indeed, the same analysis can be repeated with x, a, and x_0 complex-valued; the condition for stability is then $|1 + a\Delta t| < 1$. Calling $z := a\Delta t$, we have $|z + 1| \leq 1$. This is a disc of radius 1 centred at $(-1, 0)$ in the complex plane (Figure 16.1). All points strictly inside this disc yield a stable Euler-integration of the ODE.

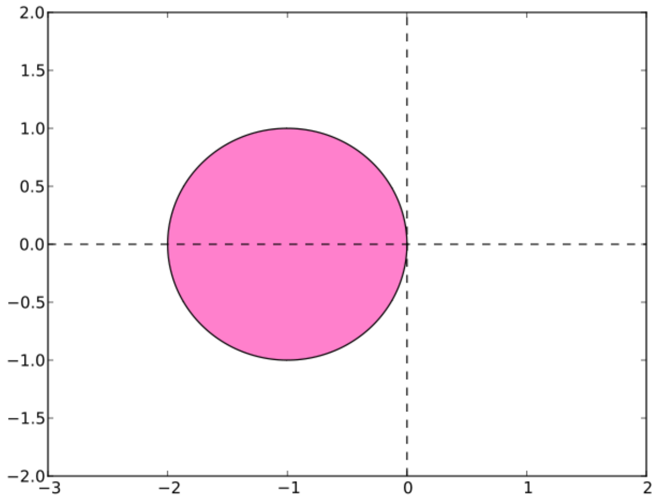


Figure 16.1: Stability region for $dx/dt = ax$ (complex-valued) using the Euler method. The complex variable z whose axes are shown here represents the quantity $a\Delta t$.

This mini-study of integrations of $dx/dt = ax$ with the Euler method leads to the following important conclusion:

Common Programming Error:

Using the Euler method to integrate an ODE – it is usually unstable and is therefore only really a pedagogical tool.

Chapter 17

Runge–Kutta methods

Overview

In the last chapter we approximated solutions of the ODE

$$\frac{dx}{dt} = F(x,t), \qquad x(t=0) = x_0$$

using Euler's method, and found that for certain (even very simple) ODEs, the approximation was numerically unstable. In this chapter, we therefore develop other more robust methods for solving ODEs, and discuss their practical implementation.

17.1 Second-order Runge–Kutta

Recall the Euler method one more time:

$$x_{n+1} = x_n + \Delta t F(x_n, t_n), \qquad x_{n=0} = x_0.$$

Compare this with the true solution over the same interval:

$$X_{n+1} = X_n + \int_{t_n}^{t_{n+1}} F(X(t'), t')\mathrm{d}t'. \tag{17.1}$$

Clearly, the true solution involves contriubtions from the integral of F not just from the point $t = t_n$, but from points in time all across the interval $[t_n, t_{n+1}]$. So a better way of approximating the integral in Equation (17.1) would be using a midpoint-rule, e.g.

$$X_{n+1} \approx X_n + \Delta t F(X_{n+1/2}, t_{n+1/2}).$$

Or, using x_n and x_{n+1} for the approximate solution,

$$x_{n+1} = x_n + \Delta t F(x_{n+1/2}, t_{n+1/2}).$$

The only problem – we don't know what $x_{n+1/2}$ is. However, we can estimate it from doing an Euler step over an interval $[t_n, t_n + (1/2)\Delta t]$:

$$x_{n+1/2} = x_n + \tfrac{1}{2}\Delta t F(x_n, t_n).$$

Thus, we have the **second-order Runge–Kutta scheme**:

$$\begin{aligned} x_{n+1/2} &= x_n + \tfrac{1}{2}\Delta t F(x_n, t_n), & (17.2a) \\ x_{n+1} &= x_n + \Delta t F(x_{n+1/2}, t_{n+1/2}), & (17.2b) \end{aligned}$$

subject to $x_{n=0} = x_0$. This method is second-order accurate with respect to truncation errors.

17.2 Fourth-order Runge–Kutta

A more popular method which is thought to achieve the optimum balance between simplicity and stability and accuracy is the fourth-order Runge–Kutta scheme (RK4):

$$\begin{aligned} k_1 &= \Delta t F(x_n, t_n), \\ k_2 &= \Delta t F\left(x_n + \tfrac{1}{2}k_1, t_n + \tfrac{1}{2}\Delta t\right), \\ k_3 &= \Delta t F\left(x_n + \tfrac{1}{2}k_2, t_n + \tfrac{1}{2}\Delta t\right), \\ k_4 &= \Delta t F\left(x_n + k_3, t_n + \Delta t\right), \\ x_{n+1} &= x_n + \tfrac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right), \end{aligned} \qquad (17.3)$$

subject to $x_{n=0} = x_0$. This method is **fourth-order accurate** with respect to truncation errors. It is much more stable than the Euler method. Although the discussion of the stability of RK4 is beyond the scope of this module, it suffices to say that the stability region shown in Figure 16.1 is substantially extended to the north, south, and west by the use of RK4 (it is not extended eastward, and integrations of $dx/dt = ax$ with RK4 remain numerically unstable for $a \in \mathbb{R}^+$). Anyway, here is what Boyd [p. 174] has to say about RK4:

Runge-Kutta schemes have considerable virtues... First, RK4 is fourth order, that is, has an error decreasing as $O(\Delta t^4)$... Second, it is stable with a rather long time step compared to other methods... Third, Runge–Kutta methods are"self-starting" and do not require a different time-marching scheme to compute the first couple of time steps, as is true of [other methods].

17.3 ODEs with Matlab's built-in functions

Matlab's built-in workhouse ODE solver is called ODE45. This is an adaptive solver built around RK4 and RK5. The RK4 algorithm has a further nice property in addition to those listed in Section 17.2: for very little extra computational expense, one may compute Runge-Kutta methods of adjacent orders simultaneously. These computations can then be used to estimate the error associated with an RK4 timestep. If the error is too large, the timestep can be reduced until the error is acceptably small. Because RK4 is self-starting, this kind of adaptive timestepping is feasible. Thus, one does not specify a timestep in calling ODE45: Matlab chooses it adaptively according to the method just desscribed. However, Matlab checks the timestep not only for accuracy (via the RK4-RK5 comparison), but also for stability. A simple sample code is shown now:

```
function [t,x]=rk_first_order(x0,t0,t1)

% ODE to solve is dx/dt=Fxt, with initial data x(t0)=x0.
% Time interval on which solution is found is [t0,t1].
% Outputs - "t" is a vector of time points;
% "x" is the corresponding vector of x-points.
% These are not set by the user, rather they are determined by Matlab's
% adaptive RK4-RK5 algorithm.

%%% To apply Matlab's default tolerances, use this line:
[t,x] = ode45(@Fxt,[t0 t1],x0,options);

%%% To set your own absolute tolerance, use these lines:
% options = odeset('AbsTol',1e-10);
% [t,x] = ode45(@Fxt,[t0 t1],x0,options);

function dxdt = Fxt(t,x)
    % User-defined function - put whatever you like here, but watch
    % out for ./ and .*
    dxdt=x.*(1-x);
end

end
```

sample_matlab_codes/rk_first_order.m

17.4 A sample project

As a more complicated exaple in Matlab, we numerically solve the equation for the damped, driven, linear pendulum:

$$m\frac{d^2x}{dt^2} = f(x,t), \qquad f(x,t) = -\gamma m\frac{dx}{dt} - m\omega_0^2 + f_0\cos(\omega t).$$

We re-scale time, writing $\tau = \omega t$, such that the equation is re-written as

$$\frac{d^2x}{d\tau^2} + \widehat{\gamma}\frac{dx}{d\tau} + \widehat{\omega}_0^2 x = \frac{F_0}{m\omega^2}\cos\tau.$$

We omit the (by now needless) carets over the symbols and re-write this as a system of first-order equations:

$$\frac{d}{d\tau}\begin{pmatrix} x \\ v \end{pmatrix} = \begin{pmatrix} v \\ -\gamma v - \omega_0^2 x + \alpha\cos(\tau) \end{pmatrix},$$

where $\alpha = F_0/(m\omega^2)$.

```matlab
function [t,x,v,E]=rk_pendulum(x0,v0,t0,t1,gamma_val,omega0,alpha_val)

% ODE to solve linear damped driven oscillator equation:
% x0 - initial condition on position x
% v0 - initial condition on velocity v
% t0 - initial time
% t1 - final time
% gamma_val - value of paramter alpha
% omega0 - value of parameter omega_0
% alpha_val - value of parameter alpha
%
% The names "gamma" and "alpha" are reserved for built-in functions in
% Matlab; their use as user-defined variables can lead to errors.

 options = odeset('AbsTol',1e-12);
 [t,X] = ode45(@FXt,[t0 t1],[x0,v0],options);

% Reshape output array X: the first column should be position and the
% second column should be velocity.

x=X(:,1);
v=X(:,2);

% The following is a test: E should be constant for gamma_val=alpha_val=0:

E=(1/2)*(v.^2)+(1/2)*(omega0^2)*(x.^2);
```

```

function dXdt = FXt(t,X)
    dXdt = zeros(2,1);     % a column vector
    dXdt(1)=X(2);
    dXdt(2)=-gamma_val*X(2)-(omega0^2)*X(1)+alpha_val*cos(t);
end

end
```

sample_matlab_codes/rk_pendulum.m

Exercise 17.1 *Consider the following tasks concerning the code for solving for the motion of the damped driven linear pendulum:*

1. *Implement the pendulum code yourself.*
2. *For $\gamma = \alpha = 0$ show that energy is conserved – to within numerical error.*
3. *For $\gamma = \alpha = 0$ show that the motion is periodic. This can be done by plotting an (x, v) and verifying that this is a closed curve.*
4. *Show analytically (i.e. from the equation of motion) that this closed curve should be an ellipse.*
5. *For $\gamma > 0$ and $\alpha = 0$ show numerically that a plot in (x, v) space is a spiral starting at (x_0, v_0) and spiralling in to the origin. Show also (numerically) that on average, the energy decays as $E \sim \mathrm{e}^{-\gamma t}$.*
6. *For $\gamma > 0$ and $\alpha = 0$ examine the three following cases:*
   - $\gamma > 2\omega_0$ – *overdamped,*
   - $\gamma < 2\omega_0$ – *underdamped,*
   - $\gamma = 2\omega_0$ – *critically damped.*

   Explain these three regimes from a physical point of view.

Exercise 17.2 *Write a Matlab script to solve the following equations numerically:*

$$
\begin{aligned}
\frac{d^2x_1}{dt^2} + x_1 - (x_2 - x_1) &= 0, \\
\frac{d^2x_2}{dt^2} + x_2 + (x_2 - x_1) &= 0,
\end{aligned}
$$

subject to the following initial conditions:

Case 1: *At $t = 0$,*

$$
\begin{aligned}
x_1(0) &= A, \qquad \dot{x}_1(0) = 0, \\
x_2(0) &= B, \qquad \dot{x}_2(0) = 0,
\end{aligned}
$$

where A and B are real parameters.

Hint: The system to be implemented in Matlab's RK4-5 solver is a four-dimensional one.

Chapter 18

Gaussian Elimination

Overview

The aim of this chapter is to investigate a simple and relatively efficient method to solve

$$\mathbf{A}\boldsymbol{x} = \boldsymbol{b},$$

where $\mathbf{A}$ is an $n \times n$ matrix, $\boldsymbol{b}$ is an $n \times 1$ column matrix, and $\boldsymbol{x}$ is an $n \times 1$ column matrix containing **unknown** elements.

18.1 Linear algebra review

Let $\mathbf{A}$ be an $n \times n$ square matrix. We call $\mathbf{A}$ a **linear operator** because it is a map

$$\begin{aligned} \mathbf{A} : \mathbb{R}^n &\rightarrow \mathbb{R}^n, \\ \boldsymbol{x} &\mapsto \mathbf{A}\boldsymbol{x}, \end{aligned}$$

that satisfies the following linearity property:

$$\mathbf{A}(\lambda\boldsymbol{x} + \mu\boldsymbol{y}) = \lambda(\mathbf{A}\boldsymbol{x}) + \mu(\mathbf{A}\boldsymbol{y}),$$

for all λ and μ scalars in $\mathbb{R}$ and all $\boldsymbol{x}$ and $\boldsymbol{y}$ column vectors in $\mathbb{R}^n$. The set

$$\ker(\mathbf{A}) = \{\boldsymbol{x} \in \mathbb{R}^n \big| \mathbf{A}\boldsymbol{x} = 0\}$$

is called the **kernel** of the linear operator $\mathbf{A}$: it is the set of all elements $\boldsymbol{x}$ that gets sent to zero under the operation of $\mathbf{A}$. On the other hand, the set

$$\mathrm{im}(\mathbf{A}) = \{\boldsymbol{y} \in \mathbb{R}^n | \boldsymbol{y} = A\boldsymbol{x}, \,\mathrm{some}\, \boldsymbol{x} \in \mathbb{R}^n\}$$

is the set of all $\boldsymbol{y}$ in $\mathbb{R}^n$ that can be written as $\mathbf{A}$ times some vector $\boldsymbol{x}$. The sets $\ker(\mathbf{A})$ and $\mathrm{im}(\mathbf{A})$ are closed under addition and scalar multiplicaation: they are therefore vector subspaces of $\mathbb{R}^n$. They can therefore be assigned a dimension. A standard result of linear algebra is the following:

$$\dim(\ker(\mathbf{A})) + \dim(\mathrm{im}(\mathbf{A})) = n \tag{18.1}$$

(the number $\dim(\mathrm{im}(\mathbf{A}))$ is often called the **rank** of the matrix $\mathbf{A}$). This leads to the following result called the **Fredholm alternative**:

Theorem 18.1 (Fredholm alternative, $\mathbb{R}^n$) *Let*

$$\begin{aligned} \mathbf{A} : \mathbb{R}^n &\to \mathbb{R}^n, \\ \boldsymbol{x} &\mapsto \mathbf{A}\boldsymbol{x}, \end{aligned}$$

be a linear operator on $\mathbb{R}^n$. Fix $\boldsymbol{b} \neq 0 \in \mathbb{R}^n$. Then, either

1. *$\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ has a unique solution for $\boldsymbol{x}$;*
2. *$\mathbf{A}\boldsymbol{x} = 0$ has a non-trivial solution for $\boldsymbol{x}$.*

If the first case holds, then the linear operator $\mathbf{A}$ is said to be **invertible**.

We can prove this theorem, but our proof does rely on Equation (18.1), whose proof you will have to obtain elsewhere, such as a proper linear algebra class. Now, given Equation (18.1), either

$$\dim(\ker(\mathbf{A})) = 0,$$

or

$$\dim(\ker(\mathbf{A})) = p, \qquad 0 < p \leq n,$$

and clearly, these two alternatives are mutually exclusive. In the first case, the image of $\mathbf{A}$ coincides with the whole of $\mathbb{R}^n$, hence every $\boldsymbol{y}$ in $\mathbb{R}^n$ can be written as $\boldsymbol{y} = \mathbf{A}\boldsymbol{x}$, for some $\boldsymbol{x} \in \mathbb{R}^n$. In particular, given the fixed vector $\boldsymbol{b}$, there exists another vector $\boldsymbol{x} \in \mathbb{R}^n$, such that $\boldsymbol{b} = \mathbf{A}\boldsymbol{x}$. For uniqueness, take $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ such that $\mathbf{A}\boldsymbol{x}_1 = \mathbf{A}\boldsymbol{x}_2 = \boldsymbol{b}$. Subtract and obtain $\mathbf{A}(\boldsymbol{x}_1 - \boldsymbol{x}_2) = 0$. Since the kernel is trivial, we obtain $\boldsymbol{x}_1 = \boldsymbol{x}_2$.

For the second case, by the definition of the kernel, there exists a nontrivial vector $\boldsymbol{x}$ such that $\mathbf{A}\boldsymbol{x} = 0$. Since the two cases are mutually exlusive, the theorem is shown.

Of course, a further result also holds:

Theorem 18.2 *Let*

$$\begin{aligned} \mathbf{A} : \mathbb{R}^n &\to \mathbb{R}^n, \\ \boldsymbol{x} &\mapsto \mathbf{A}\boldsymbol{x}, \end{aligned}$$

be a linear operator on $\mathbb{R}^n$. *Then,* $\mathbf{A}$ *is invertible if and only if*

$$\det(\mathbf{A}) \neq 0,$$

where the determinant is computed according to the usual elementary linear-algebra prescription.

The second theorem also hints at an explicit construction for the inverse:

$$(\mathbf{A}^{-1})_{ij} = \frac{1}{\det(\mathbf{A})} C_{ij}, \tag{18.2}$$

where C_{ij} is the ij^{th} cofactor of the matrix $\mathbf{A}$, computed using the usual linear-algebra prescription.

18.2 Counting the complexity of determinant calculations

Equation (18.2) is interesting from a theoretical point of view. However, it would be extremely foolish to use it to compute the inverse of a matrix in a practical setting. To see why, we demonstrate that the algorithm (18.2) requires $O(n!)$ calculations for its implementation.

For, consider the 2×2 case, with

$$\mathbf{A} = \begin{pmatrix} a_{12} & a_{21} \\ a_{12} & a_{22} \end{pmatrix}.$$

The determinant is $\det(\mathbf{A}) = a_{12}a_{22} - a_{21}a_{12}$, requiring two multiplications for its calculation. On the other hand, in the $n \times n$ case, we have

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

Evaluating along the first row, the determinant is

$$\det(\mathbf{A}) = (-1)^{1+1}a_{11}\begin{vmatrix} a_{22} & \cdots & a_{2n} \\ \vdots & & \vdots \\ a_{n2} & \cdots & a_{nn} \end{vmatrix} + (-1)^{1+2}a_{12}\begin{vmatrix} a_{21} & a_{23} & \cdots & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n3} & \cdots & a_{nn} \end{vmatrix} + \cdots$$
$$+ (-1)^{1+n}a_{1n}\begin{vmatrix} a_{21} & a_{22} & \cdots & a_{2,n-1} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{n,n-1} \end{vmatrix}, \quad (18.3)$$

where there are n subdeterminants of size $(n-1)\times(n-1)$ to evaluate. Thus,

$$[\#\text{ multiplications, determinant of size } n\times n] = n\,[\#\text{ multiplications, determinant of size } (n-1)\times(n-1)]$$

or more compactly

$$N_n = nN_{n-1}.$$

We telescope this result:

$$\begin{aligned} N_n &= nN_{n-1}, \\ &= n(n-1)N_{n-2}, \\ &= n(n-1)(n-2)\cdots 3\,N_2, \\ &= n(n-1)(n-2)\cdots 3\cdot 2, \\ &= n! \end{aligned}$$

The therefore say that the number of calculations required to compute a determinant is $O(n!)$. For large n, we may use Sterling's approximation:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n,$$

so that the number of calculations grows as $n^{1/2}(n/e)^n$. **This is a disaster!** We therefore need a better way to find the inverse of a matrix.

Another reason to avoid this kind of approach is because the definition of the determinant in Equation (18.3) is **recursive**. Suppose we have a Matlab function to compute a determinant of size $n\times n$. To do this, Matlab would have to compute determinants of size $(n-1)\times(n-1)$. However, to do this subordinate calculation, Matlab would have to compute determinants of size $(n-2)\times(n-2)$. The result would be a function that calls itself to call itself to call itself... until determinants of size 2×2 are reached. A function or algorithm that calls itself reapeatedly, Russian-dolls fashion, is called **recursive**. Matlab hates recursive function calls and sets a very low limit on the number of recursive calls that can be made to a user-defined function. For that reason, the writing of such functions are to be avoided.

Common Matlab Programming Error:

Expecting large-scale user-defined recursive functions to run in Matlab.

Clearly, a better approach is needed.

18.3 Gaussian elimination

You are already familiar with Gaussian elimination! Recall the solution of three linear simultaneous equations, e.g.

$$\begin{aligned} 2x + y - z &= 8 & (L_1) \\ -3x - y + 2z &= -11 & (L_2) \\ -2x + y + 2z &= -3 & (L_3) \end{aligned}$$

We eliminate x from the last two equations as follows:

1. Take $L_2 + (3/2)L_1$:

$$\begin{aligned} -3x - y + 2z &= -11 \\ 3x + \tfrac{3}{2}y - \tfrac{3}{2}z &= 12, \end{aligned}$$

   giving a new equation

$$y + z = 2.$$

2. Take $L_3 + L_1$:

$$\begin{aligned} -2x + y + 2z &= -3 \\ 2x + y - z &= 8 \end{aligned}$$

   giving a new equation

$$2y + z = 5.$$

3. Gather up our new system of equations:

$$\begin{aligned} 2x + \;\; y - z &= 8 \qquad (L_1') \\ y + z &= 2 \qquad (L_2') \\ 2y + z &= 5 \qquad (L_3') \end{aligned}$$

4. We repeat the Gaussian elimination on L_2' and L_3'. We take $L_3' - 2L_2'$ to obtain a new equation

$$z = -1.$$

5. We gather up our new equations again:

$$\begin{aligned} 2x + y - z &= \;\; 8 \qquad (L_1'') \\ y + z &= \;\; 2 \qquad (L_2'') \\ + z &= -1 \qquad (L_3'') \end{aligned}$$

   This is called a **triangular system system** because the equivalent matrix problem can be written as

$$\underbrace{\begin{pmatrix} 2 & 1 & -1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}}_{=\widetilde{\boldsymbol{A}}} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 8 \\ 2 \\ -1 \end{pmatrix},$$

   and the new matrix $\widetilde{\boldsymbol{A}}$ has non-zero entries only along the diagonal and in the upper triangle.

6. This problem can now be solved by **backsubstitution**:

   (a) L_3'' implies that $z = -1$.

   (b) Substitute into L_2'' to obtain $y + (-1) = 2$, hence $y = 3$.

   (c) Substitute into L_1'' to obtain $2x + (3) - (-1) = 8$, hence $x = 2$.

We are now going to systematize this procedure by considering the abstract problem

$$\begin{aligned} a_{11}x + a_{12}y + a_{13}z &= b_1, \qquad (L_1) \\ a_{21}x + a_{22}y + a_{23}z &= b_2, \qquad (L_2) \\ a_{31}x + a_{32}y + a_{33}z &= b_3. \qquad (L_3) \end{aligned}$$

This is the subject of the next chapter.

Chapter 19

Gaussian Elimination – the algorithm

Overview

We systematize the procedure of Gaussian elimination by considering the abstract problem

$$\begin{aligned} a_{11}x + a_{12}y + a_{13}z &= b_1, \qquad (L_1) \\ a_{21}x + a_{22}y + a_{23}z &= b_2, \qquad (L_2) \\ a_{31}x + a_{32}y + a_{33}z &= b_3. \qquad (L_3) \end{aligned}$$

This enables us to write a Matlab code to automate Gaussian elimination, for square matrices of any size.

19.1 The algorithm

1. **First elimination operation**, `i=1`.

   Take $L_2 - (a_{21}/a_{11})L_1$:

   $$\begin{aligned} a_{21}x + a_{22}y + a_{23}z &= b_2, \\ -\left(a_{11}\frac{a_{21}}{a_{11}}x + a_{12}\frac{a_{21}}{a_{11}}y + a_{13}\frac{a_{21}}{a_{11}}z \right.&= \left. b_1\frac{a_{21}}{a_{11}}\right). \end{aligned}$$

   The result is a new equation

   $$y\left(a_{22} - \frac{a_{12}a_{21}}{a_{11}}\right) + z\left(a_{23} - \frac{a_{13}a_{21}}{a_{11}}\right) = b_2 - \frac{b_1 a_{21}}{a_{11}}$$

Call

$$\begin{aligned} a'_{22} &= a_{22} - a_{12}\left(\frac{a_{21}}{a_{11}}\right), \\ a'_{23} &= a_{23} - a_{13}\left(\frac{a_{21}}{a_{11}}\right), \\ b'_{2} &= b_{2} - b_{1}\left(\frac{a_{21}}{a_{11}}\right), \end{aligned}$$

Hence, the new equation is

$$a'_{22}y + a'_{23}z = b'_2.$$

First elimination operation, i=1, **first new equation**:

```
(i=1)
m=A(2,1)/A(1,1)
(j=i+1=2)

... [hence, m=A(j,i)/A(i,i)]

A(2,2 ... 3) --> A(2,2 ... 3)-m*A(1,2 ... 3)

... [hence, A(j,i+1:n)=A(j,i+1:n)-m*A(i,i+1:n)

b(2) --> b(2)-m*b(1)

... [hence, b(j) --> b(j)-m*b(i)]
```

2. The same game for $L_3 - (a_{31}/a_{11})L_1$:

$$y\left(a_{32} - \frac{a_{12}a_{31}}{a_{11}}\right) + z\left(a_{33} - \frac{a_{13}a_{31}}{a_{11}}\right) = b_3 - \frac{b_1 a_{31}}{a_{11}}$$

Call

$$\begin{aligned} a'_{32} &= a_{32} - a_{12}\left(\frac{a_{31}}{a_{11}}\right), \\ a'_{33} &= a_{33} - a_{13}\left(\frac{a_{31}}{a_{11}}\right), \\ b'_3 &= b_3 - b_1\left(\frac{a_{31}}{a_{11}}\right). \end{aligned}$$

Hence, the new equation is

$$a'_{32}y + a'_{33}z = b'_3.$$

First elimination operation, i=1, **second new equation**:

```
(i=1)
m=A(3,1)/A(1,1)
(j=i+1+1=3)

... [hence, m=A(j,i)/A(i,i)]

A(3,2 ... 3) --> A(3,2 ... 3)-m*A(1,2 ... 3)

... [hence, A(j,i+1:n)=A(j,i+1:n)-m*A(i,i+1:n)

b(3) --> b(3)-m*b(1)

... [hence, b(j) --> b(j)-m*b(i)
```

3. We write down the new system:

$$\begin{aligned} a_{11}x + a_{12}y + a_{13}z &= b_1, \qquad (L'_1) \\ a'_{22}y + a'_{23}z &= b'_2, \qquad (L'_2) \\ a'_{32}y + a'_{33}z &= b'_3, \qquad (L'_3) \end{aligned}$$

4. Do another round of eliminations by taking $(L_3') - (a_{32}'/a_{22}')(L_2')$:

$$\begin{aligned} a_{32}'y + a_{33}'z &= b_3', \\ -\left(\frac{a_{32}'}{a_{22}'}a_{22}'y + \frac{a_{32}'}{a_{22}'}a_{23}'z \right.&= \left.\frac{a_{32}'}{a_{22}'}b_2'\right) \end{aligned}$$

   In other words,

$$\begin{aligned} a_{32}'y + a_{33}'z &= b_3', \\ -\left(a_{32}'y + \frac{a_{32}'a_{23}'}{a_{22}'}z \right.&= \left.\frac{a_{32}'b_2'}{a_{22}'}\right), \end{aligned}$$

   leading to a single new equation

$$\left(a_{33}' - \frac{a_{32}'a_{23}'}{a_{22}'}\right) z = b_3' - \frac{a_{32}'b_2'}{a_{22}'}.$$

   Calling

$$a_{33}'' = a_{33}' - a_{23}'\left(\frac{a_{32}'}{a_{22}'}\right), \qquad b_3'' = b_3' - b_2'\left(\frac{a_{32}'}{a_{22}'}\right),$$

   this becomes

$$\alpha_{33}''z = b_3''.$$

Second elimination operation, i=2, **first (and only) new equation**:

```
(i=2)
m=A(3,2)/A(2,2)
(j=i+1=3)

... [hence, m=A(j,i)/A(i,i)]

A(3,3) --> A(3,3)-m*A(2,3)

... [hence, A(j,i+1:n)=A(j,i+1:n)-m*A(i,i+1:n)

b(3) --> b(3)-m*b(2)

... [hence, b(j) --> b(j)-m*b(i)]
```

5. Finally, we obtain an upper triangular system:

$$\begin{aligned} a_{11}x + a_{12}y + a_{13}z &= b_1, \qquad (L_1'') \\ a_{22}'y + a_{23}'z &= b_2', \qquad (L_2'') \\ a_{33}''z &= b_3'', \qquad (L_3'') \end{aligned}$$

   which can be solved by backsubstitution.

We can now gather up the bits of pseudocode to make a set of nested loops:

```
% Loop over the number of eliminations to do, with i=1-->(n-1):
for i=1:n-1
  for j=i+1:n
    m=A(j,i)/A(i,i);
    A(j,i+1:n)=A(j,i+1:n)-m*A(i,i+1:n);
    b(j)=b(j)-m*b(i);
  end
end
```

Common Programming Error:

The Gaussian elimination pseudocode does not overwrite elements in the lower triangle (these are not accessed at all in the nested loops just given). Rather, these are not used at all in the backsubstitution.

19.2 The backsubstitution step

Consider now the modified set of equations

$$\begin{aligned} a_{11}x + a_{12}y + a_{13}z &= b_1, \qquad (L_1'') \\ a_{22}'y + a_{23}'z &= b_2', \qquad (L_2'') \\ a_{33}''z &= b_3'', \qquad (L_3'') \end{aligned}$$

Using Matlab-like notation for the arrays, we have

$$\begin{aligned} z &= \frac{b(3)}{A(3,3)}, \\ y &= \frac{b(2) - A(2,3)z}{A(2,2)}, \\ x &= \frac{b(1) - [A(1,3)z + A(1,2)y]}{A(1,1)}. \end{aligned}$$

Guess the pattern:

$$x(i) = \frac{b(i) - \sum_{k=i+1}^{n} A(i,k)x(k)}{A(i,i)},$$

with the important observation that a starting value is needed to get this algorithm going:

$$x_n = \frac{b(n)}{A(n,n)}.$$

Implement the 'for' loops:

```
x(n)=b(n)/A(n,n);


x=0*(1:n);
for i=n-1:-1:1
  x(i)=(b(i)-sum(A(i,i+1:n).*x(i+1:n)))/A(i,i);
end
```

19.3 Putting it all together

We assemble a '.m' file to do Gaussian elimination:

```
function [x,diff,A0,b0]=naive_gauss_elim()

% User-defined arrays here:

```

```
A=[2,1,-1;-3,-1,2;-2,1,2];
b=[8,-11,-3]';

b0=b;
A0=A;

n=length(b);

% We wish to solve Ax=b.

% ************************************************************************
% First, we find the upper-triangular matrix and the associated b-vector:

for i=1:n-1
    for j=i+1:n
        m=A(j,i)/A(i,i);
        A(j,i+1:n)=A(j,i+1:n)-m*A(i,i+1:n);
        b(j)=b(j)-m*b(i);
    end
end

% ************************************************************************
% Now, we do backsubstitution:

x=zeros(n,1);
x(n)=b(n)/A(n,n);

for i=(n-1):-1:1
    sum_val=0;
    for k=i+1:n
        sum_val=sum_val+A(i,k)*x(k);
    end
  x(i)=(b(i)-sum_val)/A(i,i);
end

% ************************************************************************
% Finally, we test how far from the true inverse we are:

diff=A0*x-b0;
diff=sqrt(sum(diff.*diff));

end
```

sample_matlab_codes/naive_gauss_elim.m

Chapter 20

Gaussian Elimination – performance and operation count

Overview

Gaussian elimination can fail – even for invertible matrices. We examine this issue, together with a study of the complexity of the algorithm. We demonstrate that the number of calculations is algebraically growing in n, the matrix size. This compares excellently with the nightmare of determinant calculations, which were $O(n!)$.

20.1 Pivoting

Even if we could ignore for a second the fact that the reduction of the matrix $\mathbf{A}$ to upper-triangular form requires division by diagonal elements, we are still forced to consider the following backsubstitution formula for the inversion of $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$:

$$x_i = \frac{b_i - \sum_{k=i+1}^{n} A_{ik} x_k}{A_{ii}},$$

where A is assumed to be in upper-triangular form, and where we have the starting-value

$$x_n = \frac{b_n}{A_{nn}}.$$

Thus, division by diagonal elements is undavoidable in Gaussian elimination and backsubstitution. However, just because a matrix has a zero on the diagonal, it does not mean that it is non-invertible, e.g.

$$\mathbf{A} = \begin{pmatrix} 0 & 1 \\ -1 & 1 \end{pmatrix}$$

has $\det(\mathbf{A}) = +1$ and is therefore invertible. Moreover, some matrices have diagonal entries that 'just barely zero', e.g.

$$\mathbf{A} = \begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix}, \qquad |\epsilon| \ll 1$$

and numerical Gaussian elimination can introduce large errors if the difference between ϵ and $1+\epsilon$ cannot be detected by the machine precision.

A very clever and easy way out of both problems is called **pivoting**: we swap rows in the problem so that the troublesome diagonal elements are eliminated from the upper-triangular matrix in the Gaussian-elimination process. This works because any two rows in the problem

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1, \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2, \\ \vdots \qquad\qquad & \quad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

can be interchanged without affecting the answer. The simplest possible pivoting method is called **partial pivoting**:

- Consider the i^{th} column of the matrix. Search the portion of the i^{th} column including and below the diagonal to find the element with the largest absolute value. Let p be the row index of this element.
- Interchange **rows** i and p.
- Proceed with the elimination, doing this swap once per full elimination operation.

Let's have a look at a very silly simple example by considering again the matrix

$$\mathbf{A} = \begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix}$$

For $\epsilon \neq 0$ the matrix is invertible. For definieteness, suppose we wish to solve

$$\mathbf{A}\boldsymbol{x} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}.$$

We already know the answer:

$$x_1 = \frac{1}{1-\epsilon} = 1 + O(\epsilon), \tag{20.1a}$$
$$x_2 = \frac{1-2\epsilon}{1-\epsilon} = 1 + O(\epsilon), \tag{20.1b}$$

Let's try a Gaussian elimination, starting with the equations

$$\begin{aligned} \epsilon x_1 + x_2 &= 1, \\ x_1 + x_2 &= 2. \end{aligned}$$

Although we would never do the following thing in practice, we must now take (Equation 2)-$(1/\epsilon)$(Equation 1), as this is what the algorithm tells us to do. In fact, our eye immediately suggests that we take (Equation 1)-ϵ(Equation 2); however, we must adhere scrupulously to the path mapped out by the algorithm, and investigate where it takes us. Consequently, we obtain a new pair of equations

$$\begin{aligned} \epsilon x_1 + x_2 &= 1, \\ x_2\left(1 - \frac{1}{\epsilon}\right) &= 2 - \frac{1}{\epsilon}. \end{aligned}$$

Suppose now that ϵ is less than machine epsilon, such that the computer can't tell the difference between 1 and $1 + (1/\epsilon)$. Then, $1/|\epsilon|$ is large, and the computer cannot tell the difference between $1 - (1/\epsilon)$ and $2 - (1/\epsilon)$ – it will calculate both numbers as $-1/\epsilon$, to machine precision. Then, to the computer, the second equation would look like

$$x_2\left(-\frac{1}{\epsilon}\right) = -\frac{1}{\epsilon}$$

and numerical backsubstitution would yield

$$x_2 = 1.$$

However, a further backsubstitution into the first equation would yield

$$x_1 = 0.$$

This is completely incorrect (c.f. Equation (20.1b)). **Large rounding errors have been introduced by the division by ϵ.**

Let us now implement the partial pivoting suggested at the start of the section. We swap the first and second rows to obtain an equivalent set of equations:

$$\begin{aligned} x_1 + x_2 &= 2, \\ \epsilon x_1 + x_2 &= 1, \end{aligned}$$

Following the algorithm, we take (Equation 2)-ϵ(Equation 1) to obtain

$$\begin{aligned} x_1 + x_2 &= 2, \\ x_2(1-\epsilon) &= 1 - 2\epsilon. \end{aligned}$$

Back-substitution gives

$$x_2 = \frac{1-2\epsilon}{1-\epsilon} \overset{\text{M.P.}}{=} 1,$$

and

$$x_1 = 1,$$

which is the correct answer.

Typically, pivoting is implemented once per full elimination operation, as in the following sample code:

```matlab
function [x,diff,A0,b0]=pivot_gauss_elim(n)

% A more rigorous test than before. -- A is an nxn square matrix with random
% entries between 0 and 1.
% See a later chapter for more information about random matrices.
%
% Here n is supplied by the user at the command line.
%
% Also, b is a random nx1 column vector.

A=rand(n,n);
b=rand(n,1);

b0=b;
A0=A;

% ***********************************************************************
% We wish to solve Ax=b.
% ***********************************************************************
% First, we find the upper-triangular matrix and the associated b-vector:

for i=1:n-1
```

```

    % *******************************************************************
    % Partial pivoting here:

    temp1=A(i:n,i);
    [~,p]=max(abs(temp1));
    % The next line needs a lot of caution!!!!
    p=p+i-1;

    temp2=A(i,:);
    A(i,:)=A(p,:);
    A(p,:)=temp2;

    temp3=b(i);
    b(i)=b(p);
    b(p)=temp3;

    % *******************************************************************
    % Now back to old-fahsioned Gaussian elimination.

    for j=i+1:n
        m=A(j,i)/A(i,i);
        A(j,i+1:n)=A(j,i+1:n)-m*A(i,i+1:n);
        b(j)=b(j)-m*b(i);
    end
end

% *********************************************************************
% Now, we do backsubstitution:

x=zeros(n,1);
x(n)=b(n)/A(n,n);

for i=(n-1):-1:1
    sum_val=0;
    for k=i+1:n
        sum_val=sum_val+A(i,k)*x(k);
    end
  x(i)=(b(i)-sum_val)/A(i,i);
end

% *********************************************************************
% Finally, we test how far from the true inverse we are:

diff=A0*x-b0;
```

```
diff=sqrt(sum(diff.*diff));

end
```

sample_matlab_codes/pivot_gauss_elim.m

The failure of pivoting

Assuming infinite precision, the failure of partial pivoting means that the matrix is NON-invertible.

20.2 Operation counts

Recall, we showed that direct evaluation of the determinant of an $n \times n$ matrix is $O(n!)$, and that direct inversion of a matrix is computationally infeasible as a result. In this section we show that Gaussian elimination requires far fewer operations to perform.

The analysis is quite simple and begins with the nested loops for the elimination step:

```
% Loop over the number of eliminations to do, with i=1-->(n-1):
for i=1:n-1
  for j=i+1:n
    m=A(j,i)/A(i,i);
    A(j,i+1:n)=A(j,i+1:n)-m*A(i,i+1:n);
    b(j)=b(j)-m*b(i);
  end
end
```

However, there is a third loop buried in here in the assignment

```
A(j,i+1 ... n) --> A(j,i+1 ... n)-m*A(i,i+1 ... n),
```

so the set of nested loops is really

```
% Loop over the number of eliminations to do, with i=1-->(n-1):
for i=1:n-1
  for j=i+1:n
    m=A(j,i)/A(i,i);
      for k=i+1:n
        A(j,k)=A(j,k)-m*A(i,k);
      end
    b(j)=b(j)-m*b(i);
  end
end
```

We count how many times the innermost piece of code is accessed, as this will be equal to the number of calculations performed, up to a prefactor. This number is

$$\begin{aligned}
\text{Count} &= \sum_{i=1}^{n-1}\left[\sum_{j=i+1}^{n}\left(\sum_{k=i+1}^{n}1\right)\right],\\
&= \sum_{i=1}^{n-1}\left(\sum_{j=i+1}^{n}1\right)\left(\sum_{k=i+1}^{n}1\right),\\
&= \sum_{i=1}^{n-1}(n-i)^2,\\
&= \sum_{i=1}^{n-1}\left(n^2-2ni+i^2\right),\\
&= n^2(n-1)-2n\sum_{i=1}^{n-1}i+\sum_{i=1}^{n-1}i^2.
\end{aligned}$$

Using standard formulae, this is

$$\begin{aligned}
\text{Count} &= n^3-n^2-2n\left(\tfrac{1}{2}n(n-1)\right)+\left(\tfrac{1}{6}n(n-1)(2n-1)\right),\\
&= \tfrac{1}{3}n^3+O(n^2).
\end{aligned}$$

Backsubstitution

We consider the backsubstitution step:

```
x(n)=b(n)/A(n,n);

x=0*(1:n);
for i=n-1:-1:1
  x(i)=(b(i)-sum(A(i,i+1:n).*x(i+1:n)))/A(i,i);
end
```

The 'for' loop here is really two nested loops. We make this explicit by rewriting the ‘for’ loop as follows:

```
x=0*(1:n);
for i=n-1:-1:1
  sum_val=0;
  for k=i+1:n
    sum_val=sum_val+A(i,k)*x(k);
  end
  x(i)=(b(i)-sum_val)/A(i,i);
end
```

We count the number of times the innermost piece of code is accessed:

$$\begin{aligned}
\sum_{i=n-1,\dots}^{1}\left(\sum_{k=i+1}^{n}1\right) &= \sum_{n-1,\dots}^{1}(n-i),\\
&= \sum_{m=1}^{n} m,\\
&= \tfrac{1}{2}n(n-1),\\
&= \tfrac{1}{2}n^2+O(n).
\end{aligned}$$

The total count (elimination+backsubstitution) is thus

$$\begin{aligned}
\text{Total count} &= \tfrac{1}{3}n^3+O(n^2)+\tfrac{1}{2}n^2+O(n),\\
&= \tfrac{1}{3}n^3+O(n^2).
\end{aligned}$$

In other words, the number of calculations required to do Gaussian elimination is proportional to n^3 – a dramatic improvement over determinant calculations.

Other considerations

For massive calculations (e.g. $n\sim 10^6$), even the relatively good performance of Gaussian elimination ($O(n^3)$) is not satisfactory. For such large calculations, iterative methods are preferred, where the count is $O(n_{\mathrm{c}}n^2)$, and where n_{c} is the number of iterations required for the method to converge (typically $n_{\mathrm{c}}\sim 10^2$).

20.3 The inverse of a matrix – explicitly

We are not interested in examining further methods for computing the inverse of a matrix explicitly. However, we can get it for free from Gaussian elimination as follows.

Let

$$\boldsymbol{e}_{(i)} = \big(0, 0, \cdots, 0, \underbrace{1}_{i^{\text{th}}\,\text{slot}}, 0, \cdots, 0\big)^T,$$

be an $n \times 1$ column vector and let $\mathbf{B}$ be an arbitrary $n \times n$ real matrix. We have,

$$\begin{aligned}\big(\mathbf{B}\boldsymbol{e}_{(i)}\big)_\alpha &= \sum_\beta B_{\alpha\beta}\boldsymbol{e}^{(i)}_\beta, \\ &= \sum_\beta B_{\alpha\beta}\delta_{\beta i}, \\ &= B_{\alpha i}.\end{aligned}$$

Hence, the column vector $\mathbf{B}\boldsymbol{e}_{(i)}$ is the i^{th} column of the matrix $\mathbf{B}$. Hence,

$$\mathbf{A}\boldsymbol{x} = \boldsymbol{e}_{(i)} \implies \boldsymbol{x} = (\mathbf{A}^{-1})\boldsymbol{e}_{(i)},$$

and $\boldsymbol{x}$ is the i^{th} column of the matrix $\mathbf{A}^{-1}$. Thus, n successive Gaussian eliminations, with $\boldsymbol{b} = \boldsymbol{e}_{(i)}$ and $i = 1, \cdots, n$ give the columns of $\mathbf{A}^{-1}$ explicitly; hence, $\mathbf{A}^{-1}$ itself is determined.

20.4 Matlab's built-in functions

Matlab has a built-in method to do Gaussian elimination and invert matrices. It is not much different from the .m codes we have developed in this chapter, although presumably they make more use of vectorization that we have done. These operations are listed here, and assume that a square matrix $\mathbf{A}$ and a column vector $\boldsymbol{b}$ of appropriate size are defined on the command line.

- To solve $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ for $\boldsymbol{x}$, without necessarily finding $\mathbf{A}^{-1}$ (i.e. Gaussian elimination):

  ```
  x=A\b
  ```

  or, equivalently,

  ```
  x=mldivide(A,b)
  ```

  (matrix left-divide).

- To find the inverse of a matrix, type

  ```
  A^(-1)
  ```

  It could not be easier!

Chapter 21

Operator norm, condition number

21.1 Overview

In linear-algebra calculations, we are sometimes very unfortunate, and have to solve a problem like $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ (give, fixed $\mathbf{A}$), where small changes in $\boldsymbol{b}$ produce extremely large changes in $\boldsymbol{x}$. Such problems are said to be **ill-conditioned**. The aim of this chapter is to quantify this bad behaviour.

21.2 Motivation

Consider the problem

$$\mathbf{A}\boldsymbol{x} = \boldsymbol{b},$$

where

$$\mathbf{A} = \begin{pmatrix} 1.002 & 1 \\ 1 & 0.998 \end{pmatrix},$$

and where $\boldsymbol{b} = (2.002, 1.998)^T$. Solving in the usual fashion gives $\boldsymbol{x} = (1, 1)^T$. Now, consider the effect of a small change in $\boldsymbol{b}$; suppose we have instead

$$\boldsymbol{b}' = \begin{pmatrix} 2.0021 \\ 1.998 \end{pmatrix}.$$

Solving again in the usual fashion gives

$$\boldsymbol{x}' = \begin{pmatrix} -23.95 \\ 26.00 \end{pmatrix}.$$

A small change in $\boldsymbol{b}$, with

$$\frac{\|\boldsymbol{b}' - \boldsymbol{b}\|_2}{\|\boldsymbol{b}\|_2} = 0.0025\%$$

has produced an enormous change in $\boldsymbol{x}$, with

$$\frac{\|\boldsymbol{x}' - \boldsymbol{x}\|_2}{\|\boldsymbol{x}\|_2} = 24\%$$

(here $\|\cdot\|_2$ means the usual L^2 norm for vectors). The aim of the rest of the chapter is to investigate this unusual amplification of small differences.

21.3 The operator norm

Definition 21.1 (The L^2-norm of a matrix) *Let $\mathbf{A} \in \mathbb{R}^{n\times n}$ be a real matrix. We define the L^2-norm of $\mathbf{A}$ as follows:*

$$\|\mathbf{A}\|_2 = \sup_{\boldsymbol{x}\neq 0} \frac{\|\mathbf{A}\boldsymbol{x}\|_2}{\|\boldsymbol{x}\|_2}.$$

Throughout the rest of this module, we refer to the L^2 norm of a matrix as the **operator norm**.

Theorem 21.1 (Properties of the operator norm) *Let $\mathbf{A} \in \mathbb{R}^{n\times n}$ be a real matrix. We have the following set of properties of the operator norm:*

1. *Positive-definite:* $\|\mathbf{A}\|_2 \geq 0$, $\|\mathbf{A}\|_2 = 0 \implies \mathbf{A} = 0$,
2. *Linearity under scalar multiplication:*
$$\|\mu\mathbf{A}\|_2 = |\mu|\|\mathbf{A}\|_2,$$
3. *The triangle inquality:*
$$\|\mathbf{A}+\mathbf{B}\|_2 \leq \|\mathbf{A}\|_2 + \|\mathbf{B}\|_2,$$
4. *Cauchy–Schwarz-type inequalities*
$$\|\mathbf{A}\mathbf{B}\|_2 \leq \|\mathbf{A}\|_2\|\mathbf{B}\|_2, \qquad \|\mathbf{A}\boldsymbol{x}\|_2 \leq \|\mathbf{A}\|_2\|\boldsymbol{x}\|_2,$$

for all $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n\times n}$, $\boldsymbol{x} \in \mathbb{R}^n$, and all $\mu \in \mathbb{R}$.

Exercise 21.1 *Prove Theorem 21.1.*

Lemma 21.1 *Let $\mathbf{A} \in \mathbb{R}^{n\times n}$ be a real matrix. Then,*

1. *$\mathbf{A}^T\mathbf{A}$ is symmetric;*

2. *The eigenvalues of $\mathbf{A}^T\mathbf{A}$ are non-negative.*

Proof:

1. We have
$$(\mathbf{A}^T\mathbf{A})^T = \mathbf{A}^T(\mathbf{A}^T)^T = \mathbf{A}^T\mathbf{A},$$
hence $\mathbf{A}^T\mathbf{A}$ is symmetric.

2. Let
$$(\mathbf{A}^T\mathbf{A})\boldsymbol{x} = \lambda\boldsymbol{x}.$$
We dot both sides by $\boldsymbol{x}$ using the ordinary dot product. For brevity, we use the following notation:
$$\boldsymbol{x}\cdot\boldsymbol{y} \equiv \boldsymbol{x}^T\boldsymbol{y} \equiv (\boldsymbol{x},\boldsymbol{y}).$$
We have,
$$\begin{aligned}\left(\boldsymbol{x},\mathbf{A}^T\mathbf{A}\boldsymbol{x}\right) &= \lambda\left(\boldsymbol{x},\boldsymbol{x}\right),\\ (\mathbf{A}\boldsymbol{x},\mathbf{A}\boldsymbol{x}) &= \lambda\|\boldsymbol{x}\|_2^2,\end{aligned}$$
hence, $\|\mathbf{A}\boldsymbol{x}\|_2^2 = \lambda\|\boldsymbol{x}\|_2^2$, and $\lambda \geq 0$.

Theorem 21.2 (An explicit method to compute the operator norm) *Let $\mathbf{A} \in \mathbb{R}^{n\times n}$ be a real matrix. We have the following identity:*

$$\|\mathbf{A}\|_2 = \sqrt{\lambda_{\max}}$$

where $\lambda_{\max}$ denotes the largest eigenvalue of the matrix $\mathbf{A}^T\mathbf{A}$.

Proof: We have,

$$\begin{aligned}\|\mathbf{A}\boldsymbol{x}\|_2^2 &= (\mathbf{A}\boldsymbol{x},\mathbf{A}\boldsymbol{x}),\\ &= \left(\mathbf{A}^T\mathbf{A}\boldsymbol{x},\boldsymbol{x}\right),\\ &= \left(\boldsymbol{x},\mathbf{A}^T\mathbf{A}\boldsymbol{x}\right).\end{aligned}$$

Now $\mathbf{A}^T\mathbf{A}$ is a real, symmetric $n\times n$ matrix so by the spectral theorem, the eigenvectors of $\mathbf{A}^T\mathbf{A}$

span $\mathbb{R}^n$ and are orthonormal. Thus, we can write

$$\boldsymbol{x} = \sum_i \alpha_i \boldsymbol{x}_i, \qquad \left(\mathbf{A}^T\mathbf{A}\right)\boldsymbol{x}_i = \lambda_i \boldsymbol{x}_i, \qquad (\boldsymbol{x}_i, \boldsymbol{x}_j) = \delta_{ij}.$$

Thus, we have

$$\begin{aligned} \left(\boldsymbol{x}, \mathbf{A}^T\mathbf{A}\boldsymbol{x}\right) &= \left(\sum_i \alpha_i \boldsymbol{x}_i\right) \cdot \left(\sum_j \lambda_j \alpha_j \boldsymbol{x}_j\right), \\ &= \sum_{ij} \alpha_i \alpha_j \lambda_j \delta_{ij}, \\ &= \sum_i \alpha_i^2 \lambda_i. \end{aligned}$$

Hence,

$$\|\mathbf{A}\boldsymbol{x}\|_2^2 = \left(\boldsymbol{x}, \mathbf{A}^T\mathbf{A}\boldsymbol{x}\right) = \sum_i \alpha_i^2 \lambda_i. \tag{21.1}$$

Let $\lambda_{\max} = \max_i \lambda_i$, and let $i_{\max}$ be the index of the maximal eigenvalue. The expression (21.1) is maximized by taking $\alpha_i = 0$ unless $i = i_{\max}$, i.e. $\boldsymbol{x} \propto \boldsymbol{x}_{i_{\max}}$. Note that this argument is not affected by the presence of eigenspaces of dimension higher than one, i.e. it is not affected by degenerate eigenvalues. Thus,

$$\|\mathbf{A}\boldsymbol{x}\|_2^2 = \alpha_{i_{\max}}^2 \lambda_{\max},$$

and

$$\frac{\|\mathbf{A}\boldsymbol{x}\|_2^2}{\|\boldsymbol{x}\|_2^2} = \lambda_{\max}.$$

In other words,

$$\sup_{\boldsymbol{x}\neq 0} \frac{\|\mathbf{A}\boldsymbol{x}\|_2}{\|\boldsymbol{x}\|_2} = \sqrt{\lambda_{\max}},$$

as required.

21.4 The condition number

Let $\mathbf{A}$ be an invertible matrix, and let

$$\begin{aligned} \mathbf{A}\boldsymbol{x}_1 &= \boldsymbol{b}_1, \\ \mathbf{A}\boldsymbol{x}_2 &= \boldsymbol{b}_2. \end{aligned}$$

We consider

$$\boldsymbol{x}_2 - \boldsymbol{x}_1 = \mathbf{A}^{-1}\left(\boldsymbol{b}_2 - \boldsymbol{b}_1\right).$$

Using the operator norm, it follows that

$$\|\boldsymbol{x}_2 - \boldsymbol{x}_1\|_2 \leq \|\mathbf{A}^{-1}\|_2 \| \left(\boldsymbol{b}_2 - \boldsymbol{b}_1\right) \|_2.$$

Divide both sides by $\|\boldsymbol{x}_1\|_2$:

$$\frac{\|\boldsymbol{x}_2 - \boldsymbol{x}_1\|_2}{\|\boldsymbol{x}_1\|_2} \leq \frac{\|\mathbf{A}^{-1}\|_2 \| \left(\boldsymbol{b}_2 - \boldsymbol{b}_1\right) \|_2}{\|\boldsymbol{x}_1\|_2} \tag{21.2}$$

We want an estimate of bad behaviour – but this should be independent of the solution and depend only on chosen input parameters. Consider

$$\begin{aligned}
\|\boldsymbol{b}_1\|_2 &= \|\mathbf{A}\boldsymbol{x}_1\|_2, \\
\|\boldsymbol{b}_1\|_2 &\leq \|\mathbf{A}\|_2 \|\boldsymbol{x}_1\|_2, \\
\frac{\|\boldsymbol{b}_1\|_2}{\|\mathbf{A}\|_2} &\leq \|\boldsymbol{x}_1\|_2, \\
\frac{\|\mathbf{A}\|_2}{\|\boldsymbol{b}_1\|_2} &\geq \frac{1}{\|\boldsymbol{x}_1\|_2},
\end{aligned}$$

or

$$\frac{1}{\|\boldsymbol{x}_1\|_2} \leq \frac{\|\mathbf{A}\|_2}{\|\boldsymbol{b}_1\|_2}.$$

Shove this into Equation (21.2)

$$\frac{\|\boldsymbol{x}_2 - \boldsymbol{x}_1\|_2}{\|\boldsymbol{x}_1\|_2} \leq \|\mathbf{A}^{-1}\|_2 \| \left(\boldsymbol{b}_2 - \boldsymbol{b}_1\right) \|_2 \left(\frac{1}{\|\boldsymbol{x}_1\|_2}\right) \leq \|\mathbf{A}^{-1}\|_2 \| \left(\boldsymbol{b}_2 - \boldsymbol{b}_1\right) \|_2 \left(\frac{\|\mathbf{A}\|_2}{\|\boldsymbol{b}_1\|_2}\right)$$

Tidy up:

$$\frac{\|\boldsymbol{x}_2 - \boldsymbol{x}_1\|_2}{\|\boldsymbol{x}_1\|_2} \leq \|\mathbf{A}\|_2 \|\mathbf{A}^{-1}\|_2 \left(\frac{\|\boldsymbol{b}_2 - \boldsymbol{b}_1\|_2}{\|\boldsymbol{b}_1\|_2}\right).$$

Definition 21.2 (Condition number) *Let* $\mathbf{A} \in \mathbb{R}^{n\times n}$. *We call*

$$\kappa(\mathbf{A}) := \|\mathbf{A}\|_2 \|\mathbf{A}^{-1}\|_2$$

the **condition number** *of* $\mathbf{A}$.

Chapter 22

Condition number, continued

Overview

Recall our results concerning the condition number: for the problem pair $\mathbf{A}\boldsymbol{x}_1 = \boldsymbol{b}_1$ and $\mathbf{A}\boldsymbol{x}_2 = \boldsymbol{b}_2$, we have

$$\frac{\|\boldsymbol{x}_2 - \boldsymbol{x}_1\|_2}{\|\boldsymbol{x}_1\|_2} \leq \kappa(\mathbf{A})\frac{\|\boldsymbol{b}_2 - \boldsymbol{b}_1\|_2}{\|\boldsymbol{b}_1\|_2}$$

where $\kappa(\mathbf{A}) = \|\mathbf{A}\|_2\|\mathbf{A}^{-1}\|_2$. In this section we examine further practical implications of this definition.

22.1 Roundoff and ill-conditioned matrices form a toxic mix

Consider again the solution $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ for a nearly-singular matrix $\mathbf{A}$. Suppose that $\boldsymbol{b}$ is determined by some subordinate numerical procedure. Because of rounding error, there can be errors in the way that $\boldsymbol{b}$ is computed, leading to a difference between the true value (call it $\boldsymbol{b}_0$), and the computed value, referred to here as $\boldsymbol{b}_0 + \delta\boldsymbol{b}$. Suppose now that the computed value $\boldsymbol{b}_0 + \delta\boldsymbol{b}$ is fed into the matrix-inversion problem. This can lead to a dramatically poor estimate for $\boldsymbol{x}$, as we now see. For definiteness, we let $\mathbf{A}\boldsymbol{x}_0 = \boldsymbol{b}_0$ denote the true solution.

Theorem 22.1 *Let $\mathbf{A} \in \mathbb{R}^{n\times n}$ be an invertible matrix and let $\boldsymbol{b}_0 \in \mathbb{R}^n$. Define $\boldsymbol{x}_0$ as the solution of the linear system*

$$\mathbf{A}\boldsymbol{x}_0 = \boldsymbol{b}_0.$$

Let $\delta\boldsymbol{b} \in \mathbb{R}^n$ be a small perturbation of $\boldsymbol{b}$ and define $\boldsymbol{x}_0 + \delta\boldsymbol{x} \in \mathbb{R}^n$ as the solution of

$$\mathbf{A}(\boldsymbol{x}_0 + \delta\boldsymbol{x}) = \boldsymbol{b}_0 + \delta\boldsymbol{b}.$$

Then,

$$\frac{\|\delta \boldsymbol{x}\|_2}{\|\boldsymbol{x}_0\|_2} \leq \kappa(\mathbf{A}) \frac{\|\delta \boldsymbol{b}\|_2}{\|\boldsymbol{b}\|_2}.$$

Proof: Form the difference

$$\mathbf{A}(\boldsymbol{x}_0 + \delta \boldsymbol{x}) - \mathbf{A}\boldsymbol{x}_0 = (\boldsymbol{b}_0 + \delta \boldsymbol{b}) - \boldsymbol{b}_0,$$

i.e.

$$\mathbf{A}\delta \boldsymbol{x} = \delta \boldsymbol{b}.$$

Beause $\mathbf{A}$ is invertible, we have

$$\delta \boldsymbol{x} = \mathbf{A}^{-1}\delta \boldsymbol{b}.$$

We compute:

$$\|\delta \boldsymbol{x}\|_2 = \|\mathbf{A}^{-1}\delta \boldsymbol{b}\|_2 \leq \|\mathbf{A}^{-1}\|_2 \|\delta \boldsymbol{b}\|_2.$$

Hence,

$$\frac{\|\delta \boldsymbol{x}\|_2}{\|\boldsymbol{x}_0\|_2} \leq \|\mathbf{A}^{-1}\delta \boldsymbol{b}\|_2 \leq \|\mathbf{A}^{-1}\|_2 \|\delta \boldsymbol{b}\|_2 \left(\frac{1}{\|\boldsymbol{x}_0\|_2}\right). \tag{22.1}$$

But $\mathbf{A}\boldsymbol{x}_0 = \boldsymbol{b}_0$, hence $\|\mathbf{A}\boldsymbol{x}_0\|_2 = \|\boldsymbol{b}_0\|_2$, and

$$\|\boldsymbol{b}_0\|_2 = \|\mathbf{A}\boldsymbol{x}_0\|_2 \leq \|\mathbf{A}\|_2 \|\boldsymbol{x}_0\|_2.$$

Hence,

$$\frac{\|\boldsymbol{b}\|_2}{\|\mathbf{A}\|_2} \leq \|\boldsymbol{x}_0\|_2,$$

and

$$\frac{1}{\|\boldsymbol{x}_0\|_2} \leq \frac{\|\mathbf{A}\|_2}{\|\boldsymbol{b}\|_2} \tag{22.2}$$

Combining Equations (22.1) and (22.2), we have

$$\frac{\|\delta \boldsymbol{x}\|_2}{\|\boldsymbol{x}_0\|_2} \leq \|\mathbf{A}^{-1}\|_2 \|\delta \boldsymbol{b}\|_2 \left(\frac{\|\mathbf{A}\|_2}{\|\boldsymbol{b}\|_2}\right),$$

or

$$\frac{\|\delta \boldsymbol{x}\|_2}{\|\boldsymbol{x}_0\|_2} \leq \|\mathbf{A}^{-1}\|_2 \left(\frac{\|\delta \boldsymbol{b}\|_2}{\|\boldsymbol{b}_0\|_2}\right),$$

hence

$$\frac{\|\delta \boldsymbol{x}\|_2}{\|\boldsymbol{x}_0\|_2} \leq \kappa(\mathbf{A}) \frac{\|\delta \boldsymbol{b}\|_2}{\|\boldsymbol{b}_0\|_2},$$

as required.

22.2 The condition number and singular matrices

Theorem 22.2 *Let $\mathbf{A} \in \mathbb{R}^n$ be 'close to singular', in the sense that*

$$\|\mathbf{A} - \mathbf{A}_0\|_2 = \epsilon,$$

where $\mathbf{A}_0$ is a singular matrix and ϵ is small and positive. Then the condition number of $\mathbf{A}$ is large in the sense that

$$\kappa(\mathbf{A}) \geq \|\mathbf{A}\|_2/\epsilon.$$

Proof: By definition,

$$\begin{aligned} \kappa(\mathbf{A}) &= \|\mathbf{A}\|_2\|\mathbf{A}^{-1}\|_2, \\ &= \|\mathbf{A}\|_2 \left(\sup_{\boldsymbol{x}\neq 0} \frac{\|\mathbf{A}^{-1}\boldsymbol{x}\|_2}{\|\boldsymbol{x}\|_2} \right), \\ &\geq \|\mathbf{A}\|_2 \left(\frac{\|\mathbf{A}^{-1}\boldsymbol{x}\|_2}{\|\boldsymbol{x}\|_2} \right), \end{aligned}$$

for any non-zero $\boldsymbol{x} \in \mathbb{R}^n$. Since $\mathbf{A}^{-1}$ is invertible, we call

$$\mathbf{A}^{-1}\boldsymbol{x} = \boldsymbol{y},$$

and $\boldsymbol{x} = \mathbf{A}\boldsymbol{y}$. Hence,

$$\kappa(\mathbf{A}) \geq \|\mathbf{A}\|_2 \left(\frac{\|\boldsymbol{y}\|_2}{\|\mathbf{A}\boldsymbol{y}\|_2} \right). \tag{22.3}$$

Now here is the clever thing: $\mathbf{A}_0$ is not invertible, so

$$\dim\left(\mathrm{im}(\mathbf{A}_0)\right) < n,$$

hence

$$\dim\left(\ker(\mathbf{A}_0)\right) \geq 1,$$

Hence, we can choose our $\boldsymbol{y}$-vector to be in the kernel of $\mathbf{A}_0$:

$$\mathbf{A}_0\boldsymbol{y} = 0.$$

Starting with Equation (22.3), we have the following string:

$$\begin{aligned} \kappa(\mathbf{A}) &\geq \|\mathbf{A}\|_2 \left(\frac{\|\boldsymbol{y}\|_2}{\|\mathbf{A}\boldsymbol{y}\|_2} \right), \\ &= \|\mathbf{A}\|_2 \left(\frac{\|\boldsymbol{y}\|_2}{\|\mathbf{A}\boldsymbol{y} - \mathbf{A}_0\boldsymbol{y}\|_2} \right). \end{aligned}$$

Hence,

$$\frac{1}{\kappa(\mathbf{A})} \leq \frac{1}{\|\mathbf{A}\|_2}\frac{\|\mathbf{A}\boldsymbol{y}-\mathbf{A}_0\boldsymbol{y}\|_2}{\|\boldsymbol{y}\|_2} \leq \frac{1}{\|\mathbf{A}\|_2}\left(\sup_{\boldsymbol{y}\neq 0}\frac{\|\mathbf{A}\boldsymbol{y}-\mathbf{A}_0\boldsymbol{y}\|_2}{\|\boldsymbol{y}\|_2}\right) = \frac{\|\mathbf{A}-\mathbf{A}_0\|_2}{\|\mathbf{A}\|_2}.$$

In other words,

$$\frac{1}{\kappa(\mathbf{A})} \leq \frac{\epsilon}{\|\mathbf{A}\|_2},$$

or

$$\kappa(\mathbf{A}) \geq \frac{\|\mathbf{A}\|_2}{\epsilon},$$

as required.

Thus, a matrix that is close to singular has a large condition number. The converse is also true: if a matrix is close to singular in a sense to be described below, then it must have a large condition number. To see why this is the case, two lemmas are required.

Lemma 22.1 *Let $\mathbf{B} \in \mathbb{R}^{n\times n}$ be an invertible matrix with spectrum*

$$\mathrm{spec}(\mathbf{B}) = \{\lambda_1, \cdots, \lambda_n\}.$$

Then,

1. *$\lambda_i \neq 0$, for each $i = 1, \cdots, n$.*
2. *$\mathrm{spec}(\mathbf{B}^{-1}) = \{\lambda_1^{-1}, \cdots, \lambda_n^{-1}\}$.*

Proof: For the first part, we have $\mathbf{B}\boldsymbol{x}_i = \lambda_i\boldsymbol{x}_i$ for each eivenvalue/eigenvector pair. Recall, $\boldsymbol{x}_i \neq 0$, by definition of an eigenvector. Hence, the only way for $\mathbf{B}\boldsymbol{x}_i$ to be equal to zero is for λ_i to be zero. But this is impossible, because the kernel of $\mathbf{B}$ is trivial, since $\mathbf{B}$ is invertible. Hence, $\lambda_i \neq 0$.

For the second part, we have

$$\mathbf{B}\boldsymbol{x}_i = \lambda_i\boldsymbol{x}_i,$$

for some eigenvector-eigenvalue pair $(\boldsymbol{x}_i, \lambda_i)$. Multiply both sides of this equation on the left by $\mathbf{B}^{-1}$:

$$\boldsymbol{x}_i = \lambda_i\mathbf{B}^{-1}\boldsymbol{x}_i,$$

and re-arrange to obtain

$$\frac{1}{\lambda_i}\boldsymbol{x}_i = \mathbf{B}^{-1}\boldsymbol{x}_i,$$

and the result is shown.

Lemma 22.2 *Let $\mathbf{A} \in \mathbb{R}^{n\times n}$ be a real invertible matrix. Then the condition number of $\mathbf{A}$ can be written as*

$$\kappa(\mathbf{A}) = \sqrt{\frac{\lambda_{\max}}{\lambda_{\min}}},$$

where $\lambda_{\min}$ denotes the **smallest** *eigenvalue of $\mathbf{A}^T\mathbf{A}$.*

Proof: From the definition of the operator norm, we already have

$$\kappa(\mathbf{A}) = \sqrt{\lambda_{\max}}\left(\sup_i |\mu_i|\right)^{1/2},$$

where the μ_i's are the eigenvalues of $(\mathbf{A}^T\mathbf{A})^{-1}$. Hence, from Lemma (22.2), we have

$$\{\mu_1, \cdots, \mu_n\} = \left\{\frac{1}{\lambda_1}, \cdots, \frac{1}{\lambda_n}\right\},$$

where we use λ_i to denote the (positive) eigenvalues of $\mathbf{A}^T\mathbf{A}$. Thus, the maximum value of μ_i is got from

$$[\text{Maximum value of } \mu_i] = 1/[\text{Minimum value of } \lambda_i] = 1/\lambda_{\min}.$$

Putting this all together, we have

$$\kappa(\mathbf{A}) = \sqrt{\frac{\lambda_{\max}}{\lambda_{\min}}},$$

and the result is shown.

From the eigenvalue relation $\mathbf{B}\boldsymbol{x} = \lambda\boldsymbol{x}$ for a generic square matrix $\mathbf{B}$, we see that a matrix fails to be invertible if it has a zero eigenvalue. For, then we would have $\mathbf{B}\boldsymbol{x} = 0$ for a nonzero vector $\boldsymbol{x}$, such that $\dim(\ker(\mathbf{B})) \geq 1$. Hence, we would have $\dim(\mathrm{im}(\mathbf{B})) < n$, implying a non-invertible matrix. We therefore make the following definition:

Definition 22.1 *We say that a matrix is 'close to singular' if the smallest eigenvalue of $\mathbf{A}^T\mathbf{A}$ is small.*

This makes sense: we would like to show that $\mathbf{A}$ itself has eigenvalues that are 'close' to zero. We let $\mathbf{A}\boldsymbol{x} = \mu\boldsymbol{x}$ and take operator norms on both sides, giving

$$|\mu| = \frac{\|\mathbf{A}\boldsymbol{x}\|}{\|\boldsymbol{x}\|} = \frac{(\mathbf{A}\boldsymbol{x}, \mathbf{A}\boldsymbol{x})^{1/2}}{\|\boldsymbol{x}\|_2} = \frac{(\mathbf{A}^T\mathbf{A}\boldsymbol{x}, \boldsymbol{x})^{1/2}}{\|\boldsymbol{x}\|_2} \geq \epsilon^{1/2}.$$

Thus, in a worst-case scenario, we would have $|\mu| = \epsilon^{1/2}$, meaining that $\mathbf{A}$ itself would be close to singuler (i.e. close to having a zero eigenvalue).

We now prove the following partial converse to Theorem 22.2:

Theorem 22.3 *Let $\mathbf{A} \in \mathbb{R}^{n\times n}$ be close to singular in the sense described in Definition 22.1. Then*

$\mathbf{A}$ *has a large condition number, in the sense that*

$$\kappa(\mathbf{A}) = \|\mathbf{A}\|_2/\epsilon^{1/2},$$

where ϵ is some small positve number.

Proof: From Lemma 22.2, we have

$$\kappa(\mathbf{A}) = \sqrt{\frac{\lambda_{\max}}{\lambda_{\min}}},$$

where $\lambda_{\min}$ denotes the **smallest** eigenvalue of $\mathbf{A}^T\mathbf{A}$. By assumption, $\lambda_{\min} = \epsilon$, where ϵ is a small positive number. Hence,

$$\kappa(\mathbf{A}) = \sqrt{\lambda_{\max}/\epsilon},$$

or

$$\kappa(\mathbf{A}) = \|\mathbf{A}\|_2/\epsilon^{1/2},$$

as required.

Thus, a matrix is close to singular if and only if it has a large condition number. We also know that large condition numbers amplify errors in matrix inversions. We must therefore be very careful when dealing with numerical models with near-singular matrices.

22.3 Estimating the condition number numerically

From the definition of the condition number, we know that

$$\kappa(\mathbf{A}) = \|\mathbf{A}\|_2\|\mathbf{A}^{-1}\|_2$$

for a square matrix $\mathbf{A}$. It would appear that to calculate the condition number numerically, we need to invert $\mathbf{A}$. However, this would be a bad idea, as such inversions can only be done reliably when the condition number is small. This leads to circular reasoning: to invert a matrix numerically we need to know the condition number, for which we need to invert the matrix and so on. Clearly, we need a shortcut to find $\kappa(\mathbf{A})$. The answer lies in the alternative definition

$$\kappa(\mathbf{A}) = \sqrt{\frac{\lambda_{\max}}{\lambda_{\min}}}.$$

We shall use this shortly to estimate $\kappa(\mathbf{A})$. Before doing so, we need to make a detour into the world of numerical computation of eigenvalues.

Chapter 23

Eigenvalues – the power method

Overview

23.1 The idea

In this section we consider symmetric matrices $\mathbf{B} \in \mathbb{R}^{n\times n}$. However, the method can be extended to arbitrary square matrices (the proof for the latter relies on the Jordan decomposition). We choose a random vector $\boldsymbol{x}_0$. By the spectral theorem, $\mathbf{B}$ has an eigenbasis $\{\boldsymbol{x}_i\}_{i=1}^n$, with

$$\mathbf{B}\boldsymbol{x}_i = \lambda_i \boldsymbol{x}_i, \qquad (\boldsymbol{x}_i, \boldsymbol{x}_j) = \delta_{ij}.$$

We decompose $\boldsymbol{x}_0$ in terms of this basis:

$$\boldsymbol{x}_0 = \alpha_1 \boldsymbol{x}_1 + \cdots + \alpha_n \boldsymbol{x}_n.$$

For definiteness, we focus on the non-degenerate case, where $\lambda_i = \lambda_j \implies i = j$. We let $i = 1$ label the maximal eigenvalue. If $\boldsymbol{x}_0$ is selected using a random-number generator that is based on a continuous distribution, then $\mathbb{P}(\alpha_1 = 0) = 0$, and $\boldsymbol{x}_0$ almost surely contains a contribution from the maximal eigenvector.

Exercise 23.1 *Estimate the probability in a numerical calculation that $\alpha_1 = 0$, to machine precision.*

We operate repeatedly on $\boldsymbol{x}_0$ with the matrix $\mathbf{B}$:

$$\mathbf{B}^k \boldsymbol{x}_0 = \alpha_1 \lambda_1^k \boldsymbol{x}_1 + \cdots + \alpha_n \lambda_n^k \boldsymbol{x}_n.$$

Since, by assumption, $|\lambda_1| > |\lambda_i|$ for all $i \neq 1$, we have

$$|\lambda_1|^k \gg |\lambda_i| \qquad \forall i \neq 1, \text{ as } k \to \infty.$$

Hence,

$$\mathbf{B}^k \boldsymbol{x}_0 \sim \alpha_1 \lambda_1^k \boldsymbol{x}_1, \text{ as } k \to \infty. \tag{23.1}$$

We therefore have an iterative method to compute the maximal norm-one **eigenvector**:

$$\boldsymbol{b}^{(k+1)} = \frac{\mathbf{B}\boldsymbol{b}^{(k)}}{\|\mathbf{B}\boldsymbol{b}^{(k)}\|_2}, \qquad k = 0, 1, \cdots, \qquad \boldsymbol{b}^{(0)} = \boldsymbol{x}_0.$$

We have

$$\lim_{k\to\infty} \boldsymbol{b}^{(k)} = \boldsymbol{x}_1,$$

since

1. By Equation (23.1) the vector $\boldsymbol{b}^{(k)}$ points in the direction $\boldsymbol{x}_1$, as $k \to \infty$;
2. Each vector $\boldsymbol{b}^{(k)}$ has norm one.

The maximal eigenvalue is then computed as

$$\lambda_1 = \lim_{k\to\infty} \left(\boldsymbol{b}^{(k)}, \mathbf{B}\boldsymbol{b}^{(k)}\right).$$

Below is a sample Matlab code that implements this **power method**.

```
function [lambda,x1,B]=power_method(n)

tol=1e-8;

% Input matrix whose leading eigenvalue is to be determined.  Let's keep it
% general and make it a random matrix with entries between 0 and 1.
% Here n is user-supplied at the command line.
%
B=rand(n,n);

% Focus on symmetric matrices if one wishes:
B=(B+B')/2;

% Initial guess:
x0=rand(n,1);

bk=x0;
lambda_old=dot(bk,B*bk);
```

```

% norm(v) returns the L2 norm of a vector v.
bk=B*bk/norm(B*bk);
lambda=dot(bk,B*bk);

count=1;

while(abs(lambda_old-lambda)>tol)

    lambda_old=lambda;
    bk=B*bk/norm(B*bk);
    lambda=dot(bk,B*bk);

    count=count+1;

end

x1=bk;

display(['converged in ',num2str(count),' iterations'])

% ******************************************************************
% ******************************************************************
% Compare with Matlab's built-in methods:

% Compute ALL eigenvalues and eigenvectors.
% Eigenvalues -- along the diagonal of the matrix D.
% Eigenvectors -- the columns of the matrix V are the eigenvectors.

[V,D]=eig(B);

% Convert D into an vector of eigenvalues.
dc=0*(1:n);

for i=1:n
    dc(i)=D(i,i);
end

% Pick out the eigenvalue with the largest absolute value.

[~,ix]=max(abs(dc));
lambda_true=dc(ix);

% Pick out the corresponding eigenVECTOR.
x1_true=V(:,ix);
```

```

display('true eigenvalue:')
display(lambda_true)

display('true eigenvector:')
display(x1_true)


end
```

sample_matlab_codes/power_method.m

23.2 Matlab's built-in functions

The sample code given above shows how to access Matlab's built-in functions to do eigenvalue-eigenvector calculations. As usual, these are much better than our own humble attempts. They are based on the **QR** algorithm, and returns **all** the eigenvalues and eigenvectors of a matrix (not just the leading eigenvalue-eigenvector pair). Assuming that a square matrix A (not necessarily symmetric) is defined on the command line, to compute the **eigenvalues** alone, one types

```
dc=eig(A);
```

Here, the eigenvalues are returned in an array called `dc`. If one requires the eigenvalues and eigenvectors, one would type

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

Here `D` is a diagonal square matrix whose diagonal elements contain the eigenvalues; these can be converted into a simple array using a 'for' loop:

```
n=length(D);


dc=0*(1:n);


for i=1:n
    dc(i)=D(i,i);
end
```

If i labels the i^{th} eigenvalue, then the corresponding eigenvector `vi` is accessed via the command

```
vi=V(:,i);
```

23.3 Back to the condition number

We revisit the condition number of a square matrix $\mathbf{A}$, recalling that it can be written as

$$\kappa(\mathbf{A}) = \|\mathbf{A}\|_2\|\mathbf{A}^{-1}\|_2 = \|\mathbf{A}\|_2\|\mathbf{A}^{-1}\|_2.$$

We want to compute this without computing the inverse of $\mathbf{A}$. Certainly, there is no problem in estimating $\|\mathbf{A}\|_2 = \sqrt{\lambda_{\max}}$ – we simply use the power method. To get a 'rough' estimate of $\|\mathbf{A}\|_2$ we might only do a few iterations of the power method (cutting down on the number of iterations is especially important for large matrices).

To compute $\lambda_{\min}$, the smallest eigenvalue of $\mathbf{B} := \mathbf{A}^T\mathbf{A}$, we instead compute the largest eigenvalue of $\mathbf{B}^{-1}$. We use a variant of the power method, with the iteration

$$\boldsymbol{b}^{(k+1)} = \frac{\mathbf{B}^{-1}\boldsymbol{b}^k}{\|\boldsymbol{b}^{(k)}\|_2}.$$

This still looks like a matrix inversion is needed. However, we re-write this equation as

$$\mathbf{B}\boldsymbol{b}^{(k+1)} = \frac{\boldsymbol{b}^{(k)}}{\|\boldsymbol{b}^{(k)}\|_2}, \tag{23.2}$$

and this can be solved for $\boldsymbol{b}^{(k+1)}$ without actually inverting $\mathbf{B}$ (e.g. Jacobi or Gauss-Seidel methods – covered in more advanced Computational Science modules). We would again iterate Equation (23.2) a few times and estimate $\lambda_{\min}$, giving

$$\kappa(\mathbf{A}) \approx \left[\lambda_{\max}^{\text{est}}\lambda_{\min}^{\text{est}}\right]^{1/2}.$$

If our estimate of $\kappa(\mathbf{A})$ is large, we know that doing a full matrix inversion of $\mathbf{A}$ might be dodgy.

Common Programming Error:

Estimates for the condition number are norm-dependent. In this module, we have used the L^2 operator norm; other operator norms will produce different estimates for the condition number.

However, as a general rule, if the condition number appears large in the L^2 sense, it probably is large, and care is needed in doing matrix inversions.

23.4 Google

To rank pages in a web search according to importance, Google finds the leading eigenvalue-eigenvector pair of a huge matrix (the ‘search matrix’). The power method is used for this calculation. Rather alarmingly, a multi-billion dollar company has at its heart a concept that can be encapsulated in 72 lines of Matlab.

Chapter 24

Fitting polynomials to data

Overview

In this chapter we learn a method to 'fit' a polynomial to a dataset. This is an extremely useful idea that finds applications in all areas of science. The idea is to start with a measured set of data relating an independent variable x to a dependent variable y, and find a 'nice' curve connecting these variables.

24.1 The idea

Suppose in an experiment we want to measure the relationship between an independent variable x which we freely vary, and a dependent variable y, which is affected by changes in x. We would create a data set

$$\{(x_i, y_i)\}_{i=1}^{N},$$

consisting of N (hopefully independent) measurements of the phenomenon. We would plot these on a graph such as Figure 24.1. We want to find a mathematical description of the blue curve – the curve that gives an analytical relationship for the 'best' approximation of the relationship between x and y.

To find this curve, we **assume** that it can be described by a polynomial

$$f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_Mx^M$$

where M is some degree to be determined (M will of course be different from N and must satisfy

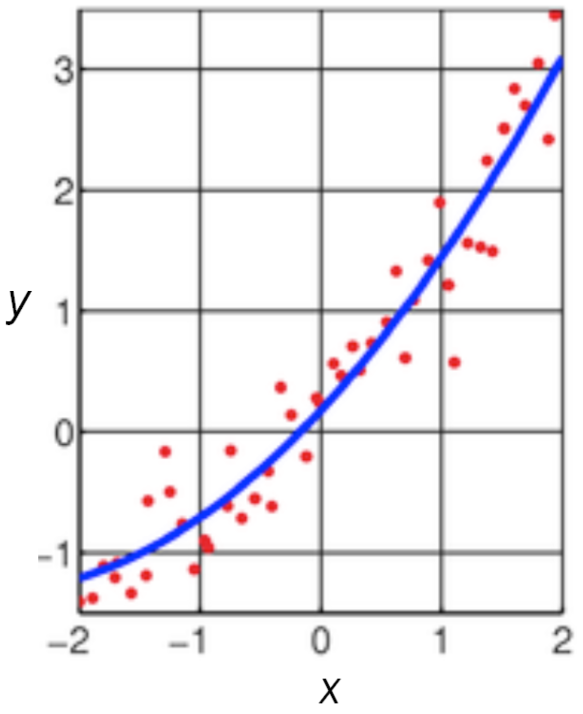


Figure 24.1: The idea of a least-squares approximation.

$M \leq N$). We fix the coefficients a_i by minimizing the square distance

$$D = \sum_{i=1}^{N} \left[f(x_i) - y_i \right]^2 . \tag{24.1}$$

24.2 The minimization

We minimize D in Equation (24.1) by solving

$$\frac{\partial D}{\partial a_i}, \qquad i = 1, \cdots, M.$$

We have

$$\begin{aligned} \frac{\partial D}{\partial a_n} &= \frac{\partial}{\partial a_n} \sum_{i=1}^{N} \left[a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_n x_i^n + \cdots + a_M x_i^M \right]^2 , \\ &= 2 \sum_{i=1}^{N} \left[a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_n x_i^n + \cdots + a_M x_i^M - y_i \right] x_i^n . \end{aligned}$$

Thus,

$$\frac{\partial D}{\partial a_n} = 0 \implies \sum_{i=1}^{N} \left[a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_n x_i^n + \cdots + a_M x_i^M\right] x_i^n = \sum_{i=1}^{N} y_i x_i^n, \qquad n = 1, \cdots, M.$$

In this equation, the x_i's and the y_i's are known from measurements and the a-coefficients are unknown. We therefore re-write this equation in matrix form and solve the a-coefficients:

$$\begin{aligned}
\sum_{i=1}^{N} \left[a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_n x_i^n + \cdots + a_M x_i^M\right] &= \sum_{i=1}^{N} y_i, \\
\sum_{i=1}^{N} \left[a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_n x_i^n + \cdots + a_M x_i^M\right] x_i &= \sum_{i=1}^{N} y_i x_i, \\
\sum_{i=1}^{N} \left[a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_n x_i^n + \cdots + a_M x_i^M\right] x_i^2 &= \sum_{i=1}^{N} y_i x_i^2, \\
\vdots \qquad &= \vdots \\
\sum_{i=1}^{N} \left[a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_n x_i^n + \cdots + a_M x_i^M\right] x_i^M &= \sum_{i=1}^{N} y_i x_i^M,
\end{aligned}$$

Re-write again:

$$\begin{aligned}
a_0 \left(\sum_i 1\right) + a_1 \left(\sum_i x_i\right) + \cdots + a_M \left(\sum_i x_i^M\right) &= \sum_{i=1}^{N} y_i, \\
a_0 \left(\sum_i x_i\right) + a_1 \left(\sum_i x_i^2\right) + \cdots + a_M \left(\sum_i x_i^{M+1}\right) &= \sum_{i=1}^{N} y_i x_i, \\
a_0 \left(\sum_i x_i^2\right) + a_1 \left(\sum_i x_i^3\right) + \cdots + a_M \left(\sum_i x_i^{M+2}\right) &= \sum_{i=1}^{N} y_i x_i^2, \\
\vdots \qquad &= \vdots \\
a_0 \left(\sum_i x_i^M\right) + a_1 \left(\sum_i x_i^{M+1}\right) + \cdots + a_M \left(\sum_i x_i^{2M}\right) &= \sum_{i=1}^{N} y_i x_i^M,
\end{aligned}$$

This is an $(M+1)\times(M+1)$ system for the unknown a-coefficients. We re-write it in a compact notation, in matrix terms:

$$\underbrace{\begin{pmatrix} S_0 & S_1 & \cdots & S_M \\ S_1 & S_2 & \cdots & S_{M+1} \\ \vdots & & & \vdots \\ S_M & S_{M+1} & \cdots & S_{2M} \end{pmatrix}}_{=\mathbf{M}} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_M \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_M \end{pmatrix}, \qquad (24.2)$$

where

$$S_n = \sum_i x_i^n, \qquad n = 0, \cdots, M,$$

and

$$b_n = \sum_i y_i x_i^n, \qquad n = 0, \cdots, M.$$

Provided $\mathbf{M}$ is invertible, Equation (24.2) is solved and the a-coefficienits are obtained:

$$\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_M \end{pmatrix} = \mathbf{M}^{-1} \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_M \end{pmatrix}. \qquad (24.3)$$

Exercise 24.1 *Write a Matlab function to implement the algorithm in Equations* (24.2)–(24.3). *It should read in vectors* x *and* y *from the command line, as well as a trial polynomial degree* M. *See below for how to generate test vectors to validate your code.*

Writing test vectors

Suppose we have a code that implements the algorithm in Equations (24.2)–(24.3), and that we want to test it. We would create an array of equally-spaced x-values (say):

```
x=0:0.1:20;
```

We would then create a test vector `y` consisting of a definite signal (some well-defined curve), as well as some random noise, taken (say) from a normal distribution:

```
    sd=10;    y=2+0.5*x-2*(x.^2)+0.1*(x.^3)+sd*rand(1,length(x));
```
(24.4)

This creates a vector y that consists of the well-defined curve

$$2 + 0.5x - 2x^2 + 0.1x^3,$$

superimposed on which are random perturbations drawn from a **normal** distrubution of standard deviation `sd=10` and mean zero (to visualize, plot (x, y) in Matlab!).

Suppose that we have a matlab function whose first line is

```
a_vec=least_squares(xin,yin,M);
...
```

A priori, we do not know what is the best value for M. We would make a few attempts, by calling our function multiple times as follows:

```
a_vec1=least_squares(xin,yin,1);
a_vec2=least_squares(xin,yin,2);
a_vec3=least_squares(xin,yin,3);
a_vec4=least_squares(xin,yin,4);
```

We would then calculate the square distance D between the polynomial curve and the data in each case. There is a simple tool in Matlab called **polyval**: given an array

```
a_vec=[a0,a0, ... ,aM]
```

and an input vector `x`, typing

```
yM=polyval(a_vec,x)
```

returns the polynomial $\sum_{i=0}^{M} a_i x^i$ evaluated at the x-points in the array `x`. Thus, we would type

```
y1=polyval(a_vec1,x);
y2=polyval(a_vec2,x);
y3=polyval(a_vec3,x);
y4=polyval(a_vec4,x);
```

We compute the square distance from the true curve in each case by typing

```
diff1=norm(y1-y)^2;
diff2=norm(y1-y)^2;
diff3=norm(y1-y)^2;
diff4=norm(y1-y)^2;
```

For our curve of best fit, we choose the M-value that minimizes `diff` (the **residual**). In practice, there can be a range of M-values that produce a similar residual. As a general rule of thumb, it is best to choose the minimum possible M-value, for the general philosophical reason that physical laws tend to be simple, and hence, the simplest possible approximate solution should be chosen.

Immoral and reprehensible scientific practice:

Many cases of scientific misconduct have arisen where in the absence of true data, data has been created or 'falsified' as in Equation (24.4).
We used Equation (24.4) to test a code; bad people have used similar equations to write bogus scientific papers and accrue prestige and promotions. They are usually found out.
If you do this, I will come after you with a big stick.

24.3 A built-in function

Finally, instead of our notional code 'least_squares.m' to compute the algorithm in Equations (24.2)–(24.3), there is of course a built-in Matlab function. One would type

```
a_vec=polyfit(xin,yin,M);,
```

where `xin` and `yin` are as before and M is the trial degree of the least-squares polynomial. As before, the least-squares polynomial is reconstructed via

```
yM=polyval(a_vec,x);
```

Chapter 25

Random-number generation

Overview

Can a deterministic machine such as a computer generate truly random numbers? The answer is no! In fact, what generally passes for 'random' numbers on a computer are sequences of numbers generated by an iterative map with nice properties. Such sequences – generated using a deterministic map are called 'quasi-random'.

Random numbers are required frequently in computation:

- Creating random matrices for testing codes.
- Creating initial codnitions for a simulation. Random initial conditions remove any 'bias' that could creep into a simulation from user-selected initial conditions.
- Data encryption.
- Choosing the order in which experiemntal data is analyzed, drawing subjects from a wider pool for a test in social science, thereby eliminating selection bias.
- Picking the lottery numbers.
- Solving stochastic differential equations in a pathwise manner (for stock-market modelling, fluids simulations &c&c).

25.1 Matlab's built-in random-number generator

Typing

```
r=rand;
```

one obtains a number $r \in (0,1)$. The number r is a realization of a random variable with a uniform distribution in $(0,1)$. In other words, the probability that a realization random variable r is between x and $x + \mathrm{d}x$ is as follows:

$$\mathbb{P}(x < r < x + \mathrm{d}x) = \mathrm{d}x \times \begin{cases} 1, & \text{if } 0 < x < 1, \\ 0, & \text{otherwise.} \end{cases}$$

The realizations r of the random variable are drawn from a long list of numbers called the **global stream**. One can find out certain properties of the global stream by typing `RandStream.list`; on my computer I received the following answer:

```
>> RandStream.list

The following random number generator (RNG) algorithms are available:

mcg16807:     Multiplicative congruential generator,
with multiplier 7^5, modulo 2^31-1
mlfg6331_64:  Multiplicative lagged Fibonacci generator,
with lags 63 and 31, 64 bit (supports parallel streams)
mrg32k3a:     Combined multiple recursive generator
(supports parallel streams)
mt19937ar:    Mersenne Twister with Mersenne prime 2^19937-1
shr3cong:     SHR3 shift-register generator summed with CONG
linear congruential generator
swb2712:      Modified Subtract-with-Borrow generator, with lags 27 and 12
```

To find out which method is currently being used to create the global stream, one types `stream = RandStream.getDefaultStream`; on my computer I got

```
>> stream = RandStream.getDefaultStream

stream =
mt19937ar random stream (current default)
             Seed: 0
         RandnAlg: Ziggurat
```

Here 'mt19937ar' refers to something called the 'Mersenne twister' algorithm; in this chapter we discuss a simpler algorithm which has some of the same features the Mersenne twister.

25.2 The Linear congruential generator

This is an iterative map for generating a sequence of integers $I_1, I_2, \cdots$, each between $[0, \mathrm{RANDMAX}]$ according to the following algorithm:

$$I_{j+1} = (aI_j + c) \bmod m, \tag{25.1}$$

where

- $m > 0$ is the modulus,
- a is the multiplier, $0 \le a < m$,
- c is the increment, $0 \le c < m$.

The initial value I_0 is called the **seed**. Here $\bmod m$ means 'the remainder upon division by m'.

The $\bmod m$ in Equation (25.1) 'folds back' an integer I larger than m into the interval $[0, m-1]$. Thus, the maximum number in the I-sequence is $m-1$, hence

$$\mathrm{RANDMAX} \le m - 1.$$

For a given m, suppose that a and c are chosen very cleverly so that for the first m iterations, the I-numbers are all different. Because of the $\bmod m$ in Equation (25.1), the next I-number must be come from the list of numbers already genrated, and the list will begin to repeat itself.

Theorem 25.1 *The linear congruential algorithm* (25.1) *is periodic, with period at most m.*

This is bad – we were looking for a completely random sequence of numbers, not a periodic list! The way out of course is to make m extremely large, so that it takes a long time before one notices the periodicity.

Theorem (25.1) says the period is 'at most' m – for very bad choices of a and c the period can be much smaller, which again is disastrous. The idea therefore is to choose m, a, and c in such a way that a huge chunk of the I-sequence must be taken before any repetition is observed.[1]

[1]As in the episode of 'Numbers' Series 3, 'Traffic', see `http://numb3rs.wolfram.com/303/demonstrations.html`

The linear congruential algorithm has all sorts of other drawbacks – see *Numerical Recipes in C* (Chapter 7) for a complete discussion. It would appear to be the 'Euler method of random-number generators' – good for pedagogical instruction, but little more. I would not trust my credit card details to an encryption scheme based on this algorithm! Moreover, I would not try to make my own linear congruential genearator without studying a bit of number theory and reading Knuth's book (Volume 2, Chapter 1).

Random numbers in a interval

To get (pseudo) random (pseudo) real numbers in the interval $(0, 1)$, one simply takes

$$r = I_i/(\mathrm{RANDMAX} + 1).$$

25.3 Seeding the random-number generator

One can have the computer choose the initial condition I_0. This can add a (desirable) amount of uncertainty to the outcome of the sequence (25.1). On the other hand, often one wishes to repeat the same simulation over and over again (e.g. for validation purposes), and having a different seed each time will lead to different outcomes. For that reason, RNGs typically allow one to fix the seed. There is a procedure for doing this in Matlab, for any of the RNG algorithms available.

To do this, we create a **new stream** of random numbers based on whatever Matlab's default RNG is, together with the seed 1. In Matlab versions 7.10 and later, this is done as follows:

```
% Check which random stream is accessed:
stream = RandStream.getDefaultStream;
% Grab method into a new string:
method=stream.Type;

% Create a new stream of random numbers:

newstream= RandStream(method,'seed',1);
```

From now on, one must be careful to draw the random numbers from the new stream. This can be done by passing an additional argument to commands such as `rand`, e.g.

```
r=rand(newstream)
```

A further pitfall is the following:

> **Common Matlab Programming Error:**
>
> Seeding the random number generator at lots of points in a code. It should only be seeded at the very top of a code. Otherwise, the sequence of numbers will repeat with a very short period!

In other situations, it is desirable to have uncertainty in the outcome of the pseudorandom sequence, but for that uncertainty to be controlled by the user. Computer nerds like to this by using 'Unix Epoch time' as the seed. This is the number of seconds that have elapsed since Midnight 1 January 1970 (really, it should be midnight Coordinated Universal Time (UTC)). It is computed in Matlab as

```
S=etime(clock,datevec('01/01/1970'));
```

As before, we create a new stream of random numbers based on whatever Matlab's default RNG is, together with the seed S:

```
% Check which random stream is accessed:
stream = RandStream.getDefaultStream;
% Grab method into a new string:
method=stream.Type;

% Create a seed using Unix Epoch time:

S=etime(clock,datevec('01/01/1970'));

% Create a new stream of random numbers:

newstream= RandStream(method,'seed',S);
```

Again, random numbers must now be drawn from the new stream, e.g. `r=rand(newstream);`.

Matlab code

A practical Matlab example to draw numbers for the Irish National lottery (integers $1, 2, \cdots, 45$).

```
function [rvec]=lotto()

% Check which random stream is accessed:
stream = RandStream.getDefaultStream;
display('default stream:')
display(stream)
% Grab method into a new string:
method=stream.Type;

% Create a seed using Unix Epoch time:

S=etime(clock,datevec('01/01/1970'));

% Create a new stream of random numbers using the following parameters:
% Method: Matlab's default.
% Seed: Unix Epoch time.

newstream= RandStream(method,'seed',S);
display('custom stream:')
display(newstream)


% The following implementation of the built-in function "randi"
% draws the desired numbers from a uniform distribution into an array.
%
% Syntax:
% rvec=randi(streamname,toprange,nrow,ncol);
%
% 1.  The first argument, streamname is optional.
%     We are entering our custom stream as an argument.
% 2.  The second argument, toprange is the maximum integer drawn.  The
%     minimum integer is by default set to one.
% 3.  The third argument is the number of rows in the output array.
% 4.  the fourth arguemnt is the number of columns in the same array.

rvec = randi(newstream,45,6,1);
```

sample_matlab_codes/lotto.m

Exercise 25.1 *Explain why this is a good method for assigning 'quick-picks' to participants in the National Lottery, but would be a very bad way to conduct the actual draw.*

25.4 Matlab's built-in functions – Other aspects

We have already seen in Section 25.3 that the basic RNG in Matlab can be modified in (at least) two ways:

- Selecting a new seed;
- Using extensions to `rand` to generate random integers instead of random (pseudo) real numbers in $(0,1)$. However, these random numbers are always drawn from a **uniform distribution** – numbers within range are all equally likely to be drawn, whether they are big or small, positive or negative.

The Matlab family of RNGs can be extended further to generate random numbers from non-uniform distributions. For example,

```
r=randn(10000,1)
```

is a random vector with 10,000 elements, all of which are drawn from the standard normal distributioni of mean 0 and standard deviation 1. The default stream is used.

Just how normal this vector is can be examined by drawing a histogram:

```
hist(r,50);
```

Here, the second argument is the number of **bins**, or the number of intervals in the discrete histogram. This can be chosen to make the graph as pretty as possible. We can do better by comparing this numerical distribution to the true normal distribution

$$P(x)=\frac{1}{\sqrt{2\pi}}\mathrm{e}^{-x^2/2}.$$

Create a new vector of x-points:

```
xx=-4:0.1:4;
```

Create a new histrogram with bins centred at the `xx`-points:

```
yy=hist(r,xx);
```

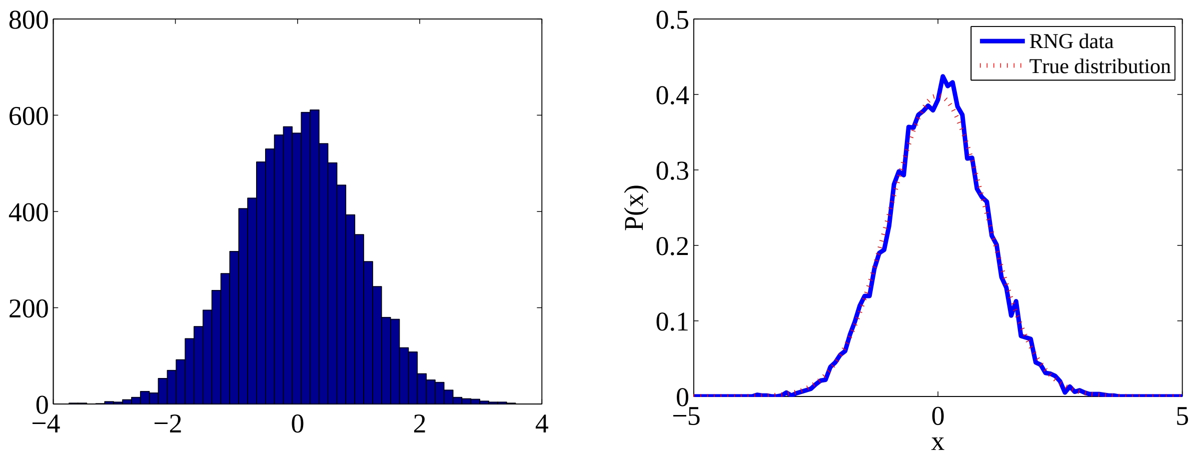
Normalize this new vector to be a PDF:

```
yy=yy/(sum(yy)*(xx(2)-xx(1)));
```

Plot and compare!

```
plot(xx,yy,xx,(1/sqrt(2*pi))*exp(-xx.^2/2))
```

The results are impressive (Figure 25.1)!



(a) (b)

Figure 25.1: Matlab example showing normally-distributed data from an RNG calculation

Appendix A

Calculus theorems you should know

Theorem A.1 (Triangle Inequality) *Let x and y be real numbers. Then*

$$|x+y| \leq |x| + |y|.$$

This result extends (e.g. by induction) to n real numbers $x_1, \cdots, x_n$:

$$\left|\sum_{i=1}^{n} x_i\right| \leq \sum_{i=1}^{n} |x_i|.$$

Theorem A.2 (Intermediate value theorem) *Let $f : [a,b] \to \mathbb{R}$ be a continuous real-valued function, with $f(a) < f(b)$. Then for each real number u with $f(a) < u < f(b)$, there exists at least one value $c \in (a,b)$ such that $f(c) = u$.*

Theorem A.3 (Mean-value theorem) *Let f be a continuous function on the closed interval $[a,b]$ with $a \leq b$, and let f be differentiable on the open interval (a,b). Then there exists a point $c \in (a,b)$ such that*

$$f'(c) = \frac{f(b)-f(a)}{b-a}.$$

Theorem A.4 (Mean-value theorem for integrals) *Let f be a continuous function on the closed interval $[a,b]$ with $a \leq b$. Then there exists a point $c \in (a,b)$ such that*

$$f(c) = \frac{1}{b-a}\int_a^b f(x)\,\mathrm{d}x.$$

Theorem A.5 (Taylor's Remainder Theorem) *Let f be a continuous function on the closed interval $[a,b]$ with $a \leq b$, and let f be $n+1$ times differentiable on the open interval (a,b). Then there exists a point $\xi \in [a,b]$ such that*

$$f(a) = f(a) + f'(a)(b-a) + \tfrac{1}{2}f''(a)(b-a)^2 + \cdots + \frac{1}{n!}f^{(n)}(a)(b-a) + \tfrac{1}{(n+1)!}f^{(n+1)}(\xi)(b-a)^{n+1}.$$

The term

$$R_n(b) = \tfrac{1}{(n+1)!}f^{(n+1)}(\xi)(b-a)^{n+1} \tag{A.1}$$

is called the **remainder**.

Theorem A.6 (Taylor's Theorem) *Let f be a continuous function on the closed interval $[a,b]$ with $a \leq b$, and (1) let f be infinitely many times differentiable on the open interval (a,b). If, furthermore, the following condition holds (condition (2)):*

$$\lim_{n\to\infty} R_n(b) = 0, \tag{A.2}$$

then

$$f(b) = \sum_{p=0}^{\infty} \tfrac{1}{p!}f^{(p)}(a)(b-a)^p. \tag{A.3}$$

Note the pair of conditions (1) and (2) in Theorem (A.6). Condition (1) does not by itself guarantee that an infinite taylor series of the kind (A.3) exists; consider for example the Taylor series of x centred at zero for the function

$$f(x) = \begin{cases} 0, & \text{if } x = 0, \\ \mathrm{e}^{-1/x^2}, & \text{otherwise.} \end{cases}$$

The Taylor series of $f(x)$ centred at 0 is zero: for $x \neq 0$ we have a Taylor series

$$f_T(x) = 0,$$

which does not agree with the generating function

$$f(x) = \mathrm{e}^{-1/x^2};$$

condition (2) is therefore important.

Appendix B

Facts about Linear Algebra you should know

Overview

In this appendix, let V be a real vector space with dimension $n < \infty$, and equipped with a scalar product

$$\begin{aligned} (\cdot|\cdot) : V \times V &\to \mathbb{R} \\ \boldsymbol{x}, \boldsymbol{y} &\mapsto (\boldsymbol{x}|\boldsymbol{y}). \end{aligned}$$

B.1 Orthogonality

- Two vectors $\boldsymbol{x}$, $\boldsymbol{y}$ are said to be orthogonal if $(\boldsymbol{x}|\boldsymbol{y}) = 0$.
- The set $\{\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n\}$ is called a **basis** for V if
  - $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n$ are linearly independent;
  - $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n$ span V.

  Thus, for any $\boldsymbol{x} \in V$, there exist real scalars $\alpha_1, \cdots, \alpha_n$ such that

  $$\boldsymbol{x} = \sum_{i=1}^{n} \alpha_i \boldsymbol{x}_i.$$

- A basis $\{\boldsymbol{x}_i\}_{i=1}^n$ for V is said to be **orthonormal** if

  $$(\boldsymbol{x}_i|\boldsymbol{x}_j) = \delta_{ij}.$$

Given an orthonormal basis $\{\boldsymbol{x}_i\}_{i=1}^n$, we have, for arbitrary $\boldsymbol{x} \in V$,

$$\begin{aligned}
\boldsymbol{x} &= \sum_{i=1}^n \alpha_i \boldsymbol{x}_i, \\
(\boldsymbol{x}_j|\boldsymbol{x}) &= \sum_{i=1}^n \alpha_i (\boldsymbol{x}_j|\boldsymbol{x}_i), \\
(\boldsymbol{x}_j|\boldsymbol{x}) &= \sum_{i=1}^n \alpha_i \delta_{ij},
\end{aligned}$$

hence

$$\alpha_j = (\boldsymbol{x}_j|\boldsymbol{x}), \qquad \boldsymbol{x} = \sum_{j=1}^n (\boldsymbol{x}_j|\boldsymbol{x})\boldsymbol{x}_j. \tag{B.1}$$

In Quantum Mechanics, Equation (B.1) is called the **completeness relation**.

B.2 The Spectral Theorem

Throughout this section, let $V = \mathbb{R}^n$. The **usual basis** means

$$\begin{aligned}
\boldsymbol{e}_1 &= (1, 0, 0, \cdots, 0, 0), \\
\boldsymbol{e}_2 &= (0, 1, 0, \cdots, 0, 0), \\
\vdots &= , \\
\boldsymbol{e}_n &= (0, 0, 0, \cdots, 0, 1).
\end{aligned}$$

An arbitrary vector in $\mathbb{R}^n$ is written as a Cartesian n-tuple, $\boldsymbol{x} = (x_1, \cdots, x_n)^T$, which can be written in terms of the usual basis as

$$\boldsymbol{x} = \sum_{i=1}^n x_i \boldsymbol{e}_i.$$

In this section, we are interested in real, square $(n \times n)$ symmetric matrices; let $\mathbf{A}$ be such a matrix:

$$\begin{aligned}
\mathbf{A} : \mathbb{R}^n &\rightarrow \mathbb{R}^n, \\
\boldsymbol{x} &\mapsto \mathbf{A}\boldsymbol{x},
\end{aligned}$$

such that $(\mathbf{A}\boldsymbol{x})_i = \sum_{j=1}^n A_{ij} x_j$. The symmetricness of $\mathbf{A}$ means that

$$A_{ij} = A_{ji}.$$

We have a simple lemma:

Lemma B.1 *The eigenvalues of* $\mathbf{A}$ *are real, and eigenvectors corresponding to distinct eigenvalues are orthogonal.*

A deeper result is the following:

Theorem B.1 *The eigenvectors of* $\mathbf{A}$ *span* $\mathbb{R}^n$ *and can be chosen to form an orthonormal set.*

This is a special case of the celebrated **spectral theorem** – the crowning achievement of Linear Algebra.