CS370

Course Notes for Numerical Computation

Fall 2024

Christopher Batty, Toshiya Hachisuka

UNIVERSITY OF WATERLOO

FACULTY OF MATHEMATICS
DAVID R. CHERITON SCHOOL OF COMPUTER SCIENCE

September 4, 2024

Contributions to these notes have been made
by past and present instructors for CS 370.

The support of the Mathematics Endowment Fund,
(MEF) in preparing these notes, is gratefully acknowledged.

We also acknowledge the MEF for funding the development of the
Forensic Science content of the course (2017).

Contents

1 Floating Point Number Systems **9**

1.1 Pitfalls in Floating Point Computation 9

1.1.1 Example 1 9

1.1.2 Example 2 10

1.2 Introduction to Floating Point Numbers 11

1.2.1 Real Numbers and Normalized Digital Expansions 11

1.2.2 A Floating Point Number Format 12

1.2.3 A Note on IEEE Floating Point Standards 13

1.3 Absolute and Relative Error 13

1.4 Properties of Floating Point Numbers 14

1.4.1 Relating Real and Floating Point Numbers 14

1.4.2 Mathematical Relationship Between $x \in \mathbb{R}$ and $fl(x) \in \mathrm{F}$ 15

1.4.3 Arithmetic Operations Under Floating Point 17

1.5 Round-off Error Analysis and Cancellation 17

1.6 Conditioning and Stability 20

1.7 A Stability Analysis 21

1.8 Exercises for Floating Point Numbers 22

2 Interpolation **25**

2.1 Polynomial Interpolation 26

2.1.1 Linear Equations: Vandermonde Systems 26

2.1.2 Lagrange form 27

2.2 Piecewise Polynomial Interpolation 28

2.3 Piecewise Hermite Interpolation 30

2.4 Spline Interpolants 31

2.5 Efficient Computation of Splines 33

2.6 Spline Energy 36

2.7 B-Splines and Bézier Curves 37

2.7.1 Bernstein Polynomials 37

2.7.2 Bézier Curves 38

2.7.3 B-spline Curves 39

2.7.4 Exercises for Interpolation 40

3 Planar Parametric Curves **43**
3.1 Interpolating Curve Data by a Parametric Curve . . . 44
3.2 Graphics for Handwriting . . . 45

4 Differential Equations **47**
4.1 Introduction . . . 47
4.1.1 Other Differential Equations . . . 50
4.2 Approximating Methods . . . 51
4.2.1 The Forward Euler Method . . . 52
4.2.2 Discrete Approximations . . . 56
4.2.3 The Trapezoidal and Modified Euler Methods . . . 57
4.2.4 Runge-Kutta Methods . . . 61
4.3 Global vs Local Error . . . 61
4.4 Practical Issues . . . 62
4.4.1 Time Step Control I: Basic Principles . . . 64
4.5 Convergence . . . 66
4.5.1 Truncation Error of the Forward Euler Method . . . 66
4.6 Stability . . . 69
4.7 Conversion of High Order ODEs to First Order Systems . . . 74
4.8 Stiff Differential Equations . . . 75
4.9 Exercises for ODEs . . . 76

5 Discrete Fourier Transforms **77**
5.1 Introduction to Fourier Analysis . . . 77
5.2 Fourier Series . . . 79
5.3 Discrete Fourier Transform (DFT) . . . 81
5.4 Discrete Fourier Transform (DFT) II . . . 86
5.5 Inverse Discrete Fourier Transform (IDFT) . . . 89
5.5.1 Lack of standardization . . . 90
5.6 1-D image compression . . . 90
5.7 2-D image compression . . . 91
5.8 Fast Fourier Transform . . . 93
5.9 FFT Algorithm (Butterfly) . . . 96
5.10 Aliasing . . . 99
5.11 Correlation Function . . . 100
5.12 A Two Dimensional FFT . . . 101
5.13 The Continuous Fourier Transform . . . 102
5.14 Exercises for Discrete Fourier Analysis . . . 103

6 Numerical Linear Algebra **105**
6.1 Solving via Matrix Factorization . . . 105
6.1.1 Gaussian Elimination ≡ Matrix Factorization . . . 108
6.1.2 Cost of Matrix Factorization . . . 110
6.1.3 Pivoting and Factorization . . . 111
6.2 Condition numbers and Norms . . . 113

6.2.1 Properties of Norms . . . 113
6.2.2 Matrix Norms . . . 113
6.2.3 Conditioning . . . 114
6.2.4 The residual . . . 115
6.3 Exercises for Numerical Linear Algebra . . . 116

7 Google Page Rank 119
7.1 Introduction . . . 119
7.2 Representing Random Surfer by a Markov Chain Matrix . . . 121
7.2.1 Dead End Pages . . . 121
7.2.2 Cycling Pages . . . 122
7.3 Markov Transition Matrices . . . 123
7.4 Page Rank . . . 124
7.5 Convergence Analysis . . . 126
7.5.1 Some Technical Results . . . 127
7.5.2 Convergence Proof . . . 128
7.6 Practicalities . . . 129
7.7 Summary . . . 130
7.7.1 Some Other Interesting Points . . . 130

8 Least Squares Problems 131
8.1 Finding Parameters . . . 131
8.2 Least Squares Fitting . . . 132
8.2.1 Total squared error . . . 134
8.2.2 The Normal Equations . . . 134
8.2.3 The QR decomposition . . . 137
8.3 Exercises for Least Squares Problems . . . 140

A Local Error of an ODE Method 141

B Complex Numbers and Roots of Unity 143
B.1 Review of Complex Numbers . . . 143
B.1.1 Roots of unity . . . 144
B.1.2 Orthogonality property . . . 146

C Computerized Tomography (CT) 147
C.1 Continuous Fourier Transforms . . . 149
C.2 The Slice Theorem . . . 150
C.3 Filtered Backprojection . . . 152
C.3.1 Filtering Operation . . . 153
C.3.2 Back Projection . . . 155
C.3.3 Example Reconstruction . . . 155

D Big Oh 159

Chapter 1

Floating Point Number Systems

1.1 Pitfalls in Floating Point Computation

The content of most mathematical courses assumes that all arithmetic yields *exact* results, say working over the real number line $\mathbb{R}$. However, real numbers can have potentially infinitely many digits (e.g., π or e), while computers possess only finite storage and processing speeds. Therefore, to solve many-real world problems that involve the real numbers, we instead typically use a computer's *floating point number system*, which is an alternative number system in which most real numbers have only an *approximate* representation. In this section, we give two examples of computational pitfalls that result from the use of such inexact numbers and associated arithmetic operations.

1.1.1 Example 1

Suppose we want to evaluate the integral

$$I_n = \int_0^1 \frac{x^n}{x+\alpha}\, dx \quad ; \quad \alpha \quad \text{constant parameter.} \tag{1.1.1}$$

Noting that, for $n = 0$

$$\begin{aligned} I_0 &= \int_0^1 \frac{1}{x+\alpha}\, dx \\ &= \log\left[\frac{1+\alpha}{\alpha}\right] \end{aligned}$$

and for $n > 0$

$$\begin{aligned} I_n + \alpha I_{n-1} &= \int_0^1 \frac{x^n + \alpha x^{n-1}}{x+\alpha}\, dx \\ &= \int_0^1 x^{n-1}\, dx \\ &= \frac{1}{n} \end{aligned}$$

we can derive a recurrence formula for I_n,

$$I_n = \frac{1}{n} - \alpha I_{n-1} \ . \tag{1.1.2}$$

Therefore, we could simply evaluate I_n for any $n \geq 0$ by noting that $I_0 = \log\left(\frac{1+\alpha}{\alpha}\right)$, and then use equation (1.1.2) to evaluate I_n starting from I_0.

Consider coding this idea in C using standard *floating point* arithmetic (in "single precision", which we will define a bit later.) Executing such a code, for $\alpha = .5$ we obtain

$$I_{100} = 6.64 \times 10^{-3} \tag{1.1.3}$$

while for $\alpha = 2.0$ we get

$$I_{100} = 2.1 \times 10^{22} \ . \tag{1.1.4}$$

Are both of these results correct? Let's derive a simple bound to check if they are reasonable. When $\alpha > 1$, then $(x + \alpha) > 1$ for $0 \leq x \leq 1$, therefore

$$\begin{aligned} \int_0^1 \frac{x^n}{x+\alpha}\, dx &\leq \int_0^1 x^n \, dx \\ &= \frac{1}{n+1} \ . \end{aligned}$$

Thus, equation (1.1.4) is clearly incorrect, since $1/(n+1) \leq 1$ for all $n \geq 0$. What happened? It turns out that due to the way that computer arithmetic errors can accumulate, equation (1.1.2) is fine for $\alpha = .5$, but produces garbage for $\alpha = 2.0$. That is, while the recursion we derived is exactly correct under the real numbers, the output of our algorithm computed using floating point numbers can sometimes be entirely incorrect!

1.1.2 Example 2

Suppose we would like to compute the value $e^{-5.5}$ on a simple calculator that does not have a built-in e^x function. One possible method is to use the Taylor series expansion

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ... \tag{1.1.5}$$

If our calculator is also limited to carrying only five significant figures, we get

$$e^{-5.5} = 1.0 - 5.5 + 15.125 - 27.730 + ... \tag{1.1.6}$$

and after 25 terms, additional terms no longer change the sum.

Another method to evaluate $e^{-5.5}$ also uses a Taylor series by noting that

$$e^{-x} = \frac{1}{e^x} = \frac{1}{1 + x + \frac{x^2}{2!} + ...} \tag{1.1.7}$$

so that

$$e^{-5.5} = \frac{1}{1 + 5.5 + 15.125 + ...}. \tag{1.1.8}$$

(Importantly, in this new expression $x = 5.5$ instead of $x = -5.5$.) Again after 25 terms, additional terms no longer change the sum.

Let us now evaluate the results. The correct answer, rounded to five significant digits, is

$$e^{-5.5} = .0040868. \tag{1.1.9}$$

However, the computational results obtained by the two methods outlined above are

$$\begin{aligned} e^{-5.5} &= 1 - 5.5 + 15.125 - ... = .0026363 \\ e^{-5.5} &= \frac{1}{1 + 5.5 + 15.125 + ...} = .0040865. \end{aligned}$$

In each case we employed a perfectly valid Taylor series expression for the value $e^{-5.5}$ and our calculator used the same number of significant figures. Why is the result obtained by the first method so much less accurate?

1.2 Introduction to Floating Point Numbers

1.2.1 Real Numbers and Normalized Digital Expansions

For many of our later discussions in this course, we will simply assume we are using the arithmetic of the real number system, denoted by $\mathbb{R}$. The real number system $\mathbb{R}$ is infinite in two senses

1. it is infinite in *extent*, in the sense that there are numbers x in $\mathbb{R}$ such that $|x|$ is arbitrarily large

2. it is infinite in *density*, in the sense that any interval $I = \{x \mid a \leq x \leq b\}$ of $\mathbb{R}$ is an infinite set.

Computer systems can only represent finite sets of numbers, so all the actual implementations of algorithms must use approximations to $\mathbb{R}$ and inexact arithmetic. There are two standard approximations to $\mathbb{R}$ used by computers, both of which are *floating point number systems*.

To describe computer floating point number systems, we need to look first at the representation of real numbers as *normalized digital expansions* relative to some *base number* for the digits. As humans, we (usually) use the digits of base 10, i.e. 0,1,2,...,8,9, called the decimal digits. For example, the representation of the rational number, 73/3, as a normalized decimal digit expansion is

$$0.24333333... \times 100 \ = 2 \times 10 + 4 \times 10^0 + 3 \times 10^{-1} + 3 \times 10^{-2}...$$

By "normalized" we mean that we have rewritten the number so that the first nonzero digit lies one place to the right of the decimal point; we have achieved this shifting of the

decimal point by multiplying by some power of the base number (in this case $10^2 = 100$), so the number being represented remains unchanged. (Other normalization conventions are also common, e.g., placing the first non-zero digit one place to the *left* of the decimal point.)

Computers, however, typically use 2 as the base number for the digits, which are 0,1; or they use 16, in which case the digits are 0,1,2, ... , 8,9,A,B, ...,E,F. Now

$$73/3 = 24 + 1/3 = 16 + 8 + 1/4 + (1/4)(1/3)$$

using rational number representations. So for base 2, 73/3 has normalized binary digit expansion

$$0.11000010101... \times 2^5 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}...$$

while for base 16, 73/3 has normalized hexadecimal digit expansion

$$0.18555... \times 16^2 = 1 \times 16^1 + 8 \times 16^0 + 5 \times 16^{-1} + 5 \times 16^{-2}...$$

Having considered a few examples, we can now summarize the general case. Let β be a positive integer that is to be used as the base for a number system. Thus

$\beta = 10$: the decimal number system;

$\beta = 2$: the binary number system;

$\beta = 16$: the hexadecimal number system.

Each positive number in $\mathbb{R}$ can be represented by an infinite base β expansion in the normalized form

$$0.d_1 d_2 d_3 \cdots \times \beta^p$$

where

- d_k are digits in base β, that is, 0, 1, ..., $\beta - 1$;
- 'normalized' means $d_1 \neq 0$;
- the exponent p is an integer (positive, negative or zero).

The sequence of digits $0.d_1 d_2 d_3...$ is called the *significand* or *mantissa*.

1.2.2 A Floating Point Number Format

We emphasize that the normalized form above still describes the real numbers, which we previously observed to be infinite in two respects. Floating point number systems therefore limit the infinite density of $\mathbb{R}$ by allowing only a finite number, t, of digits in the significand. They also limit the infinite extent of $\mathbb{R}$ by allowing only a finite set of integer values for the exponent p; that is, $L \leq p \leq U$ for specified integers $L < 0$ and $U > 0$. Thus, every floating point number system F that we will consider can be characterized by four integer parameters, $\{\beta, t, L, U\}$. The numbers in such a system are precisely those of the form

$$\pm\, 0.d_1 d_2 ... d_t \times \beta^p \quad \text{for } L \leq p \leq U \text{ and } d_1 \neq 0$$
$$\text{or} \quad 0 \quad \text{(a very special floating point number)}.$$

1.2.3 A Note on IEEE Floating Point Standards

In practice, there are two standardized floating point number systems for digital computers that are very widely used in the design of software and hardware:

IEEE single precision: $\{\beta = 2;\ t = 24;\ L = -126;\ U = 127\}$.

IEEE double precision: $\{\beta = 2;\ t = 53;\ L = -1022;\ U = 1023\}$.

Caveats: The general floating point format we have adopted differs in a few ways from that prescribed by the IEEE. For example, the exponent bounds listed above are misleading, because IEEE uses a normalization convention in which the first non-zero digit lies to the *left* of the decimal point, rather than the right. In implementation, the IEEE standards also reserve certain bit configurations for a few additional special numbers, including 0, $\pm\infty$, and `NaN` (for Not a Number). IEEE also slightly extends the range of available numbers near zero by allowing so-called denormalized or subnormal numbers. For simplicity, we will not consider these details in this course.

Although less common, IEEE also defines standards for higher precision numbers (e.g., quad or even oct precision) and lower precision numbers (half precision). The latter have gained popularity in recent years for training of neural networks, in cases where high performance is essential and lower precision is acceptable.

1.3 Absolute and Relative Error

To understand the impact of the approximations caused by computing with floating point arithmetic rather than real arithmetic, we must consider the error that doing so induces. When we obtain a computed result x of some calculation and we wish to discuss its relationship with the correct mathematical result x_{exact}, we can measure either the *absolute error*:

$$Err_{abs} = |x_{exact} - x|$$

or the *relative error*:

$$Err_{rel} = \frac{|x_{exact} - x|}{|x_{exact}|}\,. \tag{1.3.1}$$

Note: When measuring relative error, occasionally the value used in the denominator is the computed value $|x|$, rather than $|x_{exact}|$ as stated above. In most cases, the difference between the two definitions is insignificant. (We will also occasionally use the *signed error* and *signed relative error*, which are defined as above but without the absolute value signs.)

For the type of computations performed on a digital computer it is often the relative error measure which is most useful, because there is a close relationship between relative error and the number of significant digits in the computed result. Recall that the *significant digits* in a number are the digits starting with the first (i.e., leftmost) nonzero digit that are considered to be reliable. (e.g., if you have a weight scale that you know to be accurate within 1 gram, and when you weigh some object it says 17.73g, then the last few digits are unreliable and only the first two digits are *significant* digits.)

The computed result x is said to approximate x_{exact} to *about* s significant digits if the relative error is approximately 10^{-s}; or to be more precise, if the relative error satisfies

$$0.5 \times 10^{-s} \leq \frac{|x_{exact} - x|}{|x_{exact}|} < 5.0 \times 10^{-s}. \tag{1.3.2}$$

Consider the case of Example 2 in the first section. The correct answer for this problem (rounded to five significant digits) is

$$e^{-5.5} = .0040868.$$

The value computed by the first method is $x_1 = .0026363$ and its relative error is

$$Err_{rel} = \frac{|.0040868 - x_1|}{.0040868} \approx 3.5 \times 10^{-1}.$$

Based on equation (1.3.2), we would expect that x_1 has approximately one significant digit correct. In fact, there are zero digits correct in this case (rounded to one digit, we have 0.003 vs. 0.004). Similarly, consider the value computed in Example 2 by the second method, namely $x_2 = .0040865$. Its relative error is

$$Err_{rel} = \frac{|.0040868 - x_2|}{.0040868} \approx 0.7 \times 10^{-4}.$$

Based on equation (1.3.2), we can expect that x_2 has approximately four significant digits correct. In fact, x_2 agrees exactly in its first four (nonzero) digits, and then begins to deviate.

An important property of relative error is that it gives a measure of the number of correct significant digits independent of the actual magnitudes of the numbers involved.

1.4 Properties of Floating Point Numbers

1.4.1 Relating Real and Floating Point Numbers

Since a floating point number system can only approximate the real numbers in most cases, we must take great care to distinguish whether a give number is a floating point number or a real number. For a real number $x \in \mathbb{R}$ which has an exponent in the range allowed by F, we will express its corresponding floating point number approximation as $fl(x)$. In the general case, $fl(x) \neq x$ because most real numbers cannot be represented exactly. In order to compute $fl(x)$, we must modify x to become a valid representable floating point number, typically by eliminating some of its smaller digits.

One natural way to envision the floating point numbers is as a finite set of "slots" laid out along the infinite real number line. The placement and spacing of these slots is determine by the particular parameters of F. Converting a real number x into its floating point counterpart $fl(x)$ then corresponds to choosing an appropriate nearby slot, and in doing so we incur some amount of error. This type of error is called *round-off error*. (Note also that for any particular floating point number y, there is an infinite set of real numbers x satisfying $fl(x) = y$, i.e., corresponding to the same slot.)

A fact which may be initially surprising is that the spacing of the floating point numbers along the real line is *not* uniform; that is, floating point numbers closer to zero in magnitude (i.e., more negative exponents) are spaced closer together, and numbers having larger magnitudes (i.e., more positive exponents) are spaced further apart. This fact arises from a tradeoff in the design of standard floating point number systems: allowing the exponent value p to vary lets us represent much larger and smaller numbers than we could otherwise, at the cost of this slightly unintuitive behavior. (By contrast, so-called *fixed point* number systems are an alternative in which the exponent, or scale factor, is fixed in advance and cannot change. In such systems, the spacing and number of stored decimal places is therefore also fixed. We can see now the meaning of the name *floating point*: we are allowing the decimal or radix point to "float around", by changing the exponent value, in order to access numbers that have much larger or smaller magnitudes than would be possible with a fixed radix point.)

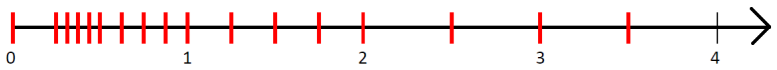


Figure 1.1: A visualization of the complete set of non-negative numbers in the floating point number system F where $\beta = 2, t = 3, L = -1$, and $U = 2$. Each red segment is a slot corresponding to a valid floating point number. They are spaced non-uniformly along the real number line.

Another key point is that the exponent bounds L and U limit the magnitude of the numbers we can represent. If a real number x too large in magnitude (i.e., its corresponding exponent in normalized form would be greater than U), it would not make sense to simply return the closest valid number in F for $fl(x)$, since the resulting $fl(x)$ may be arbitrarily different from x. We instead say that *overflow* has occurred, which is a type of error or *exception*. Typical floating point systems will yield a result of $\pm\infty$ instead. Similarly, if a real number x is too small in magnitude to be represented (i.e., its corresponding exponent in normalized form would be less than L), we say that *underflow* has occurred. Underflow is also considered to be an exception, albeit sometimes less destructive in practice. The value returned when underflow occurs is simply zero. Going forward, our analyses of the behavior of floating point numbers will assume that neither overflow nor underflow has occurred, unless indicated otherwise.

1.4.2 Mathematical Relationship Between $x \in \mathbb{R}$ and $fl(x) \in \mathbf{F}$

An important consequence of the design of floating point number systems is that the largest *relative* error that can occur in representing a real number x by its floating point approximation $fl(x)$ is bounded, for all x whose exponents are in the valid range $[L, U]$. This maximum relative error measure of machine precision is called *machine epsilon*, denoted here by E. It is also called the *unit round-off error* since E can be defined as the smallest number such that, using $\mathrm{F}(\beta, t, L, U)$,

$$fl(1 + E) > 1\,.$$

That is, what is the smallest number we can add to 1 before the result returned is actually a different floating point number? If we try to pick a value for E that is too small, we will just get back 1 again because we don't have enough digits of precision (t is too small) to store the result as a valid and different floating point number.

The value of machine epsilon will depend on exactly how we determine $fl(x) \in \mathrm{F}$ from the real number x, i.e., which slot we assign the real number to. The different available strategies are called rounding modes or rounding schemes. Perhaps the most obvious choice is to take the closest slot to x; this is consistent with our usual notion of rounding (or round-to-nearest) since we are effectively rounding away any smaller digits that we do not have room to store. Another common option is called truncation (or round-towards-zero or "chopping"), in which any digits after t are simply discarded. In our number line view, this scheme corresponds to taking the next slot (floating point number) that is *smaller* in magnitude (i.e., closer to zero).

If a computer uses base β arithmetic with t digits in the significand, i.e., $\mathrm{F}(\beta, t, L, U)$, and uses round-to-nearest, then the value of *machine epsilon* (or unit round-off error) is

$$E = \frac{1}{2}\,\beta^{1-t}\ . \tag{1.4.1}$$

If truncation is used instead, then E will be

$$E = \beta^{1-t}\ . \tag{1.4.2}$$

For practical purposes, we are usually only interested in the order of magnitude of E (e.g., is it around 10^{-3} or 10^{-8}?), as this gives a sufficient indication of the accuracy of our floating point system.

In real-world floating point systems, there are additional choices available for rounding modes, including round-up, round-down, and round-towards-infinity, as well as options for tie-breaking under round-to-nearest (when the real value is halfway between two floating point numbers), such as round-to-odd, round-to-even, etc. For simplicity, this course will only consider truncation or round-to-nearest (with tie-breaking by rounding up, if it arises).

From the bounds described above and the definition of relative error, we can observe that the signed error between any nonzero real number x and its floating point representation $fl(x)$ can be written as a small multiple of x, as follows. If $x \neq 0$, the signed relative error of using $fl(x)$ for x is $\delta = (fl(x) - x)/x$ which, by the design of F, does not exceed E in size. That is,

$$fl(x) - x \quad = \quad \delta x \quad \text{where } |\delta| \leq E.$$

Thus we have the relationship

$$fl(x) = x\,(1 + \delta) \tag{1.4.3}$$

where δ is *some* value (positive, negative or zero), which must satisfy $-E \leq \delta \leq E$. In our analyses of round-off error, it is critical to remember that E is defined as a **positive** number (which gives a bound on the magnitude of δ), whereas δ is a **signed** number (which gives the error for a specific conversion of x to $fl(x)$) – they must not be used interchangeably.

1.4.3 Arithmetic Operations Under Floating Point

Having discussed floating point number systems and what numbers they can faithfully represent, we must now consider how to perform arithmetic operations on such numbers and what assurances we have on the accuracy of the results. The IEEE floating point standard essentially says that a single arithmetic operation in F must be done so that, if the exact real arithmetic result would have been x, the computed result is the corresponding representable floating point number $fl(x)$. In real-world implementations of floating point numbers, this fundamental guarantee is achieved by temporarily using one or more extra digits during the calculation of that single arithmetic operation, and rounding the result back into F. As before, an *exception* occurs if the resulting exponent is out of range, which leads to overflow if the exponent is too large, or underflow if the exponent is negatively too large.

Let us denote the floating point addition operator for F by $\oplus$, so as to distinguish it from the real addition operator, $+$. (We will likewise use $\ominus, \oslash, \otimes$ for floating point subtraction, division, and multiplication). Translating the guarantee above into mathematics using (1.4.3), for adding two floating point numbers $w, z \in F$, we have

$$w \oplus z = fl(w + z) = (w + z)(1 + \delta) \tag{1.4.4}$$

where we still have the bound $|\delta| \leq E$. Other basic floating point arithmetic operations provide the analogous guarantee, i.e., that the result will be *as if* we performed the operation with real arithmetic and converted the result back into a floating point number. However, this guarantee holds only for each *single operation* in isolation, as we will discuss in the next section.

Exercise

Make up an example to show that the sum of three numbers in F, computed using $\oplus$, depends on the order in which the terms are added. That is, find values of a, b, $c \in F$ such that

$$(a \oplus b) \oplus c \neq a \oplus (b \oplus c)\,.$$

1.5 Round-off Error Analysis and Cancellation

Although floating point number systems provide the guarantee that each individual floating point arithmetic operation is done with a relative error bounded by E, it is not the case that the result of a sequence of two or more floating point arithmetic operations has a relative error that is bounded by E. Consider adding three floating point numbers a, b, $c \in$ F using the floating point addition operation in F. How does $a + b + c$ differ from $(a \oplus b) \oplus c$? That is, what is the size of the relative error in this sum, computed using F? Answering this question amounts to performing a *round-off error analysis* to determine a bound.

The rule of equation (1.4.4), says we can express each floating point addition as a real addition, followed by a conversion back to floating point using $fl(\cdot)$. Each conversion

will incur some relative error that will depend on how far the resulting real is from its floating point counterpart; since we have two additions, we will need two relative errors, δ_1 from the first $\oplus$ and δ_2 from the second $\oplus$. Straightforwardly applying (1.4.4) gives

$$\begin{aligned}(a \oplus b) \oplus c &= fl(a+b) \oplus c = fl(fl(a+b)+c) \\ &= fl((a+b)(1+\delta_1)+c) = ((a+b)(1+\delta_1)+c)(1+\delta_2) \\ &= (a+b+c+(a+b)\delta_1))(1+\delta_2). \end{aligned} \tag{1.5.1}$$

Thus we arrive at a rather nontrivial expression for the signed error of the floating point result compared to the real result:

$$(a+b+c) - (a \oplus b) \oplus c = -(a+b)\delta_1(1+\delta_2) - (a+b+c)\delta_2. \tag{1.5.2}$$

Now, if $a+b+c \neq 0$, then the relative error in the floating point sum is well defined. Let us use err_{rel} for the signed version of the relative error defined in (1.3.1), that is, $Err_{rel} = |err_{rel}|$. In this case,

$$err_{rel} = \frac{a+b+c-(a \oplus b) \oplus c}{(a+b+c)}.$$

From (1.5.2), we see that

$$\begin{aligned} err_{rel} &= \frac{-(a+b)\delta_1(1+\delta_2) - (a+b+c)\delta_2}{a+b+c} \\ &= -\left(\frac{a+b}{a+b+c}\right)\delta_1(1+\delta_2) - \delta_2. \end{aligned} \tag{1.5.3}$$

Look at what (1.5.3) tells us:

- the error due to the second $\oplus$ contributes $-\delta_2$ to err_{rel}. Fortunately, this contribution cannot be bigger than E in magnitude.
- the error due to the first $\oplus$ contributes $-((a+b)/(a+b+c))\delta_1(1+\delta_2)$. Now if the factor $(a+b)/(a+b+c)$ is large, then this term can be *much* bigger than δ_1 (or E). In fact, the only thing we can say for sure about this term is that it is not bigger than $(|a+b|/|a+b+c|)E(1+E)$.

When will the factor $|a+b|/|a+b+c|$ be large? Answer: the denominator must be smaller that the numerator. In other words, adding c to $a+b$ must exhibit some "cancellation". E.g., $a = 10{,}000$, $b = 0.1$, and $c = -10{,}000$. Then $a+b = 10{,}000.1$ and $a+b+c = 0.1$, yielding a factor of 100,001.

This example clearly confirms that the simple bound of E for the relative error of a single floating point arithmetic operation certainly does not hold for a sequence of such operations, and, furthermore, shows that the relative error can depend heavily on the input numbers involved.

We have been looking in detail at a tiny computation, two additions. Of course, computations of significance involve 10^m arithmetic operations for $m = 2, 3, 4 \cdots$. Getting an understanding of the impact of floating point arithmetic on the accuracy of the

results is a major undertaking. Theoretical estimates for, or bounds on, the error size is called round-off error analysis. Such analysis has been carried out for some common standard computations. A useful result of such analysis is a *condition number*, which describes the worst case possibility of error magnification during the calculation as a function of the input data. We will now continue our tiny round off error analysis and produce a condition number for the addition of three numbers.

Looking at (1.5.2), introducing absolute values on both sides, and applying absolute value rules (triangle inequality, etc.), we can show the following

$$|(a+b+c)-((a\oplus b)\oplus c)| \le |-(a+b)\delta_1-(a+b+c)\delta_2-(a+b)\delta_1\delta_2| \tag{1.5.4}$$
$$\le |a+b||\delta_1|+|a+b+c||\delta_2|+|a+b||\delta_1||\delta_2| \tag{1.5.5}$$
$$\le (|a|+|b|)|\delta_1|+(|a|+|b|+|c|)|\delta_2|+(|a|+|b|)|\delta_1||\delta_2| \tag{1.5.6}$$
$$\le (|a|+|b|+|c|)\,(|\delta_1|+|\delta_2|+|\delta_1|\,|\delta_2|). \tag{1.5.7}$$

In the last step we have slightly increased the upper bound (right hand side) by adding $|c||\delta_1|+|c||\delta_1||\delta_2|$ to make the bound symmetric in terms of the variables a, b and c. In other words, the new upper bound is no longer affected by the order of the terms in the sum; once can show it applies equally to $(a\oplus b)\oplus c$ or even $a\oplus(b\oplus c)$ or $b\oplus(c\oplus a)$. (This is convenient, though it does weaken the bound somewhat.)

If $a+b+c\neq 0$ then the size of the relative error in the floating point sum,

$$Err_{rel}=\frac{|(a+b+c)-((a\oplus b)\oplus c)|}{|a+b+c|},$$

is bounded as follows, from (1.5.7):

$$Err_{rel}\le\frac{|a|+|b|+|c|}{|a+b+c|}\,(|\delta_1|+|\delta_2|+|\delta_1||\delta_2|)\le\frac{|a|+|b|+|c|}{|a+b+c|}\,(2\,E+E^2), \tag{1.5.8}$$

recalling again that $|\delta_i|\le E$. Consider what (1.5.8) tells us:

- if $|a+b+c|\approx|a|+|b|+|c|$ (for example, if a, b, c are all positive, or all negative) then Err_{rel} is bounded by $(2\,E+E^2)$ which is small;
- if $|a+b+c|\ll|a|+|b|+|c|$ then Err_{rel} may be quite large, namely up to $(2\,E+E^2)$ multiplied by the "magnification factor" $\frac{|a|+|b|+|c|}{|a+b+c|}$.

The quantity $(|a|+|b|+|c|)/(|a+b+c|)$ is the condition number for the computation of adding a, b, and c, in any order.

In what situations will the factor $\frac{|a|+|b|+|c|}{|a+b+c|}$ be very large? This will happen when the denominator (the actual sum) is much smaller than the numerator (the sum of the absolute values). In other words, it can occur when there is a sum of both positive and negative values such that a phenomenon known as *cancellation* occurs.

For example, in the floating point number system $\mathrm{F}(10,5,-10,10)$ suppose that

$$a=10000.\,,\ b=3.1416\,,\ c=-10000.$$

Then $|a|+|b|+|c| = 20003.1416$ and $a+b+c = 3.1416$ so the relative error bound given by (1.5.3) is

$$Err_{rel} \leq 6367.2\,(2\,E + E^2) \approx 0.6$$

since $E = \frac{1}{2}\,10^{-4}$. (Note, in particular, that since the unit round-off error E is a small quantity we have $2\,E + E^2 \approx 2\,E$.) This relative error of 0.6×10^0 is very large, implying that there may be no significant digits correct in the result. Indeed, for this example the computation proceeds as follows:

$$(a \oplus b) \oplus c = 10003. \oplus (-10000.) = 3.0000$$

compared with the true sum which is 3.1416 and therefore the computed sum actually has one significant digit correct. Adding these three numbers in the order $(a \oplus c) \oplus b$, is the same as $(a \oplus b) \oplus c$ with

$$a = 10000.\,,\ b = -10000.\,,\ c = 3.1416$$

In this case, we get:

i) all five digits of the computed result correct

ii) the same error bound from (1.5.3).

So the condition number-based error bound of (1.5.3) is a bound for the worst case of a, b, c.

By contrast, using the same floating point number system $\mathrm{F}(10, 5, -10, 10)$, suppose that all three summands are positive:

$$a = 10000.\,,\ b = 3.1416\,,\ c = 10000.$$

In this case the relative error bound given by (1.5.3) is

$$Err_{rel} \leq 2\,E + E^2 \approx 2\,E \approx 10^{-4}$$

which implies that we can expect about four correct significant digits in the result. (This is a "best case" situation, where the relative error bound is a small multiple of E.) The actual computation for this case is as follows:

$$(a \oplus b) \oplus c = 10003. \oplus 10000. = 20003.$$

compared with the true sum which is 20003.1416 and we can see that we have, in fact, all five significant digits correct in this case.

1.6 Conditioning and Stability

It is important to understand the concept that some problems, as posed, may be *well-conditioned* and some may be *ill-conditioned*. More precisely, we like to have a measure

of how well-conditioned (or how ill-conditioned) a given problem may be. The concept of conditioning may be defined as follows.

Consider a Problem P with input values I and output values O. If a relative change of size ΔI in one or more input values causes a relative change in the mathematically correct output values which is guaranteed to be small (i.e., not too much larger than ΔI) then Problem P is said to be *well-conditioned*. Otherwise, Problem P is said to be *ill-conditioned*.

Remark 1: The above definition is *independent* of any particular choice of algorithm and independent of any particular computer number system. It is a statement about the mathematical problem.

Remark 2: There is always a "sliding scale" for "how well-conditioned" or "how ill-conditioned" a particular problem is. That is, we can compare the *relative* conditioning of different problems, but there is no absolute threshold for ill-conditioned vs. well conditioned.

It is important to distinguish conditioning, which is a property of a problem itself, from the somewhat related concept of stability, which is instead a property of an algorithm or a numerical process used to (approximately) solve a problem. We would again like small errors in the input or floating point calculations of an algorithm to lead only to correspondingly small changes in the output of that algorithm. For example, if a small error is steadily magnified by the algorithm, we consider the algorithm to be numerically unstable, since the error may grow to destroy the accuracy of our answer. If instead small errors shrink away towards zero during the algorithm's execution, then the algorithm is considered stable. We will consider an example to illustrate this idea.

1.7 A Stability Analysis

Consider the computation discussed in our earlier example,

$$I_n = \int_0^1 \frac{x^n}{x+\alpha}\, dx\,.$$

We state without proof that this is a well-conditioned problem (which could be solved by standard algorithms for numerical integration, sometimes called quadrature algorithms). We had developed an algorithm based on a recurrence formula, and found that for some values of the parameter α, very inaccurate results were obtained from this algorithm.

We can perform a stability analysis for the recurrence equation (1.1.2) as follows. Let's suppose that the floating point representation of I_0 introduces some initial error ϵ_0. Let's see how this error propagates using the recurrence relation (1.1.2). For simplicity, assume that no other errors are introduced at any stage of the computation after I_0 is computed. Let $(I_n)_A$ be the approximate value of I_n, i.e., the computed value of I_n including the effects of ϵ_0. Let $(I_n)_E$ be the exact value of I_n. Let $\epsilon_n = (I_n)_A - (I_n)_E$, i.e., the error at step n due to the initial error ϵ_0. The exact I_n satisfies

$$(I_n)_E \;=\; \frac{1}{n} - \alpha\,(I_{n-1})_E$$

while the approximate I_n satisfies

$$(I_n)_A = \frac{1}{n} - \alpha\ (I_{n-1})_A\ .$$

Subtracting these two equations gives the error at step n as

$$(I_n)_A - (I_n)_E = \left(\frac{1}{n} - \frac{1}{n}\right) - \alpha\ ((I_{n-1})_A - (I_{n-1})_E)$$

or

$$\begin{aligned} \epsilon_n &= (-\alpha)\ \epsilon_{n-1} \\ &= (-\alpha)(-\alpha)\epsilon_{n-2} \\ &... \\ &= (-\alpha)^n\ \epsilon_0\ . \end{aligned}$$

Thus if $|\alpha| > 1$ then any initial error ϵ_0 is magnified by an unbounded amount as $n \to \infty$. On the other hand, if $|\alpha| < 1$ then any initial error is "damped out", i.e., approaches zero as $n \to \infty$. We conclude that the algorithm is stable if $|\alpha| < 1$ and the algorithm is unstable if $|\alpha| > 1$.

Can you figure out a way to use equation (1.1.2) so that it is stable when $|\alpha| > 1$? Hint: try going backwards.

1.8 Exercises for Floating Point Numbers

1. The numbers in a floating point system are defined by a base β, a mantissa length t, and an exponent range $[L,\ U]$. A nonzero floating point number x has the form

$$x = \pm .b_1 b_2 \cdots b_t \times \beta^e$$

Here $.b_1 b_2 \cdots b_t$ is the *mantissa* and e is the exponent. The exponent satisfies $L \le e \le U$. The b_i are base-β digits and satisfy $0 \le b_i \le \beta - 1$. Nonzero floating point numbers x are normalized : $b_1 \neq 0$. Zero is represented by both zero mantissa and zero exponent.

   (a) What is the largest value of n so that $n!$ can be exactly represented in a floating point system where $(\beta, t, L, U) = (2, 5, -10, 10)$. To obtain full marks, you must show your work.

   (b) On a base-2 machine, the distance between 7 and the next largest floating point number is 2^{-12}. What is the distance between 70 and the next largest floating point number?

   (c) Assume that x and y are normalized positive floating numbers in a base-2 computer with t-bit mantissa. How small can $y - x$ be if $x < 8 < y$?

2. Consider a fictitious floating number system composed of the following numbers:

$$S = \{ \pm b_1.b_2b_3 \times 2^{\pm y} : \ b_2, b_3, y = 0 \text{ or } 1, \\ \text{and } b_1 = 1 \ \text{ unless } b_1 = b_2 = b_3 = y = 0 \} .$$

i. e. each number is normalized unless it is a zero.

(a) Plot the elements of S on the real axis.

(b) Show how many elements are contained in S. What are the values of OFL, UFL, and the machine epsilon?

3. Using the floating-point number format $(\beta, t, U, L) = (2, 20, -200, 200)$, store the distance between Earth and Sun ($1.5 * 10^8$ kilometers) and the distance between Toronto and Waterloo (75 kilometers). What length does the last bit of the mantissa represent in each case?

Chapter 2

Interpolation

It is often the case that one has a discrete (finite) set of data $(x_1, y_1), \ldots, (x_n, y_n)$ which describes the behaviour of some (unknown) function $y = p(x)$ and one wishes to determine $p(x)$, or at least some approximation to $p(x)$. One can then use this function to approximately evaluate the data at some unknown values, determine instantaneous changes at some points (i.e., derivatives), and so on.

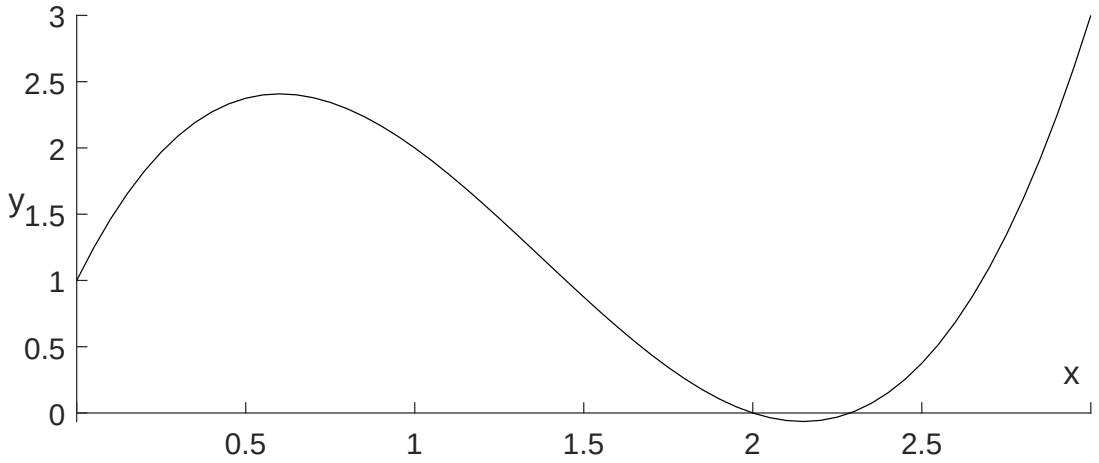
At the very least one desires that our approximate function $p(x)$ has the property that it *interpolates* (i.e., goes exactly through) the data points, that is, that

$$p(x_1) = y_1, \ldots, p(x_n) = y_n. \tag{2.0.1}$$

For example, suppose that we have the following four (x, y) points in the plane: $(0, 1)$, $(1, 2)$, $(2, 0)$ and $(3, 3)$. Then the polynomial

$$p(x) = \frac{4}{3}\,x^3 - \frac{11}{2}\,x^2 + \frac{31}{6}\,x + 1$$

interpolates the 4 points. Graphically, we see the following:



Hence, the function can also give approximations of y for x values besides the original points. For example when $x = \frac{3}{2}$ then we have that $y = p(x) = \frac{7}{8}$.

In general, an interpolation function $p(x)$ for a given data set is not unique. In fact, there are infinitely many functions that will pass through any given data set. We will therefore restrict ourselves to a few specific types of interpolation strategies.

There are two primary topics covered in this chapter. The first topic is polynomial interpolation where $p(x)$ is a polynomial. This topic is basic to numerical computation in at least two ways:

- for applications involving small numbers of points, that is, typically n not bigger than 5 or 6;
- as a component (conceptually, or explicitly) of a larger computation. For example polynomial interpolation is used in numerical integration, solving differential equations, optimization, and other mathematical computations.

The second topic of this chapter is piecewise polynomial interpolation. In this case $p(x)$ is defined as different polynomials for different intervals of x. We discuss two particularly common cases:

- piecewise linear interpolation, in which $p(x)$ is linear on each subinterval;
- cubic splines, in which $p(x)$ is cubic on each subinterval.

Both of these interpolating functions play an important role in the representation of curves for computer graphics, which is the application that we emphasize in this topic.

2.1 Polynomial Interpolation

The easiest type of function to use for interpolation is to chose $p(x)$ to be a polynomial. Practically speaking, it is simple and easy to compute. Theoretically speaking, it has a nice existence and uniqueness property.

Theorem: Given n data pairs (x_i, y_i), $i = 1, \ldots, n$ with distinct x_i, there is a unique polynomial $p(x)$ of degree not exceeding $n - 1$ that interpolates this data.

There are several ways to prove this theorem (known as the *unisolvence theorem*). One is based on the Vandermonde system of equations, discussed in the next subsection. Another is based on directly constructing $p(x)$ using the Lagrange polynomial form. In fact both approaches can also be used to construct the interpolation polynomial.

2.1.1 Linear Equations: Vandermonde Systems

Polynomials of degree $n - 1$ or less are commonly represented in the form

$$p(x) = c_1 + c_2 x + \cdots + c_n x^{n-1}, \tag{2.1.1}$$

which uses n coefficients $c_1, ..., c_n$. The easiest method of finding a polynomial interpolant is to set up and solve a linear system of equations. This can be done by setting up a

system of n linear equations $p(x_i) = y_i$ and solving for the unknowns. For example, for the 4 planar points given previously we would have that $p(x) = c_1 + c_2x + c_3x^2 + c_4x^3$ with an associated linear system given by

$$\begin{array}{rcl} p(0) &=& 1 \\ p(1) &=& 2 \\ p(2) &=& 0 \\ p(3) &=& 3 \end{array} \quad \Longrightarrow \quad \begin{array}{rcl} c_1 &=& 1 \\ c_1 + c_2 + c_3 + c_4 &=& 2 \\ c_1 + 2c_2 + 4c_3 + 8c_4 &=& 0 \\ c_1 + 3c_2 + 9c_3 + 27c_4 &=& 3 \end{array}.$$

In general, if we had a set of data $(x_1, y_1), ..., (x_n, y_n)$, and want a polynomial of the form (2.1.1) then we can set up the linear system $V \cdot \vec{c} = \vec{y}$ where

$$V = \begin{bmatrix} 1 & x_1 & ... & x_1^{n-1} \\ 1 & x_2 & ... & x_2^{n-1} \\ & & ... & \\ 1 & x_n & ... & x_n^{n-1} \end{bmatrix}, \quad \vec{c} = \begin{bmatrix} c_1 \\ \\ \\ c_n \end{bmatrix} \quad \text{and} \quad \vec{y} = \begin{bmatrix} y_1 \\ \\ \\ y_n \end{bmatrix}.$$

Matrices of the form V are called *Vandermonde* matrices. All of the data required for creating one is contained in its second column, i.e., $V_{i,2} = x_i$.

These facts have both practical and theoretical implications. The theoretical implication is that we can prove the basic theorem by showing that V is non-singular. Indeed, the usual proof of this theorem is based on establishing that

$$det(V) = \prod_{i<j}(x_i - x_j).$$

The practical implication is that we have reduced computing the interpolating polynomial to solving a linear system of equations.

2.1.2 Lagrange form

Expressing polynomials in monomial form (that is, in terms of powers of x^i as done above) is often a natural choice that is straightforward to compute with. There still remain, however, alternative forms that are also very useful. In particular, there are other forms that can be more efficient for common uses of polynomial interpolation. The Lagrange form is an example of one such alternate form.

Given a set of data (x_i, y_i), $i = 1, \ldots, n$, consider a set of n *Lagrange basis functions*, $L_k(x)$, $k = 1, \ldots, n$, defined as

$$L_k(x) = \frac{(x - x_1) \cdots (x - x_{k-1})(x - x_{k+1}) \cdots (x - x_n)}{(x_k - x_1) \cdots (x_k - x_{k-1})(x_k - x_{k+1}) \cdots (x_k - x_n)}.$$

These polynomials have the property that $L_i(x_j) = 1$ if $i = j$ and $L_i(x_j) = 0$ for all other j. If we set

$$p(x) = y_1L_1(x) + \cdots + y_iL_i(x) + \cdots + y_nL_n(x)$$

then $p(x)$ is a polynomial of degree $n-1$ which interpolates the n points since for each i we have

$$\begin{aligned} p(x_i) &= y_1 L_1(x_i) + \cdots + y_i L_i(x_i) + \cdots + y_n L_n(x_i) \\ &= y_1 \cdot 0 + \cdots + y_i \cdot 1 + \cdots + y_n \cdot 0 \\ &= y_i. \end{aligned}$$

For the cubic example from the previous page, we have that

$$\begin{aligned} L_1(x) &= \tfrac{(x-1)(x-2)(x-3)}{(-1)(-2)(-3)} = \tfrac{(x-1)(x-2)(x-3)}{-6} \\ L_2(x) &= \tfrac{(x-0)(x-2)(x-3)}{(1)(-1)(-2)} = \tfrac{(x)(x-2)(x-3)}{2} \\ L_3(x) &= \tfrac{(x-0)(x-1)(x-3)}{(2)(1)(-1)} = \tfrac{(x)(x-1)(x-3)}{-2} \\ L_4(x) &= \tfrac{(x-0)(x-1)(x-2)}{(3)(2)(1)} = \tfrac{(x)(x-1)(x-2)}{6} \end{aligned}$$

with

$$p(x) = L_1(x) + 2L_2(x) + 0 \cdot L_3(x) + 3L_4(x).$$

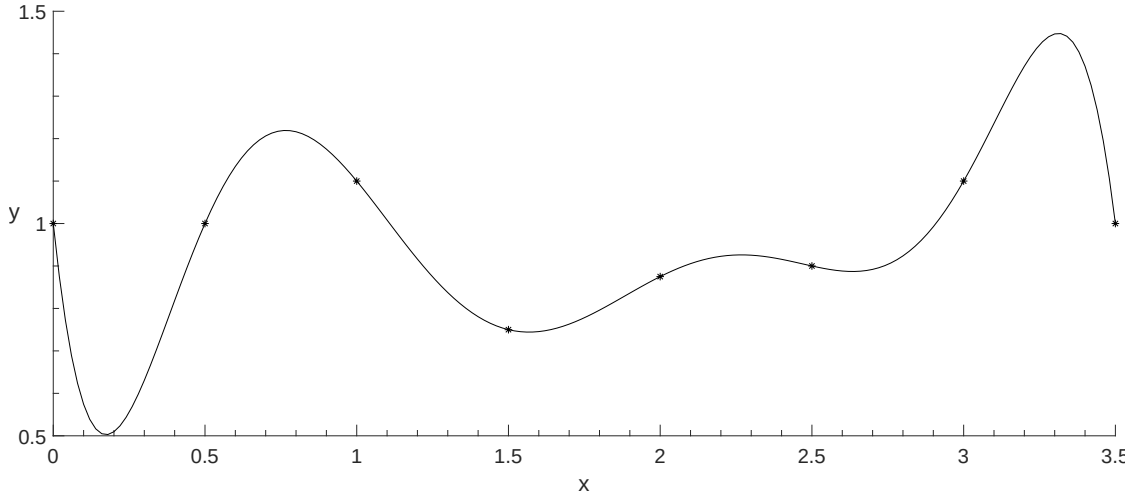
One advantage of the Lagrange form is that the interpolating polynomial can be written down directly, without needing to solve a (Vandermonde) linear system.

2.2 Piecewise Polynomial Interpolation

Unfortunately a polynomial becomes a poor approximant to a set of points as the number of points increases. Indeed for large sets of points such high degree interpolating polynomials tend to oscillate wildly in order to go through all the points. For example, if we look for a degree 7 polynomial which interpolates the 8 points $(0,1)$, $(\frac{1}{2},1)$, $(1,\frac{11}{10})$, $(\frac{3}{2},\frac{3}{4})$, $(2,\frac{7}{8})$, $(\frac{5}{2},\frac{9}{10})$, $(3,\frac{11}{10})$ and $(\frac{7}{2},1)$ we get

$$p(x) = -\frac{53}{225}\,x^7 + \frac{221}{75}\,x^6 - \frac{3296}{225}\,x^5 + \frac{733}{20}\,x^4 - \frac{172247}{3600}\,x^3 + \frac{2254}{75}\,x^2 - \frac{8183}{1200}\,x + 1$$

which looks like



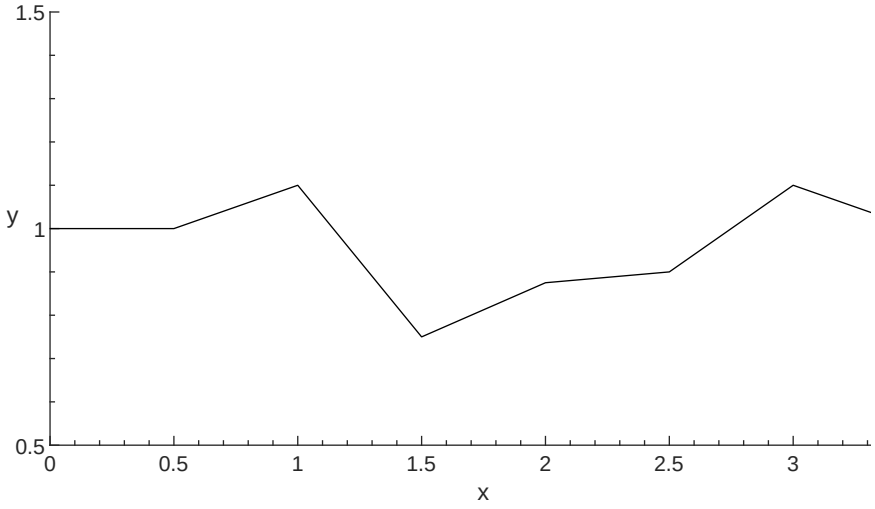
This is not as "nice" of a fit to the points as we might have desired or expected, e.g., the function significantly exceeds the maximum and minimum values of the data points on the interval.

Assume now that we have n points $(x_1, y_1), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$. Then a *piecewise interpolating polynomial* for these n points is a function $p(x)$ satisfying

- a function of x for $x_1 \leq x \leq x_n$,
- an interpolant, i.e., $p(x_i) = y_i$,
- a polynomial for each subinterval, $x_i \leq x_{i+1}$, usually different on each subinterval,
- continuous for the total interval $x_1 < x < x_n$.

The simplest example of a piecewise interpolating polynomial is the function which joins each point by a straight line. In this case the function is a polynomial of degree 1 in each interval.

For example, a piecewise interpolating polynomial with degree 1 components for the example at the start of the section looks like



This piecewise interpolating polynomial is given by

$$p(x) = \begin{cases} 1 & 0 \leq x \leq \frac{1}{2} \\ \frac{x}{5} + \frac{9}{10} & \frac{1}{2} \leq x \leq 1 \\ \frac{-7x}{10} + \frac{9}{5} & 1 \leq x \leq \frac{3}{2} \\ \ldots & \ldots \\ \ldots & \ldots \\ \frac{-x}{5} + \frac{17}{10} & 3 \leq x \leq \frac{7}{2} \end{cases}$$

As a second example, the Matlab or Python command `plot(x,y)`, for vector arguments `x` and `y`, initiates a plot of the piecewise linear interpolant of (x_i, y_i), $i = 1, \ldots, n$.

Of course, the example of piecewise interpolating polynomials given above also has limitations. In particular, the derivatives of the functions are discontinuous at the inter-

polation points, leading to sharp kinks in the graph. In the next section we will look at piecewise polynomials called *splines* which are smooth at all points. In order to make the computations easier we first discuss in the next section a special kind of interpolation problem called Hermite interpolation where we specify both interpolating points and also interpolating derivatives.

2.3 Piecewise Hermite Interpolation

There are also alternate forms of interpolation problems. In particular, there are interpolation problems where for a given function $p(x)$, we wish to interpolate both the function values and its derivatives. In this section we give a simple example of such an interpolant for degree 3. The results from this section are useful in the later sections on splines in order to simplify those sections.

Suppose that the domain of interest is divided into $N-1$ subintervals $[x_i, x_{i+1}],\ i = 1, ..., n-1$. We can write a cubic polynomial in the i-th interval as

$$\begin{aligned} p_i(x) &= a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \\ &\quad \text{where } x \in [x_i, x_{i+1}] \ . \end{aligned} \tag{2.3.1}$$

Thus let $p_i(x),\ i = 1, ..., n-1$, denote a cubic polynomial on the i^{th} subinterval given in the form:

$$p_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3. \tag{2.3.2}$$

Suppose we want to specify both the function values and the derivatives at $x = x_i$:

$$p(x_i) = y_i, \quad p'(x_i) = s_i, \quad i = 1, ..., n \ . \tag{2.3.3}$$

We then require the following conditions on each subinterval, i.e. for $i = 1, ..., n-1$:

$$\begin{aligned} p_i(x_i) &= y_i \\ p_i(x_{i+1}) &= y_{i+1} \\ p_i'(x_i) &= s_i \\ p_i'(x_{i+1}) &= s_{i+1} \ . \end{aligned} \tag{2.3.4}$$

The set of equations (2.3.4) gives four conditions on each particular subinterval, which uniquely determines the four unknowns a_i, b_i, c_i, d_i. The solution to the set of equations (2.3.4) is, for $i = 1, ..., n-1$:

$$\begin{aligned} a_i &= y_i \\ b_i &= s_i \\ c_i &= \frac{3y_i' - 2s_i - s_{i+1}}{\Delta x_i} \\ d_i &= \frac{s_{i+1} + s_i - 2y_i'}{(\Delta x_i)^2} \end{aligned} \tag{2.3.5}$$

where $y_i' = \frac{y_{i+1} - y_i}{\Delta x_i}$ and $\Delta x_i = x_{i+1} - x_i$.

2.4 Spline Interpolants

It follows from the set of equations (2.3.4) that at each of the $n-2$ *interior* points we have the following relationships, namely, for $i = 1, ..., n-2$:

$$\begin{aligned} p_i(x_{i+1}) &= p_{i+1}(x_{i+1}) \\ p_i'(x_{i+1}) &= p_{i+1}'(x_{i+1}). \end{aligned} \tag{2.4.1}$$

In other words, the interpolant and its first derivatives are continuous at the knots (breakpoints) x_i. Technically, we use the terminology *knots* to refer to the points where the interpolant is permitted to change its definition from one polynomial to another. We use the terminology *nodes* to refer to the points where data is specified. In this particular case, the nodes and the knots are the same points.

However, a more common problem is one where we know only the function values at the knots and we would like to produce a smooth interpolant. So the question becomes, can we choose the s_i values such that the interpolant, and the first and second derivatives of the interpolant, are all continuous at the knots? This is the problem of computing a *spline interpolant*.

A *cubic spline* $S(x)$ is a piecewise cubic polynomial such that $S(x), S'(x), S''(x)$ are all continuous at the knots. $S(x)$ is a cubic polynomial on each particular subinterval $[x_i, x_{i+1}]$, denoted by $S_i(x)$, for $i = 1, ..., n-1$. The following interpolating conditions must hold for $i = 1, ..., n-1$:

$$\begin{aligned} S_i(x_i) &= y_i \\ S_i(x_{i+1}) &= y_{i+1} \end{aligned} \tag{2.4.2}$$

(which, we may note, ensures that the interpolant $S(x)$ is continuous at the knots) and, in addition, the following derivative continuity conditions must hold at the *interior* knots, for $i = 1, ..., n-2$:

$$S_i'(x_{i+1}) = S_{i+1}'(x_{i+1}) \tag{2.4.3}$$

$$S_i''(x_{i+1}) = S_{i+1}''(x_{i+1}). \tag{2.4.4}$$

At this point, we should ask how many unknowns must be determined and how many equations have been specified. Since there are four unknowns per subinterval (four parameters are required to determine each cubic polynomial) and there are $n-1$ subintervals, the total number of unknowns is $4n-4$. We have specified above $2(n-1)$ interpolating conditions and $2(n-2)$ derivative conditions, for a total of $4n-6$ equations. This means that in order to determine a unique cubic spline interpolant, we must specify two additional equations.

The two additional equations are typically specified by imposing some type of *boundary conditions* at $x = x_1$ and $x = x_n$. A *free boundary* has

$$S''(x_1) = 0 \quad \text{or} \quad S''(x_n) = 0\ . \tag{2.4.5}$$

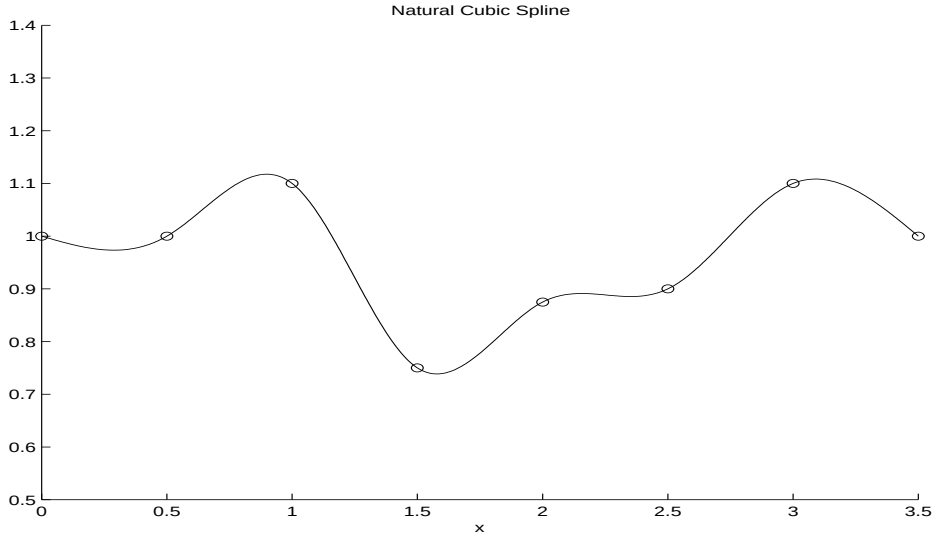
A *natural* cubic spline has free boundaries at both ends. A *clamped boundary* has

$$S'(x_1) = specified \quad \text{or} \quad S'(x_n) = specified\ . \tag{2.4.6}$$

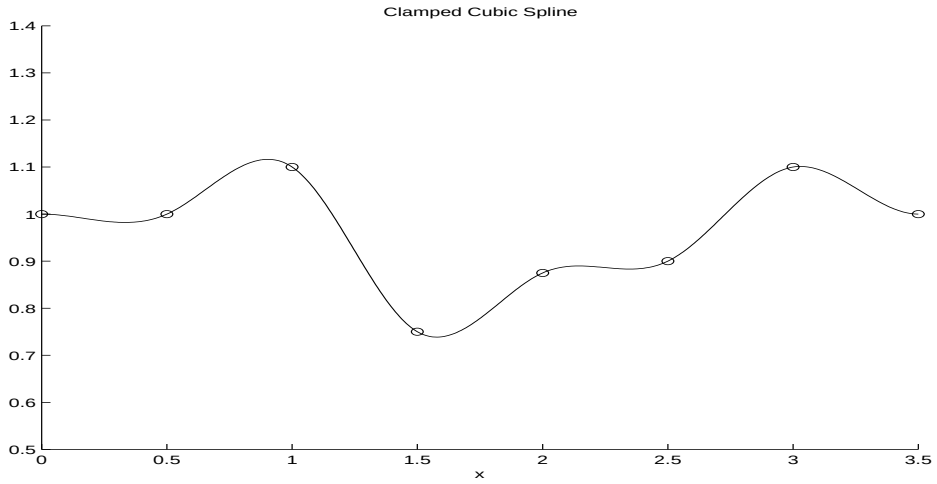
A *complete spline* or *clamped spline* has both ends clamped. A *periodic spline* has

$$\begin{aligned} S(x_1) &= S(x_n) \\ S'(x_1) &= S'(x_n) \\ S''(x_1) &= S''(x_n) \ . \end{aligned} \tag{2.4.7}$$

For example, a natural spline going through the 8 points $(0,1)$, $(\frac{1}{2},1)$, $(1,\frac{11}{10})$, $(\frac{3}{2},\frac{3}{4})$, $(2,\frac{7}{8})$, $(\frac{5}{2},\frac{9}{10})$, $(3,\frac{11}{10})$ and $(\frac{7}{2},1)$ given earlier looks like



while a complete spline (with horizontal clamps) is given by



Analytically, the natural spline for the points has the formula

$$S(x) = \begin{cases} 1.0 - 0.13820\,x + 0.55280\,x^3 & 0.0 \le x \le 0.5 \\ 1.3146 - 2.0258\,x + 3.7752\,x^2 - 1.9640\,x^3 & 0.5 \le x \le 1.0 \\ -3.5526 + 12.576\,x - 10.826\,x^2 + 2.9032\,x^3 & 1.0 \le x \le 1.5 \\ 13.835 - 22.200\,x + 12.358\,x^2 - 2.2488\,x^3 & 1.5 \le x \le 2.0 \\ -16.090 + 22.688\,x - 10.087\,x^2 + 1.4919\,x^3 & 2.0 \le x \le 2.5 \\ 30.954 - 33.765\,x + 12.495\,x^2 - 1.5189\,x^3 & 2.5 \le x \le 3.0 \\ -31.219 + 28.408\,x - 8.2297\,x^2 + 0.78379\,x^3 & 3.0 \le x \le 3.5 \end{cases}$$

while the clamped spline for the points has the formula

$$S(x) = \begin{cases} 1.0 - 0.47901\,x^2 + 0.95802\,x^3 & 0.0 \le x \le 0.5 \\ 1.3790 - 2.2741\,x + 4.0691\,x^2 - 2.0741\,x^3 & 0.5 \le x \le 1.0 \\ -3.6333 + 12.763\,x - 10.968\,x^2 + 2.9382\,x^3 & 1.0 \le x \le 1.5 \\ 13.974 - 22.453\,x + 12.509\,x^2 - 2.2789\,x^3 & 1.5 \le x \le 2.0 \\ -16.875 + 23.821\,x - 10.628\,x^2 + 1.5773\,x^3 & 2.0 \le x \le 2.5 \\ 36.366 - 40.068\,x + 14.928\,x^2 - 1.8302\,x^3 & 2.5 \le x \le 3.0 \\ -65.519 + 61.818\,x - 19.034\,x^2 + 1.9434\,x^3 & 3.0 \le x \le 3.5 \end{cases}$$

2.5 Efficient Computation of Splines

At this stage we can compute a cubic spline with any one of the above boundary conditions simply by setting up a system of $4n-4$ linear equations with $4n-4$ unknowns and solving the linear system using say Gaussian elimination. We shall learn in a later chapter that this comes at a cost of $O(n^3)$ operations. In this section we show how to determine the unknown variables in a cubic spline with a cost of $O(n)$ operations. This is basically the same cost as generating a linear spline.

Let $S_i(x)$ be expressed in the form used previously for a Hermite interpolant, namely

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \tag{2.5.1}$$

where $a_i = y_i, b_i = s_i, c_i = \dfrac{3y_i' - 2s_i - s_{i+1}}{\Delta x_i}$ and $d_i = \dfrac{s_{i+1} + s_i - 2y_i'}{(\Delta x_i)^2}$.

In this case the s_i values are *unknowns*. Note that by expressing $S_i(x)$ in the form of a Hermite interpolant, conditions (2.4.2-2.4.3) are automatically satisfied. It remains to force condition (2.4.4). Consequently, we must determine s_i such that condition (2.4.4) is satisfied. It follows from (2.5.1) that

$$S_i''(x) = 2\left(\frac{3y_i' - 2s_i - s_{i+1}}{\Delta x_i}\right) + 6\left(\frac{s_i + s_{i+1} - 2y_i'}{(\Delta x_i)^2}\right)(x - x_i) \tag{2.5.2}$$

$$S_{i+1}''(x) = 2\left(\frac{3y_{i+1}' - 2s_{i+1} - s_{i+2}}{\Delta x_{i+1}}\right) + 6\left(\frac{s_{i+1} + s_{i+2} - 2y_{i+1}'}{(\Delta x_{i+1})^2}\right)(x - x_{i+1}) \tag{2.5.3}$$

Now, setting $S_i''(x_{i+1}) = S_{i+1}''(x_{i+1}),\ i = 1, ..., n-2$ gives, using equations (2.5.2-2.5.3) along with some elementary algebra:

$$\Delta x_{i+1} s_i + 2(\Delta x_i + \Delta x_{i+1}) s_{i+1} + \Delta x_i s_{i+2} = 3(\Delta x_{i+1} y_i' + \Delta x_i y_{i+1}') \tag{2.5.4}$$

or, setting $i \to i-1$ in equation (2.5.4)

$$\begin{aligned} \Delta x_i s_{i-1} + 2(\Delta x_{i-1} + \Delta x_i) s_i + \Delta x_{i-1} s_{i+1} &= 3(\Delta x_i y'_{i-1} + \Delta x_{i-1} y'_i) \\ & \quad i = 2, ..., n-1 \ . \end{aligned} \tag{2.5.5}$$

This gives us $n-2$ equations for n unknowns. Two extra equations are determined from the boundary equations. For example, if we want to specify a complete spline (clamped at both ends) with specified slopes $s_1 = s_1^*, s_n = s_n^*$, then we would add the two equations

$$s_1 = s_1^* \tag{2.5.6}$$

$$s_n = s_n^* \tag{2.5.7}$$

to the equations (2.5.5). If we want to specify a free boundary at x_1

$$S_1''(x_1) = 0$$

then from equation (2.5.2) we have

$$2c_1 = 2\left(\frac{3y'_1 - 2s_1 - s_2}{\Delta x_1}\right) = 0$$

or

$$s_1 + \frac{s_2}{2} = \frac{3}{2} y'_1 \ . \tag{2.5.8}$$

Similarly, if we want to specify a free boundary at $x = x_n$

$$S''_{n-1}(x_n) = 0$$

then from equation (2.5.2) we have

$$2\left(\frac{3y'_{n-1} - 2s_{n-1} - s_n}{\Delta x_{n-1}}\right) + 6\left(\frac{s_{n-1} + s_n - 2y'_{n-1}}{\Delta x_{n-1}}\right) = 0$$

or

$$\frac{s_{n-1}}{2} + s_n = \frac{3}{2} y'_{n-1} \tag{2.5.9}$$

Suppose we have a set of points $x_1, ..., x_n$ with data values $y_1, ..., y_n$. Then in order to construct a spline interpolant in the form of equation (2.5.1), we need to determine the n unknowns $s_i, i = 1, ..., n$ by solving a linear system of equations given in matrix form by

$$T\vec{s} = \vec{r} \tag{2.5.10}$$

Regardless of boundary conditions, we construct the vector $\vec{r}$ of right-hand-sides from equation (2.5.5), namely for $i = 2, ..., n-1$:

$$r_i = 3(\Delta x_i y'_{i-1} + \Delta x_{i-1} y'_i) \tag{2.5.11}$$

Construct the matrix T with entries in the $i'th$ row (from equation (2.5.5)

$$\begin{aligned} T_{i,i-1} &= \Delta x_i \\ T_{i,i} &= 2(\Delta x_{i-1} + \Delta x_i) \\ T_{i,i+1} &= \Delta x_{i-1} \\ T_{i,k} &= 0 \ ; \ k \neq i-1, i, i+1 \\ & \quad \text{for } i = 2, ..., n-1 \ . \end{aligned} \tag{2.5.12}$$

The first and last rows of T and $\vec{r}$ depend on the boundary conditions. Note that $T_{i,j} = 0, \ j \neq i-1, i, i+1$, for $i = 2, ..., n-1$. If x_1 is a free boundary then from equation (2.5.8) we have

$$\begin{aligned} T_{1,1} &= 1 \\ T_{1,2} &= \frac{1}{2} \\ T_{1,k} &= 0 \ ; \ k \neq 1, 2 \\ r_1 &= \frac{3}{2} y_1' . \end{aligned}$$

If x_1 is a clamped boundary then (from equation (2.5.6))

$$\begin{aligned} T_{1,1} &= 1 \\ T_{1,k} &= 0 \ ; \ k \neq 1 \\ r_1 &= s_1^* . \end{aligned}$$

If x_n is a natural boundary then (from equation(2.5.9)) we have

$$\begin{aligned} T_{n,n} &= 1 \\ T_{n,n-1} &= \frac{1}{2} \\ T_{n,k} &= 0 \ ; \ k \neq n, n-1 \\ r_n &= \frac{3}{2} y_{n-1}' . \end{aligned}$$

If x_n is a clamped boundary (equation(2.5.7)) we have

$$\begin{aligned} T_{n,n} &= 1 \\ T_{n,k} &= 0 \ ; \ k \neq n \\ r_n &= s_n^* . \end{aligned}$$

In all cases, we must solve the matrix equation (2.5.10) for $s_1, ..., s_n$. Note that since T is diagonally dominant, a solution always exists provided the knots are distinct. T is a *tridiagonal* matrix and therefore the linear system can be solved very efficiently.

Once $s_1, .., s_n$ are known, the cubic spline is completely determined in the form of equation (2.5.1).

We leave it to the reader to figure out the first and last rows of T and r if periodic boundary conditions are imposed.

2.6 Spline Energy

Splines were originally flexible rods used by draftsman to draw smooth curves. The *energy* in a flexible elastic rod is given by

$$\int d''(x)^2 \, dx \tag{2.6.1}$$

where $d(x)$ is the displacement from an equilibrium position.

A natural spline approximation $S(x)$ to a function $y(x)$ corresponds to a spline rod which fits the data at the specified interpolating points (i.e. the spline rod is clamped at the data points), and has a total *energy* less than the original function. Another way of thinking about this is that the spline approximation has an average curvature smaller than the original function. To see this, let

$$\begin{aligned} g(x) &= y(x) - S(x) \\ g''(x) &= y''(x) - S''(x) \ . \end{aligned} \tag{2.6.2}$$

Note that $g(x_j) = g_j = 0$ for $j = 0, ..., n$ if x_j are the spline knots. Now

$$\int_{x_0}^{x_n} y''(x)^2 \, dx = \int_{x_0}^{x_n} S''(x)^2 \, dx + \int_{x_0}^{x_n} g''(x)^2 \, dx + 2 \int_{x_0}^{x_n} S''(x) g''(x) \, dx \tag{2.6.3}$$

Note that (integrating by parts)

$$\begin{aligned} \int_{x_0}^{x_n} S''(x) g''(x) \, dx &= [S''(x) g'(x)]_{x_0}^{x_n} - \int_{x_0}^{x_n} S'''(x) g'(x) \, dx \\ &= -\int_{x_0}^{x_n} S'''(x) g'(x) \, dx \quad \text{since this is a natural spline} \end{aligned} \tag{2.6.4}$$

Now, using equation (2.3.2) we get

$$\begin{aligned} \int_{x_0}^{x_n} S'''(x) g'(x) \, dx &= \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} S'''(x) g'(x) \, dx \\ &= \sum_{i=0}^{n-1} 6 d_i \int_{x_i}^{x_{i+1}} g'(x) \, dx \\ &= \sum_{i=0}^{n-1} 6 d_i \left[g_{i+1} - g_i \right] \\ &= 0 \end{aligned} \tag{2.6.5}$$

since $g_i = 0$ at the knots. Therefore, equations (2.6.4) and (2.6.5) imply

$$\int_{x_0}^{x_n} S''(x) g''(x) \, dx = 0 \tag{2.6.6}$$

and so from equations (2.6.6) and (2.6.3) we have

$$\begin{aligned}\int_{x_0}^{x_n} y''(x)^2\ dx &= \int_{x_0}^{x_n} S''(x)^2\ dx + \int_{x_0}^{x_n} g''(x)^2\ dx \\ &\geq \int_{x_0}^{x_n} S''(x)^2\ dx \end{aligned} \tag{2.6.7}$$

which shows that for a natural cubic spline we always have

$$\int_{x_0}^{x_n} S''(x)^2\ dx \leq \int_{x_0}^{x_n} y''(x)^2\ dx \tag{2.6.8}$$

2.7 B-Splines and Bézier Curves

While cubic splines are good for creating nice looking curves going through a set of interpolation points, they still have some limitations. In particular, it is often the case that one has points (x_0, y_0) ... (x_n, y_n) which give a rough drawing and one wishes to then create a smooth looking curve to provide the resulting picture. Unfortunately, when a picture is not quite correct looking one cannot simply change one of the interpolation points and expect to "correct" the curve. This is because changing one single interpolation point results in a global change to the curve.

The problem we have is that asking a curve to interpolate a set of points is too strong of a condition for creating a particular curve or image. Rather we need a combination of interpolation and *control* points. In this section we consider *Bézier curves* and *B-spline* curves in order to create curves which can be slightly corrected by moving a single control point.

2.7.1 Bernstein Polynomials

The Bernstein polynomials of degree N are defined by

$$B_{i,N}(t) = \left(\begin{array}{c} N \\ i \end{array} \right) t^i (1-t)^{N-i}, \quad i = 0, 1, ..., N. \tag{2.7.1}$$

For example, when $N = 3$ they are

$$B_{0,3}(t) = (1-t)^3, \quad B_{1,3}(t) = 3t(1-t)^2, \quad B_{2,3}(t) = 3t^2(1-t), \quad B_{3,3}(t) = t^3.$$

Properties of Bernstein Polynomials

(a) **Recurrence:**
$B_{i,N}(t)$ can be computed via the recurrence: $B_{0,0}(t) = 1$, $B_{i,N}(t) = 0$ for $i < 0$ or $i > N$ and otherwise

$$B_{i,N}(t) = (1-t)B_{i,N-1}(t) + tB_{i-1,N-1}(t).$$

(b) **Nonnegative on** $[0, 1]$**:**
$B_{i,N}(t)$ is nonnegative in interval $[0, 1]$ for all i.

(c) **Derivatives:**
$B'_{i,N}(t) = N(B_{i-1,N-1}(t) - B_{i,N-1}(t))$

(d) **Partition Unity:**
$\sum_{i=0}^{N} B_{i,N}(t) = 1.$

(e) **Polynomial Basis:**
Any polynomial of degree N can be written as a linear combination of the $B_{i,N}(t)$.

2.7.2 Bézier Curves

A Bézier curve for the points (x_0, y_0) ... (x_n, y_n) is given by

$$P(t) = \sum_{i=0}^{N} B_{i,N}(t)(x_i, y_i). \tag{2.7.2}$$

In terms of x and y coordinates we have

$$x(t) = \sum_{i=0}^{N} x_i B_{i,N}(t) \text{ and } y(t) = \sum_{i=0}^{N} y_i B_{i,N}(t).$$

Properties of Bézier Curves

(a) **End points interpolate:**

$$P(0) = (x_0, y_0), \quad P(1) = (x_n, y_n)$$

(b) **Derivatives at end points:**

$$P'(0) = N(x_1 - x_0, y_1 - y_0) \text{ and } P'(1) = N(x_n - x_{n-1}, y_n - y_{n-1})$$

(c) **Convexity:**
The Bézier curve lies in the convex hull of its set of control points [1]

The effectiveness of Bézier curves comes from the ease in which one can change the shape of the curve by simply changing the location of one of the interior control points. Changing the coordinates of any one control point, say (x_k, y_k) will change the entire curve over the interval $0 < t < 1$. However the change in shape tends to be localized since the Bernstein polynomial $B_{k,N}$ corresponding to (x_k, y_k) has a maximum at the parameter value $t = \frac{k}{N}$ and hence the majority of the changes will occur near the point $P(\frac{k}{N})$.

[1] Recall that a set S is said to be *convex* if the line segment joining any two points in S lies in S. The convex hull of a set S is the smallest convex set containing S.

2.7.3 B-spline Curves

Bézier curves are typically organized in pieces to create piecewise continuous curves. However they are also typically not smooth at the endpoints of each subinterval. B-splines are variations of Bézier curves which do have the property of smoothness. However this comes at a cost - namely these curves have no interpolation points. Rather all the points are control points. In this section we look at B-splines in the cubic case only.

Given a set of points $p_0 = (x_0, y_0)$, ... , $p_n = (x_n, y_n)$ a cubic B-spline for the i-th interval p_i to p_{i+1} is given by

$$B_i(u) = b_{-1}p_{i-1} + b_0 p_i + b_1 p_{i+1} + b_2 p_{i+2}$$

where ($u \in [0, 1]$)

$$b_{-1} = \tfrac{(1-u)^3}{6}, \qquad b_0 = \frac{u^3}{2} - u^2 + \frac{2}{3}$$
$$b_1 = -\tfrac{u^3}{2} + \tfrac{u^2}{2} + \tfrac{u}{2} + \tfrac{1}{6}, \quad b_2 = \frac{u^3}{6}.$$

Properties of Cubic B Spline Curves

- When the points are distinct then a B-spline looks like a spline curve in that both first and second derivatives are continuous. Indeed we have that

$$\begin{aligned} B_i(1) &= B_{i+1}(0) = \frac{p_i + 4p_{i+1} + p_{i+2}}{6}, \\ B_i'(1) &= B_{i+1}'(0) = \frac{-p_i + p_{i+2}}{2}, \\ B_i''(1) &= B_{i+1}''(0) = p_i - 2p_{i+1} + p_{i+2}. \end{aligned} \tag{2.7.3}$$

- When two interior points are equal then the curve and its derivative are still continuous at this point. When three interior points are equal then the curve is still continuous at this point. When four interior points are equal then the curve has a jump.

- The portion of the curve determined by each group of four points lies inside the convex hull of the four points.

- One way to handle end intervals is by attaching fictious points : p_{-2}, p_{-1} (both equal to p_0) and p_{n+1} and p_{n+2} (both equal to p_n).

- Differs from a Bézier curve because (1) it does not interpolate the end points, (2) there is no simple relationship for the derivatives of the end points.

2.7.4 Exercises for Interpolation

1. (a) Show that there is a unique cubic polynomial $p_3(x)$ for which

$$\begin{aligned} p_3(x_0) &= f(x_0), \quad p_3(x_2) = f(x_2) \\ p_3'(x_1) &= f'(x_1), \quad p_3''(x_1) = f''(x_1) \end{aligned}$$

   where $f(x)$ is a given function and $x_0 \neq x_2$.

   (b) Derive base functions analogous to Lagrange basis for $p_3(x)$.

2. For Lagrange polynomial interpolation, n data points $(x_i, y_i), i = 1, 2, ..., n$ are given.

   (a) What is the degree of each polynomial function $L_j(x)$ in the Lagrange basis?

   (b) What function results if we sum the n functions in the global Lagrange basis, that is, what is

$$g(x) = \sum_{j=1}^{n} L_j(x)$$

   Explain.

3. Let $S(x)$ be the natural spline interpolant of x^3 at $x = -3$, $x = -1$, $x = 1$ and $x = 3$. What is $S(0)$?

4. (a) If we have a natural spline:

$$S(x) = \begin{cases} a_1 + 25x + 9x^2 + x^3 & x \in [-3, -1] \\ 26 + a_2 x + a_3 x^2 - x^3 & x \in [-1, 0] \\ 26 + 19x + a_4 x^2 + a_5 x^3 & x \in [0, 3] \\ -163 + 208x - 60x^2 + a_6 x^3 & x \in [3, 4] \end{cases}$$

   Determine the values of a_1, a_2, $\cdots$, a_6.

   (b) Compute the values of $S(x)$ at -3, -1, 0, 3. Construct a cubic interpolation polynomial using the Lagrange basis. If you have troubles determining the a_i, use symbolic values of $S(x)$ to complete this part.

5. Given the function values $f(x_i) = f_i$, $i = 1, ..., n$, it is desired to interpolate this function with a quadratic spline of the form:

$$\begin{aligned} S_i(x) &= a_i(x_{i+1} - x) + b_i(x - x_i)(x - x_{i+1}) + c_i(x - x_i) \\ &\quad \text{for } x \in [x_i, x_{i+1}] \end{aligned}$$

   The spline conditions are:

$$\begin{aligned} S_i(x_i) &= f_i \ ; \ i = 1, ..., n \\ S_i(x_{i+1}) &= f_{i+1} \ ; \ i = 1, .., n-1 \\ S_{i+1}'(x_{i+1}) &= S_i'(x_{i+1}) \ ; \ i = 1, ..., n-2 \end{aligned}$$

with boundary condition (known derivative at left end).

$$S'(x_0) \quad = \quad \text{specified}$$

Determine the equations for the spline parameters a_i, b_i, c_i.

6. Is there a choice of coefficients for which the following function is a natural cubic spline? Explain why or why not.

$$S(x) = \begin{cases} x+1, & -2 \le x \le -1 \\ ax^3 + bx^2 + cx + d, & -1 \le x \le 1 \\ x - 1 & 1 \le x \le 2 \end{cases}$$

If $S(x)$ is a clamped spline instead of a natural spline, would there be a choice of coefficients? Explain why or why not.

Chapter 3

Planar Parametric Curves

In this chapter we give a short introduction to the concepts of parametric curves in two or three space along with an example where they are used.

A parametric curve in the x, y plane is a directed curve described by a pair of continuous functions, $x(t)$ and $y(t)$ of a common argument, t, usually called the parameter, for $a \leq t \leq b$. The curve, C, is the set of points $\vec{P}(t) = (x(t), y(t))$.

Examples

1. A semi-circle in the upper half plane directed from $(1, 0)$ to $(-1, 0)$.

$$x(t) = \cos(\pi t), \;\; y(t) = \sin(\pi t) \;\; 0 \leq t \leq 1$$

2. Same as above but with direction reversed.

$$x(t) = \cos(\pi(1-t)), \;\; y(t) = \sin(\pi(1-t)) \;\; 0 \leq t \leq 1$$

3. Same as above but with an alternative parametrization.

$$x(t) = \cos(\pi t^2), \;\; y(t) = \sin(\pi t^2) \;\; 0 \leq t \leq 1$$

4. The curve shown in Figure 3.1 for a particular value of the numerical coefficient, K.

$$x(t) = 1 + \tfrac{\cos(2\pi t)}{1+Kt}, \;\; y(t) = 0.8(1 + \tfrac{\sin(2\pi t)}{1+Kt}) \;\; 0 \leq t \leq 2$$

   What is the value of K?

Visually, it is clear that these example curves are smooth in some sense. Mathematically, this is reflected in the fact that all the derivatives of $x(t)$ and $y(t)$ exist, that is, the vector derivatives of $\vec{P}(t)$ exist for all k

$$\frac{d^k \vec{P}(t)}{dt^k} = \left(\frac{d^k x(t)}{dt^k}, \frac{d^k y(t)}{dt^k} \right) .$$

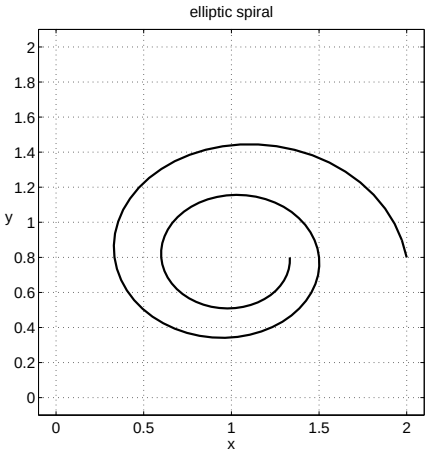


Figure 3.1: Curve number 4.

The tangent line to C at $t = t_0$ has the direction of the first derivative, $\frac{d\vec{P}(t_0)}{dt}$. This tangent is a line which can also be expressed as a parametric curve, $T_{tan}(s) = (x_{tan}(s), y_{tan}(s))$, where we have used a new parameter $-\infty < s < \infty$. Thus

$$T_{tan}(s) \quad = \quad \vec{P}(t_0) + \frac{d\vec{P}(t_0)}{dt}(s - t_0),$$

and consequently,

$$\begin{aligned} x_{tan}(s) &= x(t_0) + \frac{dx(t_0)}{dt}(s - t_0) \\ y_{tan}(s) &= y(t_0) + \frac{dy(t_0)}{dt}(s - t_0). \end{aligned}$$

A square can also be described as a parametric curve, using a piecewise definition.

$$\begin{aligned} x(t) = t, \quad & y(t) = 0 && 0 \le t \le 1 \\ x(t) = 1, \quad & y(t) = t - 1 && 1 \le t \le 2 \\ x(t) = 1 - (t - 2), \quad & y(t) = 1 && 2 \le t \le 3 \\ x(t) = 0, \quad & y(t) = 1 - (t - 3) && 3 \le t \le 4. \end{aligned} \tag{3.0.1}$$

Visually, this curve is not smooth. Mathematically, this is reflected in the fact that $x(t)$ and $y(t)$ do not have derivatives at $t = 1, 2, 3$.

3.1 Interpolating Curve Data by a Parametric Curve

Suppose we had a sequence of points that lie on a curve in the plane, (x_i, y_i), $i = 1, \ldots, n$. We can create a parametric curve that passes through these points if we can:

(a) identify suitable parameter values, t_i, $i = 1, \ldots, n$, and

(b) construct interpolating functions, $x(t)$ for data (t_i, x_i), and $y(t)$ for data (t_i, y_i).

One simple approach to step (a) is to regard the row index i as the sampled value of a real valued parameter, t, that ranges from $1 \leq t \leq n$, that is, take $t_i = i$ while for step (b), we could use piecewise linear spline interpolation. This would give us a function, $x_{pl}(t)$, for $0 \leq t \leq n$. If we did the same thing with the y coordinate data, then we would get a second function, $y_{pl}(t)$, for $0 \leq t \leq n$, and together $(x_{pl}(t), y_{pl}(t))$ would make a parametric curve passing through the data points. If we were to plot the original data using the Python command `plot(x,y)`, then we would see $(x_{pl}(t), y_{pl}(t))$.

However notice that such a curve would not necessarily be smooth. For a smoother interpolating curve, we could consider using cubic spline interpolation. That is, we interpolate (t_i, x_i), by a cubic spline function, $x_{cs}(t)$, and do the same for the y coordinate data. Then we would have a smooth interpolating parametric curve $(x_{cs}(t), y_{cs}(t))$.

How well would using a piecewise linear interpolating parametric curve work for representing the unit square curve (3.0.1)? How well would using a cubic spline interpolating parametric curve work for this same curve?

The simple idea for (a) of identifying the coordinate data with a parameter t in the range $1 \leq t \leq n$ and sample values of $t = i$ can lead to a poor looking curve in the case of cubic spline interpolation. A better idea is to let t be (approximately) the arc length distance to points on the curve, that is,

$$t_{i+1} = t_i + \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2} \ . \tag{3.1.1}$$

3.2 Graphics for Handwriting

Handwriting can be viewed as a series of curve segments in the plane. In this section we discuss the creation of a computer representation of handwriting based on interpolating parametric curves using cubic splines.

Consider the following procedure for representing a single script letter, or a word.

(a) Identify smooth curve segments that make up the script. For example, the handwritten form of a digit 3 normally has a non-smooth cusp in the middle. It could be broken up into 2 smooth curve segments. A script capital C is normally smooth enough to be a single curve segment, possibly a somewhat complex one. See Figure 3.2 for the construction of the course title.

(b) For each segment, capture suitable data points (x_i, y_i), $i = 1, \ldots, n$ on the segment, to roughly show its shape. Normally, this should take between 8 and 16 points per letter, depending on how ornate the letter is. Although this table of points will capture the shape adequately, plotting the table as a piecewise polynomial will produce a very crude image of the script letter; hence, steps (c) through (f).

(c) Introduce a parameter t_i for each data point as discussed in the preceding subsection.

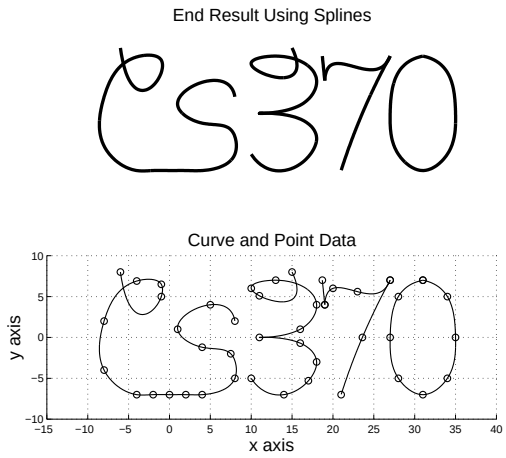


Figure 3.2: Constructing the CS 370 logo with splines

(d) Compute two cubic splines, one for the interpolating cubic spline of data (t_i, x_i), and one for data (t_i, y_i). We refer to these two spline functions as $x_{\rm cs}(t)$ and $y_{\rm cs}(t)$.

(e) Compute a finer partition, τ_j of the parameter interval $[t_1, t_n]$.

(f) Evaluate plotting points $(x_{\rm cs}(\tau_j), y_{\rm cs}(\tau_j))$.

This process has been followed to produce the CS 370 logo that appears on the title page of these notes, and other places. In Figure 3.2, we show the results of steps (a) and (b) in the lower figure, and the result of steps (e) and (f), plotted with no axes, in the upper figure.

Of course there are some details that still need to be done in order to complete the above task. For example, a simple way to accomplish steps (a) and (b) is to write a large version of the letter (or word) on squared graph paper. We then identify the segments and mark the data points on the curve segments. We introduce an x and a y axis on the paper which allows us to read off the coordinates of the sequence of points on each segment. It is efficient to pick a small number of data points chosen to catch the main features of the shape. If the curve segment is closed (for example, the letter O), then we repeat the first data point as the last one in the sequence to be sure that the interpolation parametric curve is closed.

In order to accomplish step (d) we first remark that some spline end conditions work better than others, depending on the curve, and its position in the letter. In general, this can only be determined by trial and error. However, for smooth closed curves, periodic end conditions should be used.

Chapter 4

Differential Equations

4.1 Introduction

In many physical situations one encounters the following problem: determine the behaviour of a quantity depending on a variable knowing only how the quantity changes with respect to the variable. For example, one may want a representation for the path of an object knowing only its starting location along with the knowledge that there are physical laws which relate the position, velocity and acceleration of the particular object.

Example 4.1.1 *An ecologist is studying the effects on the environment of field mice. With no limitation on food supply the population of mice can be modeled by*

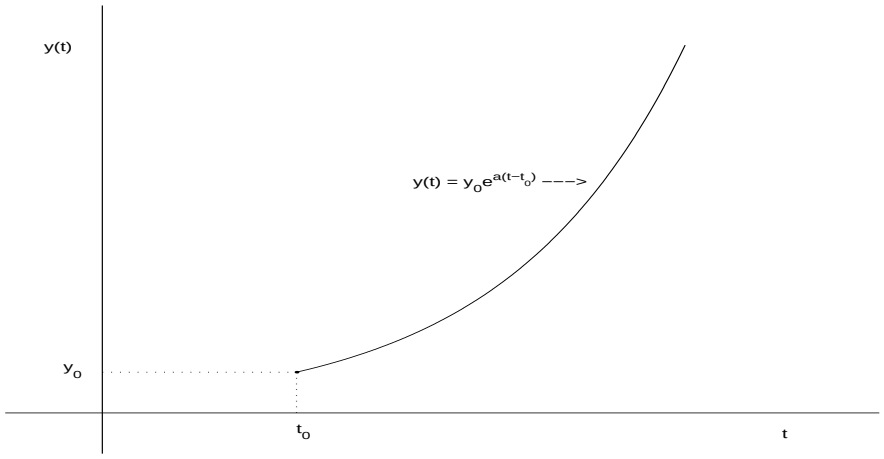
$$y'(t) = a \cdot y(t) \tag{4.1.1}$$

where $y(t)$ is the population of mice at time t and a is a constant (the net reproduction rate – determined by field experiments).

If for a given starting value (say at $t = t_0$) we have the population y_0, then it is not hard to determine that

$$y(t) = y_0 \cdot e^{a(t-t_0)} \tag{4.1.2}$$

is a solution to (4.1.1). Thus, the equation has "exponential" growth. In this case, we have a formula to determine a value for the population $y(t)$ at any time t.



The equation modeling the population of field mice is unrealistic since, in general, the food supply for a species is limited and hence a population will not continue to grow indefinitely. An alternate model for population growth is given by

$$y'(t) = y(t) \cdot (a - b \cdot y(t)) \tag{4.1.3}$$

where a and b are constants. Note that when $y(t)$ is small then

$$y'(t) \approx a \cdot y(t)$$

so that we have exponential growth for small populations. However, when $y(t) \approx \frac{a}{b}$ then

$$y'(t) \approx 0.$$

That is, the population stops growing and levels off.

In fact, there is again a closed-form solution for this initial value problem, given by

$$y(t) = \frac{a\ y_0\ e^{a(t-t_0)}}{b\ y_0\ e^{a(t-t_0)} + (a - y_0\, b)}.$$

A plot of this type of population growth, known as logistic growth, is shown in Figure 4.1.

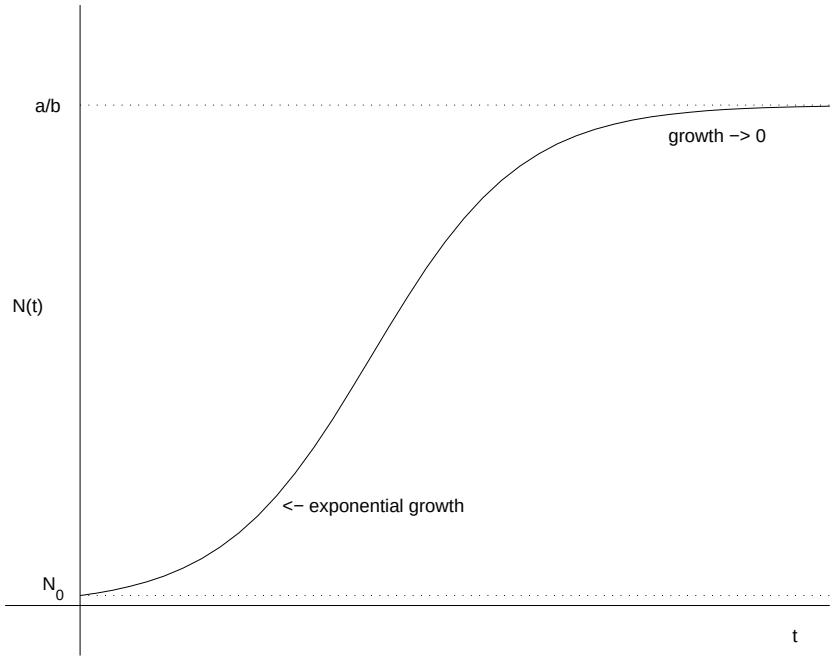


Figure 4.1: Logistic population growth. The initial population is N_0.

where again y_0 is the population at an initial time $t = t_0$.

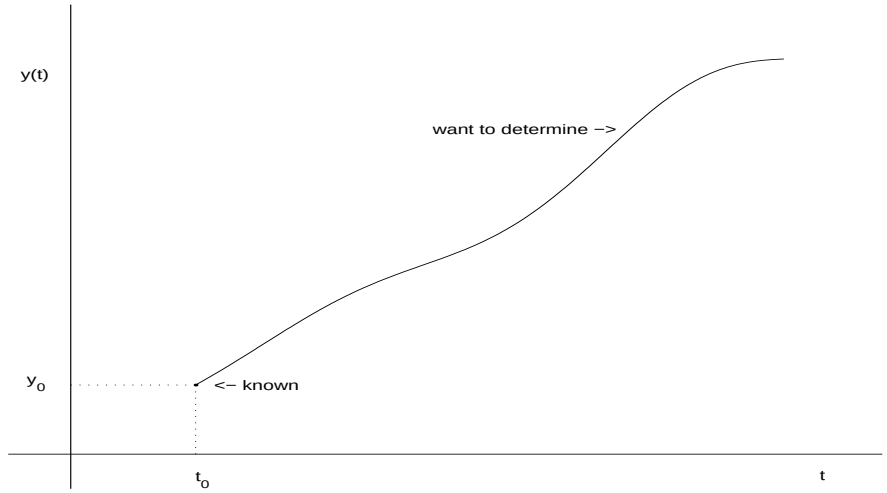
Unfortunately, both of these models are still too simple. For example, a good model of the population still needs to recognize that the birth rate of mice varies during the year as does the food supply. In addition, as we approach the carrying capacity (when

the population approaches $\frac{a}{b}$), then we expect the population growth to slow down in a different fashion. A more realistic model would then be

$$y'(t) = y(t) \cdot (a(t) - b(t) \cdot y(t)^{\alpha}), \quad y(t_0) = y_0 \tag{4.1.4}$$

where $a(t)$, $b(t)$ and α are all determined from field observations. With this more general model, it is not possible to find a closed form solution except for special $a(t)$ and $b(t)$. □

In the previous example one wants to predict $y(t)$ for all values of t. This was possible for the first two models since we had closed form solutions (in terms of exponential functions) while in the third model no closed form solution exists and hence other methods need to be used. In this case we can find numeric methods which will generate a type of "solution" to our problem.



In general we are interested in the problem of numerically solving an equation of the form

$$y'(t) = f(t, y(t)) \quad \text{with} \quad y(t_0) = y_0 \ . \tag{4.1.5}$$

Roughly speaking, the above equation describes a model of a particular quantity $y(t)$ depending on a given parameter t. The model states that there is an initial known starting value and has a description of how the quantity changes. For a fixed parameter value t, the changes depend only on the parameter and the quantity $y(t)$ at this parameter. Formally speaking, (4.1.5) specifies a function, $y(t)$; this specification is called the *initial value problem* , IVP, for $y(t)$. Its data are:

- the function of two variables, $f(t, z)$, sometimes called the dynamics function

- the values t_0, y_0, usually called the initial conditions.

For example, in the field mice population model, (4.1.3), $f(t, z) = z(a - bz)$; it is independent of t.

4.1.1 Other Differential Equations

There are other situations in which one wants to determine a particular set of quantities knowing only how those quantities change.

Example 4.1.2 *Let $\vec{P}(t) = (x(t), y(t), z(t))$ be the coordinates of an airplane, relative to some starting point. Let the velocity of the airplane $\vec{V}(t)$ be $(v_x(t), v_y(t), v_z(t))$ and the components of the acceleration $\vec{A}(t)$ be $(a_x(t), a_y(t), a_z(t))$.*

An inertial guidance system works by using accelerometers to detect acceleration . The system then solves the following **system** *of differential equations (in real time)*

$$\begin{array}{rclcrclcrcl} x'(t) & = & v_x(t), & & y'(t) & = & v_y(t), & & z'(t) = v_z(t), \\ v_x'(t) & = & a_x(t), & & v_y'(t) & = & a_y(t), & & v_z'(t) = a_z(t) \end{array} \tag{4.1.6}$$

where $x'(t) = \frac{dx}{dt}, v_x'(t) = \frac{dv_x}{dt}$ etc. At $t = 0$ we know that the airplane is at rest and so we also have the following initial conditions

$$(x(0), y(0), z(0)) = (x_0, y_0, z_0) \text{ and } (v_x(0), v_y(0), v_z(0)) = (0, 0, 0). \tag{4.1.7}$$

Equations (4.1.6) and (4.1.7) constitute a system of differential equations with an initial condition. Given the values of the x-, y-, and z-components of acceleration as a function of time, we can solve these equations to determine the position of the airplane at any time t. Note that it is crucial to specify the initial condition. □

Systems of differential equations also appear in such applications as robot control and pipeline leak detection. They also give us a way to look at higher-order equations.

Example 4.1.3 *Consider the 2nd-order linear differential equation given by*

$$y''(t) - t\ y'(t) + a\ y(t) = \sin(t), \text{ with } y(0) = y_0, y'(0) = 3.$$

We can write this equation as a first-order system of linear differential equations with variables $y(t)$ and $z(t) = y'(t)$ via

$$\begin{array}{rcl} y'(t) & = & z(t) \\ z'(t) & = & t\ z(t) - a\ y(t) + \sin(t) \end{array} \qquad \text{with } y(0) = y_0 \text{ and } z(0) = 3.$$

Example 4.1.4 *Now consider the following* two *2nd-order differential equations:*

$$\begin{aligned} x''(t) + x(t)\ y'(t) + x(t) &= y(t) \\ y''(t) - t\ y'(t) + y(t) &= \sin(t) \end{aligned}$$

$$\text{with } x(0) = x_0, x'(0) = 1, y(0) = y_0, y'(0) = 3.$$

We can write this as a linear system of first-order differential equations with four *variables. Let $z_1 = x$, $z_2 = y$, $z_3 = x'$, and $z_4 = y'$. Then we have the system*

$$\begin{aligned} z_1'(t) &= z_3(t) \\ z_2'(t) &= z_4(t) \\ z_3'(t) &= z_2(t) - z_1(t)\ z_4(t) - z_1(t) \\ z_4'(t) &= t\ z_4(t) - z_2(t) + \sin(t) \end{aligned}$$

$$\textit{with } z_1(0) = x_0, z_2(0) = y_0, z_3(0) = 1, z_4(0) = 3.$$

In general, a higher order differential equation is of the form

$$y^{(n)}(t) = f(t, y(t), y'(t), \ldots, y^{(n-1)}(t)) \tag{4.1.8}$$

with initial conditions specified for the first $n-1$ derivatives at an initial value t_0. Again such a system can be written as a system of first order equations with initial conditions. Using vector notation, the system has the form

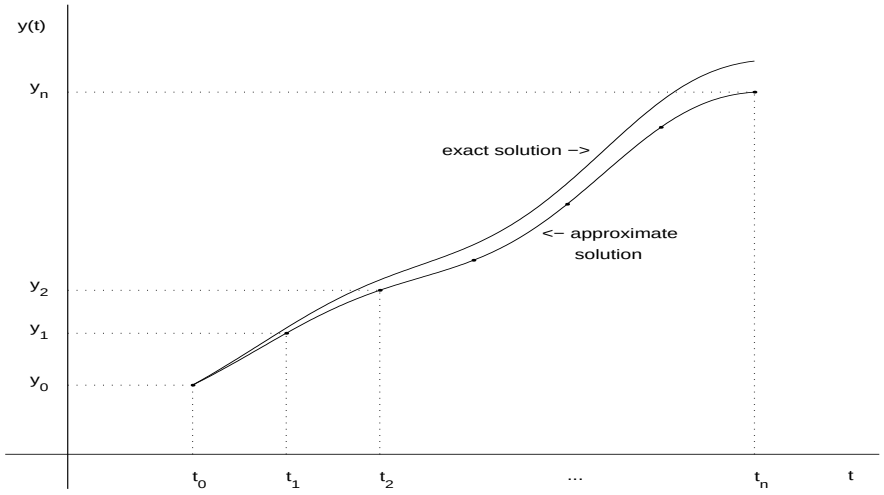
$$\mathbf{x}'(t) = \mathbf{f}(t, \mathbf{x}(t)), \quad \mathbf{x}(0) = \mathbf{x}_0.$$

See Chapter **??** for a further discussion of systems of differential equations and vector notation for them.

One can also define partial differential and systems of partial differential equations where there are quantities that depend on more than one variable. These appear in applications such as animation (for example the water waves in the movie Titanic), design of aircraft, pricing of options and hedging in finance, weather prediction and the effect of greenhouse gases on the environment.

4.2 Approximating Methods

Equation (4.1.5) is said to be a first order differential equation with an initial condition. We assume that no closed-form solution exists for such an equation and hence we need to obtain a numerical solution. A numerical solution for equation (4.1.5) means that we need to choose a set of times $t_0 < t_1 < \cdots < t_N$ at which we estimate the value of the solution, $y_0, y_1, \ldots, y_N$. Our hope is that y_n will be "close" to the true value of $y(t_n)$ for each n. Such a numerical solution will allow us to produce a plot of our function $y(t)$ in a particular interval and to approximate any value of the function (say through a process such as piecewise polynomial interpolation or spline interpolation).



The most common class of methods for determining a numerical solution for a first order initial value problem are called time stepping methods. A time step is the interval

$h_n = t_{n+1} - t_n$, which is determined by the method. Time stepping methods carry a candidate size h^{cand} for the next time step which may be revised during each time step. They have the following general form:

initialize y_0, t_0, h^{cand}, $n = 0$

repeat

i) compute y_{n+1}, and h_n using data $t_n,\ y_n,\ h^{cand}$ and $f(t, z)$
ii) $t_{n+1} \leftarrow t_n + h_n$ (4.2.1)
iii) recompute h^{cand}
iv) $n \leftarrow n + 1$

Step i) combines both advancing the solution (computing $y^{(n+1)}$) and time step size selection (computing h_n), although step iii) is also part of time step size selection.

Methods for advancing the solution can be classified into groups several ways; one such classification is

- single-step methods (where y_{n+1} is determined from (t_n, y_n) and the dynamics function, f,

- multi-step methods (where y_{n+1} is determined from (t_{n-i}, y_{n-i}) and f for $i = 0, 1, \ldots, N)$ with $N > 0$ (number of steps).

They can also be classified into *explicit* and *implicit* methods. Explicit methods are those with y_{n+1} given in terms of (t_{n-i}, y_{n-i}) and the equation for the derivative for $i = 0, 1, \ldots$ while implicit equations are those where y_{n+1} is computed by solving an algebraic equation involving f.

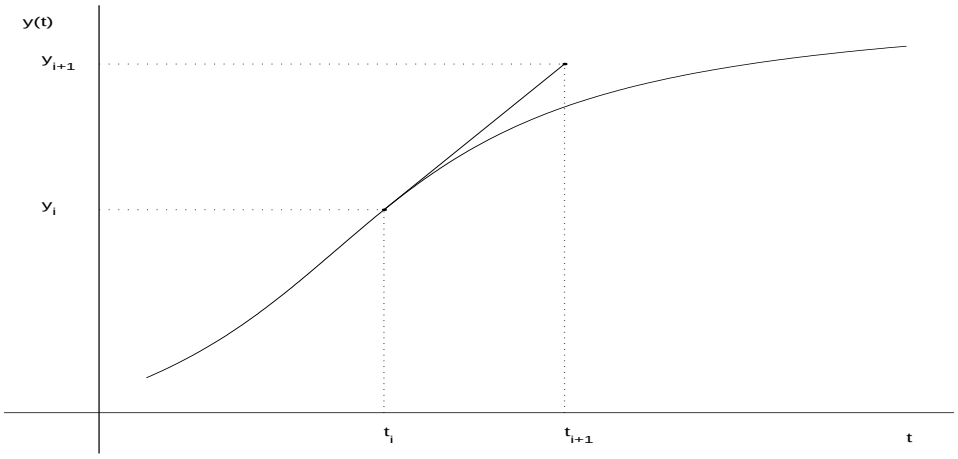
4.2.1 The Forward Euler Method

In this subsection we will focus on the solution step of the approximation process. Thus, we have the initial value problem

$$y'(t) = f(t, y(t)) \text{ with } y(t_0) = y_0,$$

and we assume that we have a given set of t points $t_0 < t_1 < \cdots < t_N$ with our problem being to find the y_i.

The Forward Euler method is the simplest technique for obtaining a numerical approximation to a first order initial value problem. Given a discrete set of points $t_0 < t_1 < \cdots < t_N$ the method uses slope as an approximation to the derivative and then develops a recursive scheme for determining the y_n values.



for each $n = 0, \ldots, N-1$ we make use of the approximation

$$\frac{y(t_{n+1}) - y(t_n)}{t_{n+1} - t_n} \approx y'(t_n) = f(t_n, y(t_n)).$$

We can rearrange the terms on this approximate relation to read as

$$y_0 = y(t_0) \text{ and } y(t_{n+1}) \approx y(t_n) + f(t_n, y(t_n))(t_{n+1} - t_n) \text{ for } n = 0, \ldots, N-1 \quad (4.2.2)$$

If we replace the approximation symbol $\approx$ by an assignment $=$, we get a method for advancing the numerical solution

$$y_0 = y(t_0) \text{ and } y_{n+1} = y_n + f(t_n, y_n)(t_{n+1} - t_n) \text{ for } n = 0, \ldots, N-1 \quad (4.2.3)$$

which is called the *Forward Euler* method.

Example 4.2.1 *Consider the Forward Euler method when applied to the initial value problem*

$$y'(t) = y(t)\ (2.5\ t - t^2\ \sqrt{y(t)}),\ \textit{with } y(0) = 1.$$

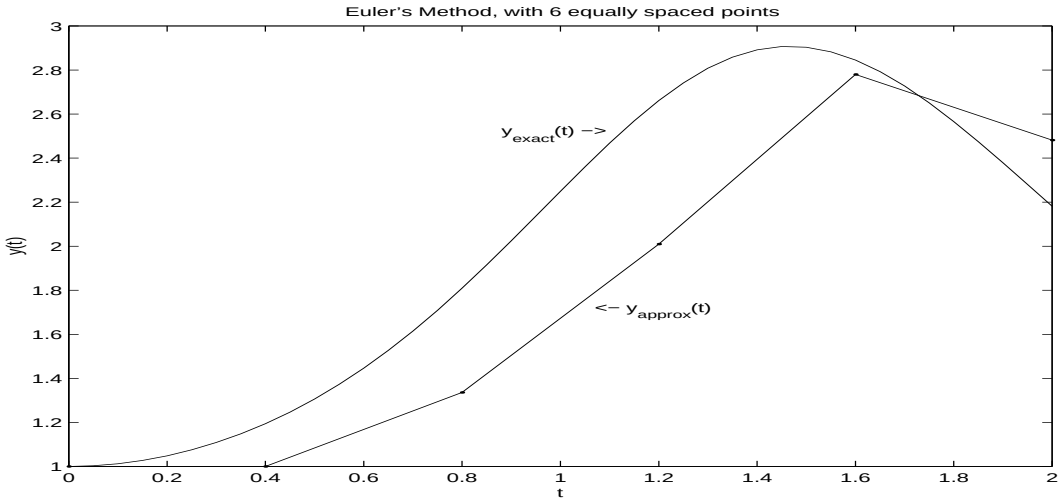
This is the same as equation (4.1.4) of Example 4.1.1 with $a(t) = 2.5t$, $b(t) = -t^2$ *and* $\alpha = \frac{1}{2}$. *We can obtain approximations to* $y(t)$ *for* $t = 0.4, 0.8, 1.2, 1.6$ *and* 2.0 *by using the recursive scheme*

$$y(0) = 1 \textit{ and } y_{n+1} = y_n + y_n \cdot (2.5\ t_n - {t_n}^2\ \sqrt{y_n}) \cdot 0.4 \textit{ for } n = 0, \ldots, 4.$$

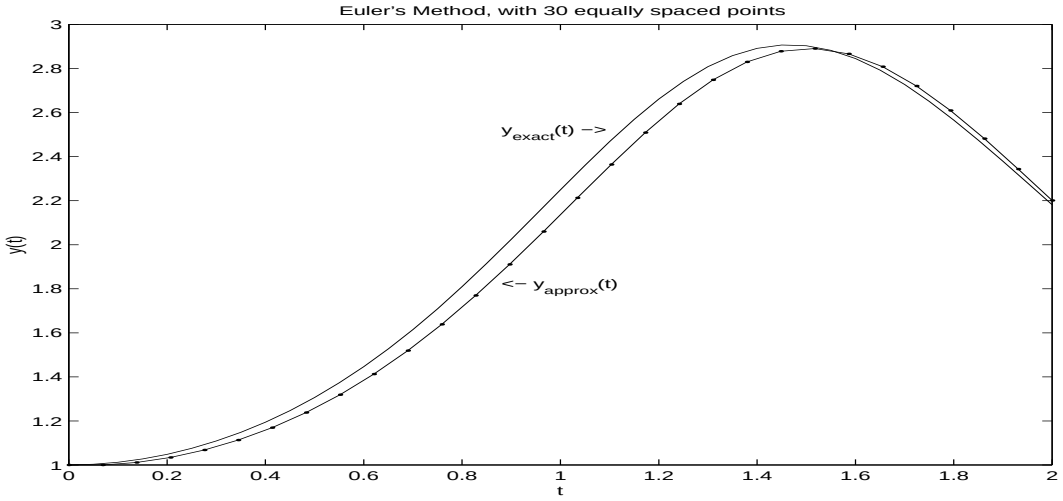
This gives the numerical solution

n	t_n	y_n
0	0	1.0
1	0.4	1.0
2	0.8	1.34
3	1.2	2.01
4	1.6	2.78

Graphically, this is given by



Clearly one would expect that more t values would provide a more accurate picture. For example, with a subdivision of 30 *equally spaced points one would get*



Example 4.2.2 *In this example, we look at an initial value problem for a system of two differential equations.*

$$\begin{aligned} \frac{dx(t)}{dt} &= x(t)\ (a - \alpha y(t)), \\ \frac{dy(t)}{dt} &= y(t)\ (-b + \beta x(t)), \end{aligned}$$

with $x(0) = x_0$ and $y(0) = y_0$ and a, b, α, β all positive constants. This is an example of a predator-prey *system where x is the prey and y is the predator. We will see here how we can use the same numerical method for this system of differential equations.*

The two-dimensional recursion for Forward Euler in this case would be given by

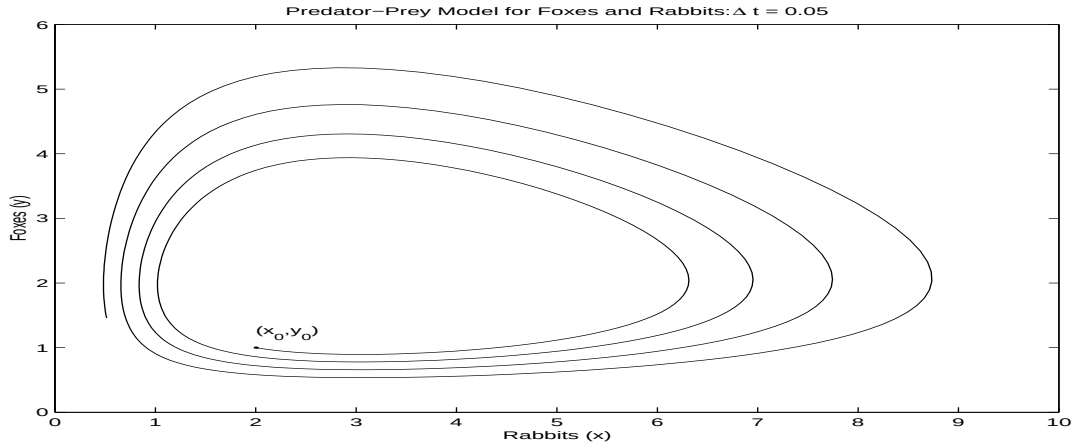
$$\begin{aligned} x_{n+1} &= x_n + x_n\ (a - \alpha y_n)\ h, \\ y_{n+1} &= y_n + y_n\ (-b + \beta x_n)\ h \end{aligned}$$

with $h = t_{n+1} - t_n$. Assume that we are studying foxes as predators and rabbits as prey,

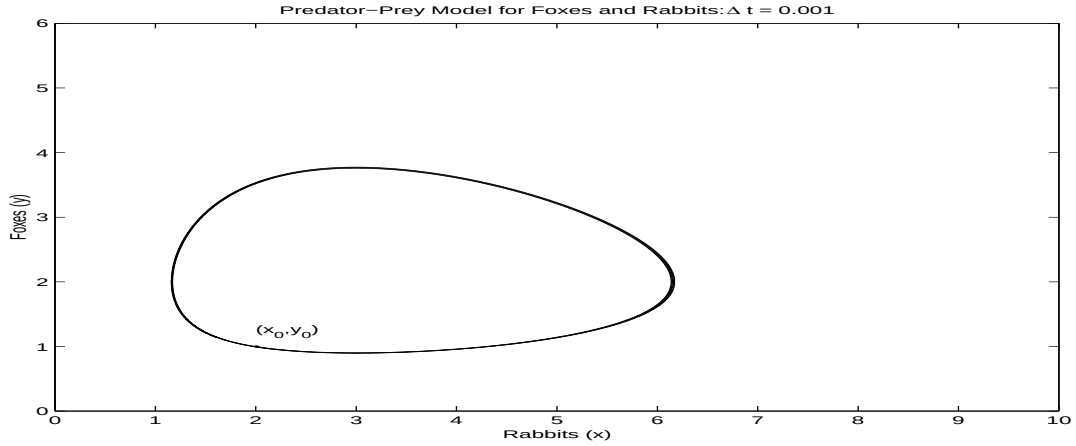
and that we obtain some field data where

$$a = 1, \alpha = 0.5, b = 0.75, \beta = 0.25.$$

Then with initial conditions: $x_0 = 2$, *(rabbits per hectare) and* $y_0 = 1$ *(foxes per* 100 *hectares) we obtain the following graph of our population of rabbits and foxes (using* 600 *timesteps)*



In this case we can increase the number of timesteps to again get a more accurate picture of how the populations change. In particular, note that when there are $30,000$ *timesteps then the populations appear to cycle, that is, they become periodic.*



Example 4.2.3 *Consider the second order differential equation*

$$y''(t) - ty'(t) + ay(t) = sin(t) \text{ with } y(0) = 1,\ y'(0) = 3$$

that was given in Example 4.1.3. In this case the equation was converted to the first order system $\mathbf{x}'(t) = \mathbf{f}(t, \mathbf{x}(t))$ *given by*

$$\begin{bmatrix} y(t) \\ z(t) \end{bmatrix}' = \begin{bmatrix} z(t) \\ tz(t) - ay(t) + sin(t) \end{bmatrix} \quad with \begin{bmatrix} y(0) \\ z(0) \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

where $\mathbf{x}(t) = (y(t), z(t))$ *and* $z(t) = y'(t)$. *A Forward Euler iteration is then given by*

$$\begin{bmatrix} y_{n+1} \\ z_{n+1} \end{bmatrix} = \begin{bmatrix} y_n \\ z_n \end{bmatrix} + h \begin{bmatrix} z_n \\ t_n z_n - a y_n + sin(t_n) \end{bmatrix} \quad with \begin{bmatrix} y_0 \\ z_0 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}.$$

4.2.2 Discrete Approximations

In the last subsection, we presented a recursive scheme based on replacing a derivative by a slope. It is clear that the smaller the interval the better the slope is at approximating a derivative. However it is not clear at this stage how good such an approximation needs to be in order to obtain acceptable answers. In this subsection, we use Taylor expansions to get an idea of the order of the approximation and to obtain new, more accurate approximations.

Recall that for any function $y(t)$, we can do a Taylor expansion about the point $t = a$:

$$y(t) = y(a) + y'(a)h + \frac{y''(a)}{2}h^2 + \frac{y'''(a)}{6}h^3 + \cdots \tag{4.2.4}$$

where $h = t - a$. We can be more precise about what happens when we truncate after a finite number of terms, in which case we have

$$y(t) = y(a) + y'(a)h + \frac{y''(a)}{2}h^2 + \frac{y'''(a)}{6}h^3 + \cdots + \frac{y^{(p)}(a)}{p!}h^p + \frac{y^{(p+1)}(\eta_t)}{(p+1)!}h^{p+1} \tag{4.2.5}$$

where η_t is a point between a and t. Often we write equation (4.2.5) as

$$y(t) = y(a) + y'(a)h + \frac{y''(a)}{2}h^2 + \frac{y'''(a)}{6}h^3 + \cdots + \frac{y^{(p)}(a)}{p!}h^p + O(h^{p+1}) \ . \tag{4.2.6}$$

If $a = t_n$ and $t = t_{n+1}$ and $p = 1$, then equation (4.2.6) becomes

$$y(t_{n+1}) = y(t_n) + y'(t_n)h + O(h^2) \ . \tag{4.2.7}$$

which can also be written

$$y'(t_n) = \frac{y(t_{n+1}) - y(t_n)}{h} + O(h) \ . \tag{4.2.8}$$

This is called a *forward difference* approximation to $y'(t_n)$ since we are using information at $t = t_n$ and the forward point $t = t_{n+1}$. Since the error term is $O(h)$ we say that the approximation is a *first order* approximation. Intuitively this says that the error is reduced proportionally to h as $h \to 0$.

Since $y'(t) = f(t, y(t))$, we can write (4.2.7) as

$$y(t_{n+1}) = y(t_n) + f(t_n, y(t_n))h + O(h^2)$$

which we recognize as the Forward Euler formula (4.2.3). The term $O(h^2)$ is called the *local truncation error* for our formula.

Technical Note: If we have a function of two variables $f(x, y)$, then the Taylor series expansion for $f(x + h_x, y + h_y)$ is

$$f(x + h_x, y + h_y) \ = \ f(x, y) + \frac{\partial f(x, y)}{\partial x}h_x + \frac{\partial f(x, y)}{\partial y}h_y + O(h_x^2) + O(h_y^2) + O(h_x h_y) \ . \tag{4.2.9}$$

4.2.3 The Trapezoidal and Modified Euler Methods

Notation: In the following, $y(t_n)$ denotes the exact solution of our differential equation at $t = t_n$ while y_n will denote the approximate solution at $t = t_n$.

In order to avoid lots of tedious notation, let us set $h = t_{n+1} - t_n$. Then the Taylor series expansions give

$$y(t_{n+1}) = y(t_n) + y'(t_n)h + \frac{y''(t_n)}{2}h^2 + \frac{y'''(\eta)}{6}h^3$$

where $\eta \in [t_n, t_{n+1}]$. We can replace $y''(t_n)$ by a first order derivative approximation

$$y''(t_n) = \frac{y'(t_{n+1}) - y'(t_n)}{h} + O(h) \tag{4.2.10}$$

to obtain

$$\begin{aligned} y(t_{n+1}) &= y(t_n) + hy'(t_n) + \left[\frac{y'(t_{n+1}) - y'(t_n)}{h} + O(h)\right]\frac{h^2}{2} + O(h^3) \\ &= y(t_n) + h\left(\frac{y'(t_{n+1}) + y'(t_n)}{2}\right) + O(h^3). \end{aligned}$$

Now, $y(t)$ satisfies the differential equation, i.e. $y'(t) = f(t, y(t))$ so that we can write

$$y'(t_n) = f(t_n, y(t_n)) \tag{4.2.11}$$

and

$$y'(t_{n+1}) = f(t_{n+1}, y(t_{n+1})). \tag{4.2.12}$$

Thus we have

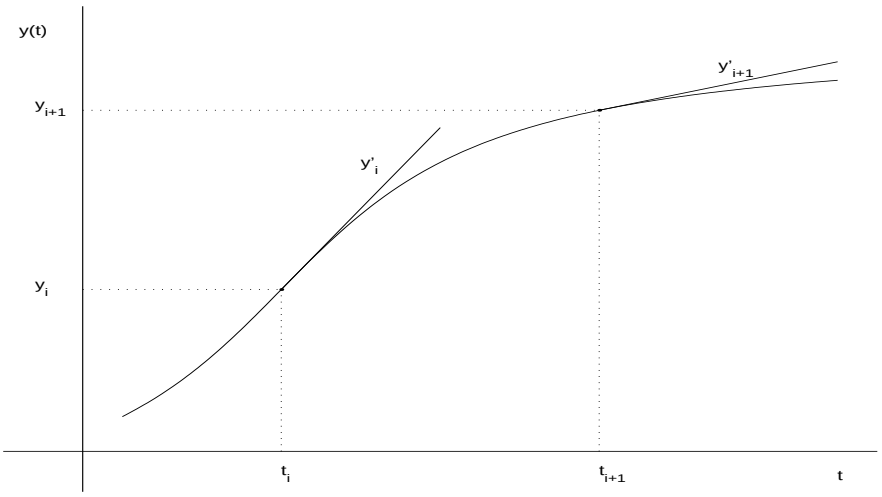
$$y(t_{n+1}) = y(t_n) + \frac{h}{2}[f(t_n, y(t_n)) + f(t_{n+1}, y(t_{n+1}))] + O(h^3). \tag{4.2.13}$$

This suggests that we define a method of advancing the solution by requiring y_{n+1} to satisfy

$$y_{n+1} = y_n + \frac{h}{2}(f(t_n, y_n) + f(t_{n+1}, y_{n+1})) \tag{4.2.14}$$

which is called the trapezoidal rule (or Crank-Nicolson) method.

We may think of (4.2.13) as approximating the derivative $y'(t)$ at a point half way between t_n, t_{n+1} by using the average of the slopes at the two end points, that is,



Since this is a centered approximation we expect a higher order method (that is, a higher power in the truncation error term). Thus we have the local error of Forward Euler as

$$LocErr_{FE} = y(t_{n+1}) - (y(t_n) + hf(t_n, y(t_n)) = O(h^2) \tag{4.2.15}$$

and the local error of the trapezoidal method as

$$LocErr_{trap} = y(t_{n+1}) - y_{t_{n+1}} = O(h^3). \tag{4.2.16}$$

A proof of the result in (4.2.16) follows.

$$\begin{aligned} LocErr_{trap} &= y(t_{n+1}) - \left(y(t_n) + \frac{h}{2}[f(t_n, y(t_n)) + f(t_{n+1}, y_{t_{n+1}})] \right) \\ &= y(t_{n+1}) - \left(y(t_n) + \frac{h}{2}[y'(t_n) + f(t_{n+1}, y_{t_{n+1}})] \right). \end{aligned}$$

Then we expand $y(t_{n+1})$ using its Taylor expansion around t_n

$$y(t_{n+1}) = y(t_n) + y'(t_n)h + y''(t_n)\frac{h^2}{2} + y'''(t_n)\frac{h^3}{6} + O(h^4),$$

and replace the expansion in $LocErr_{trap}$ as follows

$$\begin{aligned} LocErr_{trap} &= y(t_n) + y'(t_n)h + y''(t_n)\frac{h^2}{2} + y'''(t_n)\frac{h^3}{6} + \mathcal{O}(h^4) \\ &\quad - \left(y(t_n) + \frac{h}{2}[y'(t_n) + f(t_{n+1}, y_{t_{n+1}})] \right). \\ &= y'(t_n)h + y''(t_n)\frac{h^2}{2} + y'''(t_n)\frac{h^3}{6} - \frac{h}{2}\left(y'(t_n) + f(t_{n+1}, y_{t_{n+1}}) \right) + O(h^4) \\ &= y'(t_n)\frac{h}{2} + y''(t_n)\frac{h^2}{2} + y'''(t_n)\frac{h^3}{6} - \frac{h}{2}f(t_{n+1}, y_{t_{n+1}}) + O(h^4). \end{aligned}$$

Furthermore, we use (4.2.10) to get

$$\begin{aligned} LocErr_{trap} &= y'(t_n)\frac{h}{2} + \left(\frac{y'(t_{n+1}) - y'(t_n)}{h} + O(h)\right)\frac{h^2}{2} + y'''(t_n)\frac{h^3}{6} - \frac{h}{2}f(t_{n+1}, y_{t_{n+1}}) \\ &\quad + O(h^4) \\ &= y'(t_{n+1})\frac{h}{2} - f(t_{n+1}, y_{t_{n+1}})\frac{h}{2} + y'''(t_n)\frac{h^3}{6} + O(h^3) + O(h^4) \\ &= \left(y'(t_{n+1}) - f(t_{n+1}, y_{t_{n+1}})\right)\frac{h}{2} + y'''(t_n)\frac{h^3}{6} + O(h^3) + O(h^4) \\ &= \left(f(t_{n+1}, y(t_{n+1})) - f(t_{n+1}, y_{t_{n+1}})\right)\frac{h}{2} + y'''(t_n)\frac{h^3}{6} + O(h^3) + O(h^4). \end{aligned} \tag{4.2.17}$$

Note that all terms in the last equation are of order 3 or larger except of the first term, which seems to be of order 1. To proceed further we need to assume that function f is Lipschitz continuous, that is,

$$|f(t_{n+1}, y(t_{n+1})) - f(t_{n+1}, y_{t_{n+1}})| \leq L|y(t_{n+1}) - y_{t_{n+1}}| = L \cdot LocErr_{trap},$$

where L is a constant. Using the above inequality in (4.2.17) we get

$$\begin{aligned} LocErr_{trap} &= \frac{hL}{2}LocErr_{trap} + y'''(t_n)\frac{h^3}{6} + O(h^3) + O(h^4) \\ &= \frac{hL}{2}LocErr_{trap} + O(h^3), \end{aligned}$$

which is equivalent to

$$LocErr_{trap} = \frac{O(h^3)}{(1 - \frac{hL}{2})}.$$

Observe that for small h we have that $LocErr_{trap} = O(h^3)$, which is the desired result.

Note that for the Trapezoidal method, y_{n+1} appears only on the left hand side in the Forward Euler method. This is called an *explicit* method. On the other hand, equation (4.2.14) is an *implicit* method because the y_{n+1} appears on both sides of the equation (and hence one needs to do extra work to isolate this quantity in order to proceed with the algorithm. In other words, we have to solve a nonlinear equation for y_{n+1}).

One technique for handling the implicit equation (4.2.14) is to combine it with the Forward Euler recursion. That is we use (4.2.3) as a first approximation for y_{n+1} and then use this value in the right hand side of (4.2.14) for our final value of y_{n+1}. This method is called the *Modified Euler method* (also sometimes called the *Improved Euler method*). Let us proceed a bit more formally here. Going back to the trapezoidal method

$$y(t_{n+1}) \;\; = \;\; y(t_n) + \frac{h}{2}[f(t_n, y(t_n)) + f(t_{n+1}, y(t_{n+1})] + O(h^3) \tag{4.2.18}$$

while from the forward Euler approximation we have

$$y(t_{n+1}) = y(t_n) + hf(t_n, y(t_n)) + O(h^2).$$

Setting

$$y^*_{n+1} = y(t_n) + hf(t_n, y(t_n)) \tag{4.2.19}$$

gives

$$y(t_{n+1}) - y^*_{n+1} = O(h^2). \tag{4.2.20}$$

Now

$$f(t_{n+1}, y(t_{n+1})) = f(t_{n+1}, y^*_{n+1}) + \frac{\partial}{\partial y} f(t_{n+1}, y^*_{n+1})(y(t_{n+1}) - y^*_{n+1}) + O((y(t_{n+1}) - y^*_{n+1})^2) \tag{4.2.21}$$

which implies that

$$f(t_{n+1}, y(t_{n+1})) = f(t_{n+1}, y^*_{n+1}) + O(h^2). \tag{4.2.22}$$

Therefore, plugging (4.2.22) into (4.2.18) gives

$$y(t_{n+1}) \;=\; y(t_n) + \frac{h}{2}(f(t_n, y(t_n)) + f(t_{n+1}, y^*_{n+1})) + O(h^3). \tag{4.2.23}$$

Thus we have that

$$y^*_{n+1} \;=\; y_n + hf(t_n, y_n) \tag{4.2.24}$$

$$y_{n+1} \;=\; y_n + \frac{h}{2}(f(t_n, y_n) + f(t_{n+1}, y^*_{n+1})) \tag{4.2.25}$$

is an explicit method with a higher order truncation error than Forward Euler.

Technical Note: The local truncation error is exactly that - only a local measure of the error. In other words, if we assume that $y_n = y(t_n)$, the exact solution at $t = t_n$, then the local error is the error introduced just in this one step. In Section A, we give some examples of how to determine the local truncation error, using a systematic approach.

Example 4.2.4 *Consider the predator-prey system we saw in Example 4.2.2:*

$$\begin{bmatrix} x'(t) \\ y'(t) \end{bmatrix} = \begin{bmatrix} x(t)\ (a - \alpha y(t)) \\ y(t)\ (-b + \beta x(t)) \end{bmatrix}, \quad \begin{bmatrix} x(0) \\ y(0) \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}$$

with constants $a, b, \alpha, \beta > 0$. By applying the Modified Euler method, we obtain the two-dimensional combined iteration steps,

$$\begin{bmatrix} x^*_{n+1} \\ y^*_{n+1} \end{bmatrix} = \begin{bmatrix} x_n \\ y_n \end{bmatrix} + h \begin{bmatrix} x_n(a - \alpha y_n) \\ y_n(-b + \beta x_n) \end{bmatrix} \quad \textit{(Forward Euler)}$$

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} x_n \\ y_n \end{bmatrix} + \frac{h}{2} \begin{bmatrix} x_n(a - \alpha y_n) + x^*_{n+1}(a - \alpha y^*_{n+1}) \\ y_n(-b + \beta x_n) + y^*_{n+1}(-b + \beta x^*_{n+1}) \end{bmatrix} \quad \textit{(Trapezoidal)}$$

where $h = t_{n+1} - t_n$. This higher order method improves on the Forward Euler method, producing more accurate results than those obtained in Example 4.2.2.

4.2.4 Runge-Kutta Methods

We can also write this numerical method in the form

$$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_n + h, y_n + k_1) \\ y_{n+1} &= y_n + \tfrac{k_1}{2} + \tfrac{k_2}{2}. \end{aligned}$$

This is an example of a **Runge-Kutta** method. There are many other possibilities for deriving explicit Runge-Kutta methods which have local error $O(h^3)$ due to truncation. For example, we have the *midpoint* Runge-Kutta method given by

$$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_n + \tfrac{h}{2}, y_n + \tfrac{k_1}{2}) \\ y_{n+1} &= y_n + k_2. \end{aligned}$$

which is also of order $O(h^3)$. One can continue on with such ideas in order to obtain methods which have local truncation error $O(h^\alpha)$ for $\alpha = 4, 5, 6, \ldots$. The best known example of a higher order Runge-Kutta method is:

$$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_n + \tfrac{h}{2}, y_n + \tfrac{k_1}{2}) \\ k_3 &= hf(t_n + \tfrac{h}{2}, y_n + \tfrac{k_2}{2}) \\ k_4 &= hf(t_n + h, y_n + k_3) \\ y_{n+1} &= y_n + \tfrac{k_1}{6} + \tfrac{k_2}{3} + \tfrac{k_3}{3} + \tfrac{k_4}{6}. \end{aligned}$$

a method having local error $O(h^5)$.

The Runge Kutta Fehlberg method, described in the subsection 4.4 below is a nested pair of methods of orders 4 and 5 that permit both advancing the solution from y_n to y_{n+1} and choosing the step size $h_n = t_{n+1} - t_n$ as described in the conceptual time stepping scheme at (4.2.1). The 5th order method is used to advance the solution. A comparison of the result of the 4th and 5th order methods provides a local error estimate that is used to select h_n.

4.3 Global vs Local Error

In the preceding subsection we have just considered the local truncation error for Forward Euler and Modified Euler methods. This is a local error in the sense that it is the error

when making a single step. In this section, we ask how this affects the *global* error after we make a large number of steps. For example, we are interested in $y(t_n) - y_n$ for some finite $t_n > t_0$.

If we consider for example the error when using Modified Euler, then we see that we have an error term $O(h^3)$ at each step. But these errors will accumulate from step to step so that the global error in the computed solution at the last step will be larger (possibly by a significant amount) than the local error. In the worst case all the local errors accumulate. If we solve the differential equation to some time $t_n > t_0$, and if the step size h is constant for all time steps, then

$$\# \ steps = \frac{t_n - t_0}{h} = O(\frac{1}{h}) \ .$$

Therefore we have

$$\begin{aligned} \text{Global Error} &\leq \text{Local Error} \cdot \# \text{ steps} \\ &= O(h^3) \cdot O(\tfrac{1}{h}) \\ &= O(h^2). \end{aligned}$$

In general, $t_n - t_0 = \sum_{k=1}^{n} h_k$ with an average step-size of $(t_n - t_0)/n$ with the global error being approximately equal to the sum of all the local errors.

One can see that if we have a method with local error $O(h^{p+1})$ then the global error is $O(h^p)$. In general, when we refer to the order of a method then we refer to the global error. Thus we say that Modified Euler is a second order method while Forward Euler is a first order method.

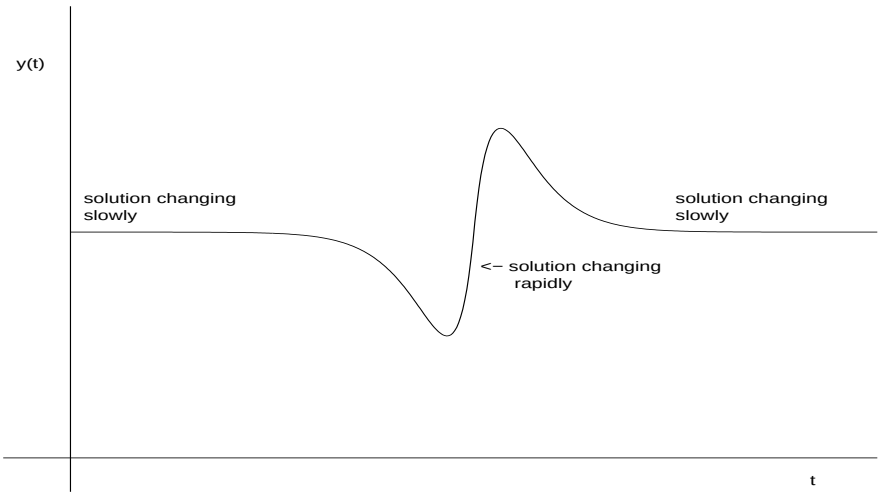
4.4 Practical Issues

The goal of ode solvers is that for a specific tolerance we determine a set of points t_n and y_n such that the local error at the n^{th} step is less than this tolerance. In previous sections we have focussed on the approximation step, that is, the step to determine the y_n. For any particular problem, this part requires only knowledge of $f(t, y(t))$. For example, one has Modified Euler which can be given as a Runge-Kutta method:

$$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_n + h, y_n + k_1) \\ y_{n+1} &= y_n + \tfrac{k_1}{2} + \tfrac{k_2}{2} \ . \end{aligned}$$

Software Libraries can be developed for the various methods with the user only required to supply a function which evaluates $f(t, y)$ for any (t, y). This is the approach used by an ODE suite, for example. The cost and efficiency of any method can be measured by the number of function evaluations required – that is, the number of times the user-supplied function $f(t, y)$ is called to advance the solution to some specified time.

Note that, up to this stage, we have discussed using constant time steps for computing our solution. In practice, this is a bad idea since the size of a time step should depend on the shape of the particular function.



In order to get good accuracy with a constant time step, we would need to choose a time step small enough so that we maintain the desired accuracy at the places where the function changes most rapidly. This results in wasted work since a much larger time step could be used where the function changes slowly.

To take advantage of the changing shape of a function, we will need to detect when to increase or decrease the timestep in response to how rapidly $y(t)$ is changing. Recall that we know the local truncation error for a given method. For example, the local truncation error for the Forward Euler method is $O(h^2)$, since

$$y(t_{n+1}) = y_n + hf(t_n, y_n) + O(h^2) \ .$$

as long as we assume that $y_n = y(t_n)$, that is, we have the exact solution at $t = t_n$.

This means that, for h sufficiently small,

$$\text{Truncation Error } \leq C \cdot h^2$$

for some constant C. If we could estimate C at each step then we could adjust the timestep to ensure that the error was within a user-specified tolerance. A standard technique is to evaluate the solution y_{n+1} with two methods, a higher-order method and a lower-order method. The constant in the order relation for the error can be estimated from the difference (in absolute value) between the two solutions. If the error is less than the user-specified tolerance then the timestep is accepted. Otherwise the timestep size is reduced and this step is repeated. This process is repeated until the tolerance is met. Conversely if the estimated error is much less than the user specified tolerance then this indicates that the timestep for the next step can be increased.

The above procedure is how the Python's `solve_ivp` works when using `method='RK45'` or `method='RK23'`. The `RK23` method combines a second-order and a third-order step with automatic stepsize control, while `RK45` does the same thing except using a fourth- and fifth-order pair of steps. In both cases, the user supplies a tolerance. The estimates are not perfect of course, but in practice the methods work very well.

As an example, the formulas for the RK45 method (known as the Runge-Kutta-

Fehlberg formula) is given by

$$\begin{aligned}
k_1 &= hf(t_n, y_n) \\
k_2 &= hf(t_n + \tfrac{h}{4}, y_n + \tfrac{k_1}{4}) \\
k_3 &= hf(t_n + \tfrac{3h}{8}, y_n + \tfrac{3k_1}{32} + \tfrac{9k_2}{32}) \\
k_4 &= hf(t_n + \tfrac{12h}{13}, y_n + \tfrac{1932k_1}{2197} - \tfrac{7200k_2}{2197} + \tfrac{7296k_3}{2197}) \\
k_5 &= hf(t_n + h, y_n + \tfrac{439k_1}{216} - 8k_2 + \tfrac{3680k_3}{513} - \tfrac{845k_4}{4104}) \\
k_6 &= hf(t_n + \tfrac{h}{2}, y_n - \tfrac{8k_1}{27} + 2k_2 - \tfrac{3544k_3}{2565} + \tfrac{1859k_4}{4104} - \tfrac{11k_5}{40}) \\
y^*_{n+1} &= y_n + \tfrac{25k_1}{216} + \tfrac{1408k_3}{2565} + \tfrac{2197k_4}{4104} - \tfrac{k_5}{5} \text{ with error } O(h^4) \\
y_{n+1} &= y_n + \tfrac{16k_1}{135} + \tfrac{6656k_3}{12825} + \tfrac{28561k_4}{56430} - \tfrac{9k_5}{50} + \tfrac{2k_6}{55} \text{ with error } O(h^5).
\end{aligned}$$

In this case the error can be estimated by

$$Error = y_{n+1} - y^*_{n+1} = \frac{k_1}{360} - \frac{128k_3}{4275} - \frac{2197k_4}{75240} + \frac{k_5}{50} + \frac{2k_6}{55}.$$

4.4.1 Time Step Control I: Basic Principles

In previous subsections we have mentioned that there are two primary issues for determining a discrete set of values which approximate a solution of a first order initial value problem. The first issue is to find a scheme to go from one (or more) y value to the next. Examples of this have been given in the previous discussions. In this subsection we deal with the second issue, that of determining the set of time steps for our approximation.

Suppose we have two estimates of y_{n+1} available, using two different methods, one of order $p-1$ the other of order p. Such methods use the higher order method to determine the next y value and the lower order method to pick h_n so as to bound our local error by a user set tolerance, tol. To see how this is done, let y^*_{n+1} be the low order estimate, and y_{n+1} be the high order estimate, with

$$\begin{aligned}
y^*_{n+1} &= y(t_{n+1}) + O(h^p) \\
&= y(t_{n+1}) + Ah^p + (\text{ smaller terms }) \qquad (4.4.1)
\end{aligned}$$

$$y_{n+1} = y(t_{n+1}) + O(h^{p+1}) \qquad (4.4.2)$$

where $h = t_{n+1} - t_n$ and A is a constant. Now, equation (4.4.1) says that

$$|y^*_{n+1} - y(t_{n+1})| = \text{ Local Error } \simeq |Ah^p|. \qquad (4.4.3)$$

Subtracting equation (4.4.1) from equation (4.4.2) gives

$$\begin{aligned}
|y^*_{n+1} - y_{n+1}| &= Ah^p + (\text{ smaller terms }) \\
&\simeq Ah^p \qquad (4.4.4)
\end{aligned}$$

and so, from equations (4.4.3-4.4.4) we get

$$\begin{aligned} \text{Local Error } = |y^*_{n+1} - y(t_{n+1})| &\simeq |y^*_{n+1} - y_{n+1}| \\ &= Ah^p. \end{aligned} \tag{4.4.5}$$

Note that we can estimate the constant A in equation (4.4.5) by

$$A = \frac{|y^*_{n+1} - y_{n+1}|}{h^p}. \tag{4.4.6}$$

Suppose that a user specifies an error tolerance of tol for each recursive step. We can then estimate the actual error in our computation of y^*_{n+1} (for this step) from

$$\text{Local Error} \simeq |y^*_{n+1} - y_{n+1}|. \tag{4.4.7}$$

If the error is less then tol, this step is accepted. If the error is greater than tol, then the step is rejected. We repeat the computation using a smaller value of h, typically $h := h/2$.

Assuming the step is successful, then we would like to determine an appropriate stepsize for the next step. Let

$$\begin{aligned} h_{old} &= t_{n+1} - t_n \\ h_{new} &= t_{n+2} - t_{n+1}. \end{aligned} \tag{4.4.8}$$

Assuming that we have successfully computed y_{n+1}, using a stepsize h_{old}, we would then like to choose an h_{new} to be used in computing y_{n+2}. We would like the error in this new step to be about tol. If the error is much less than tol, then we are wasting work, since we are computing a more accurate solution than requested by the user, which takes smaller timesteps, and costs more in terms of $flops$. On the other hand, if the error turns out to be $> tol$, then the new step will be rejected, and we have wasted work.

Now, we will make the (rather large) assumption that the constant A in equation (4.4.5) for the step $y_n \to y_{n+1}$ is the same as for the step $y_{n+1} \to y_{n+2}$. Taking a stepsize h_{new} for the step $y_{n+1} \to y_{n+2}$, would then imply that

$$y^*_{n+2} = y(t_{n+2}) + Ah^p_{new} + (\text{ smaller terms}). \tag{4.4.9}$$

Thus if the error is

$$\text{Local Error} = |y^*_{n+2} - y(t_{n+2})| \tag{4.4.10}$$

then, using equation (4.4.6), we have that

$$\begin{aligned} \text{Local Error} &\simeq Ah^p_{new} \\ &= \left(\frac{h_{new}}{h_{old}}\right)^p |y^*_{n+1} - y_{n+1}|. \end{aligned} \tag{4.4.11}$$

Now, we want this error to be about tol, so that equation (4.4.11) gives

$$\left(\frac{h_{new}}{h_{old}}\right)^p |y^*_{n+1} - y_{n+1}| \quad = \quad tol \tag{4.4.12}$$

or, rearranging, and solving for h_{new},

$$h_{new} \quad = \quad h_{old}\left(\frac{tol}{|y^*_{n+1} - y_{n+1}|}\right)^{1/p}. \tag{4.4.13}$$

Actually, since we have made several dubious approximations, we are usually a bit more conservative. As such we use something like

$$h_{new} \quad = \quad h_{old}\left(\frac{0.8\ tol}{|y^*_{n+1} - y_{n+1}|}\right)^{1/p} \tag{4.4.14}$$

to increase our margin of safety in our choice of h_{new}.

4.5 Convergence

At this stage we have not discussed the *convergence* of the methods that have been described so far.

We note that there are various types of errors that can creep into our computations. for example, there are often errors in the original data where the initial conditions are not known exactly. In some cases the differential equations can be highly sensitive with slightly perturbed initial conditions producing significantly different solutions. Such equations are said to be poorly conditioned. Being poorly conditioned is a property of the equation and there is little that can be done from an algorithm point of view.

A second type of error is due to round-off error because of the use of inexact arithmetic to solve our problems. We need to worry if these errors can accumulate from step to step in a disastrous fashion.

A third type of error is due to the use of approximations to derivatives. This is called truncation error. Again the issue is to determine how these truncations propagate from step to step.

4.5.1 Truncation Error of the Forward Euler Method

Let us consider the simple case of the Forward Euler method and see how the truncation error accumulates from step to step. We will assume that we are solving

$$y'(t) = f(t, y(t)), \quad y(0) = y_0$$

in the interval $[0, T]$ via the recursion

$$y_{n+1} = y_n + f(t_n, y_n) \cdot h \tag{4.5.1}$$

where $h = \frac{T}{N}$ is some constant interval. We denote by $y(t)$ as the exact solution of the initial value problem so that

$$e_n = y(t_n) - y_n$$

is the error at $t = t_n$.

From the Taylor series for $y(t)$ at $t = t_{n+1}$ expanded about the point $t = t_n$ we have that

$$y(t_{n+1}) = y(t_n) + f(t_n, y(t_n)) \cdot h + \frac{y''(\eta_n)}{2} \cdot h^2 \tag{4.5.2}$$

with $\eta_n \in [t_n, t_{n+1}]$. Note that the exact solution $y(t)$ does not satisfy the difference equation since the difference equation is not exact. Combining (4.5.1) and (4.5.2) gives

$$e_{n+1} = e_n + (f(t_n, y(t_n)) - f(t_n, y_n)) \cdot h + \frac{y''(\eta_n)}{2} \cdot h^2. \tag{4.5.3}$$

Assume that our function f satisfies the following *Lipschitz* condition:

$$\left| \frac{f(t, y(t)) - f(t, y_n)}{y(t) - y_n} \right| \leq K \tag{4.5.4}$$

for some constant K. This is almost the same as saying that the derivative of f as a function of the second variable is bounded. In addition, assume that

$$|y''(t)| \leq M \text{ for all } t \in [0, T] \tag{4.5.5}$$

that is that the exact solution has a bounded second derivative in the interval of interest. Then from (4.5.3), (4.5.4) and (4.5.5) we get

$$e_{n+1} = e_n + \frac{(f(t_n, y(t_n)) - f(t_n, y_n))}{y(t_n) - y_n} \cdot e_n \cdot h + \frac{y''(\eta_n)}{2} \cdot h^2. \tag{4.5.6}$$

Taking absolute values from both sides of (4.5.6) and using (4.5.4) and (4.5.5) gives

$$|e_{n+1}| \leq |e_n| + h \cdot |e_n| \cdot K + \frac{M}{2} \cdot h^2. \tag{4.5.7}$$

Now if the initial error is zero, that is, $e_0 = 0$, then we have

$$\begin{aligned}
|e_1| &\leq \tfrac{M}{2} h^2 \\
|e_2| &\leq |e_1| + h \cdot K \cdot |e_1| + \tfrac{M}{2} h^2 \\
&\leq (1 + h \cdot K) \tfrac{M}{2} h^2 + \tfrac{M}{2} h^2, \\
|e_3| &\leq |e_2| + h \cdot K \cdot |e_2| + \tfrac{M}{2} h^2 \\
&\leq (1 + h \cdot K)^2 \tfrac{M}{2} h^2 + (1 + h \cdot K) \tfrac{M}{2} h^2 + \tfrac{M}{2} h^2,
\end{aligned}$$

and in general

$$\begin{aligned} |e_n| &\leq |e_{n-1}| + h \cdot K \cdot |e_{n-1}| + \frac{M}{2}h^2 && (4.5.8)\\ &\leq \frac{M}{2}h^2 \cdot \sum_{j=0}^{n-1}(1+hK)^j && (4.5.9)\\ &= \frac{M}{2}h^2 \cdot \frac{(1+hK)^n - 1}{(1+hK)-1} && (4.5.10)\\ &\leq \frac{M}{2}h^2 \cdot \frac{(1+hK)^n}{hK} && (4.5.11)\\ &= \frac{Mh}{2K} \cdot (1+hK)^n. && (4.5.12) \end{aligned}$$

Now

$$(1+hK)^n \leq (\exp(hK))^n = \exp(nhK) \tag{4.5.13}$$

so equations (4.5.12) and (4.5.13) give

$$|e_n| \leq \frac{Mh}{2K} \cdot \exp(nhK). \tag{4.5.14}$$

If we now let $n \to \infty$ and $h \to 0$ so that $n = \frac{T}{h}$, then equation (4.5.14) gives

$$|e_n| \leq \frac{Mh}{2K} \cdot \exp(TK) = O(h).$$

Therefore Forward Euler converges to the exact solution as $h \to 0$.

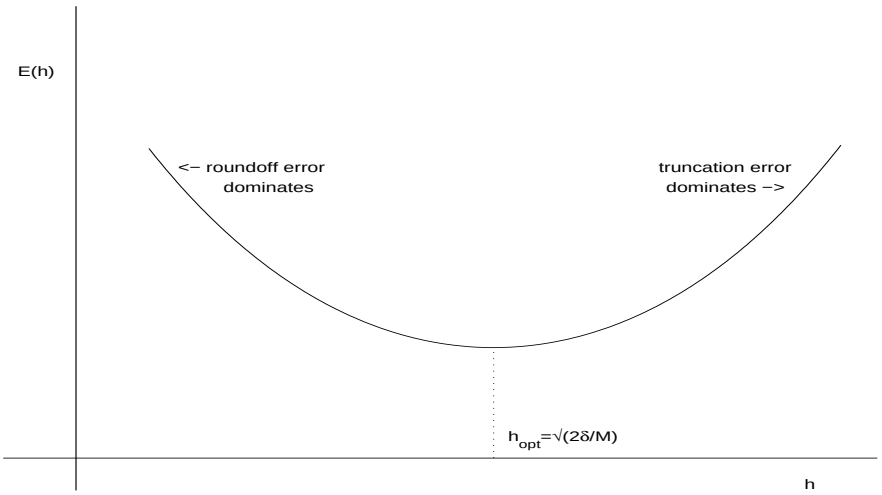
The previous analysis assumes exact arithmetic. However, it is also possible to consider the effects of introducing some round-off error at each step. If we assume that all round-off errors are at most δ, then equation (4.5.7) becomes

$$|e_{n+1}| \leq |e_n| + h|e_n|K + \frac{M}{2}h^2 + \delta. \tag{4.5.15}$$

Note that now $|e_n|$ measures both the effects of truncation error and round-off error. Again, using a similar approach as done before we obtain

$$|e_n| \leq \frac{1}{K}\left(\frac{M}{2}h + \frac{\delta}{h}\right)\exp(TK) \tag{4.5.16}$$

where we assume that $e_0 = 0$ as before. If we set $E(h) = \frac{M}{2}h + \frac{\delta}{h}$, then we can graph this as



We see that initially, as the step size tends to 0 the error decreases. However, eventually the error increases as h decreases due to round-off error. Normally, $\delta \leq \max \delta_n$ where $\delta_n \approx |y_n\ \epsilon|$ where ϵ is machine epsilon. We are usually operating in the region where $E(h)$ is still decreasing as h tends to 0.

4.6 Stability

In the previous subsection we considered the problem of errors that are introduced at each stage due to the use of approximate arithmetic and approximate formulas. In this section we consider the problem caused by introducing a small error in the initial conditions. Clearly a method will have suspect results if a small initial error results in significantly different answers.

Suppose that we solve an initial value problem specified by

$$y'(t) = f(t, y(t)) \text{ with } y(0) = y_0 + \epsilon_0,$$

that is, an initial value problem with a slightly perturbed initial condition. Such a perturbation may be due to round-off error, discretization error or just data errors in the initial conditions.

Let us assume that when we compute $y(t_1), y(t_2), \ldots, y(T)$ then we introduce no other errors at each step. We can ask the following question: What is the effect of this initial perturbation on the approximate solution at the end point T?

If the effect of this initial error becomes unbounded as the number of points $0 = t_0, t_1, \ldots, t_n = T$ is large (that is, as $n \to \infty$) then our algorithm is said to be *unstable*. Otherwise the algorithm is said to be *stable*.

One way to test for the stability of an algorithm is by introducing an error ϵ_0 and seeing if it becomes amplified exponentially as the number of t steps becomes large. For example, we can test a method by seeing how it behaves on a simple test equation, such as

$$y'(t) = -\lambda y(t), \text{ with } y(0) = y_0 \tag{4.6.1}$$

and where λ is a positive constant. This of course is a very simple equation which has the exact solution

$$y(t) = y_0 e^{-\lambda t}. \tag{4.6.2}$$

Note that the exact solution tends to 0 and $t \to \infty$ and that the exact solution is positive if y_0 is positive.

Let us look to see how the Forward Euler method with constant increments behaves with the initial value problem (4.6.2). If h is the constant interval size then the recursion is given by

$$\begin{aligned} y_{n+1} &= y_n - \lambda h y_n && (4.6.3)\\ &= (1 - \lambda h) y_n && (4.6.4)\\ &= (1 - \lambda h)^2 y_{n-1} && (4.6.5)\\ &= \ldots && (4.6.6)\\ &= (1 - \lambda h)^{n+1} y_0 && (4.6.7) \end{aligned}$$

The exact solution of equation (4.6.1) decays to 0 as $t \to \infty$ ($n \to \infty$). We obtain this same behaviour only if

$$|1 - \lambda h| \leq 1, \text{ that is } h < \frac{2}{\lambda},$$

since in this case

$$(1 - \lambda h)^n \to 0 \text{ as } n \to \infty.$$

On the other hand, if $h > \frac{2}{\lambda}$ then

$$(1 - \lambda h)^n \to \infty \text{ as } n \to \infty.$$

and so in this case the algorithm will "blow up". The same effect will occur with any round-off error. For example, let $y^{(p)}(t)$ be the solution to (4.6.2) after a slight perturbation is introduced at $t = 0$, that is,

$$y^{(p)}(0) = y_0 + \epsilon_0.$$

In this case, our recursion is again

$$\begin{aligned} y_{n+1}^{(p)} &= y_n^{(p)} - \lambda h y_n^{(p)} && (4.6.8)\\ &= (1 - \lambda h) y_n^{(p)} && (4.6.9)\\ &= (1 - \lambda h)^2 y_{n-1}^{(p)} && (4.6.10)\\ &= \ldots && (4.6.11)\\ &= (1 - \lambda h)^{n+1} (y_0 + \epsilon_0) && (4.6.12) \end{aligned}$$

Thus if $e_{n+1} = y_{n+1}^{(p)} - y_{n+1}$, then

$$e_{n+1} = (1 - \lambda h) e_n. \tag{4.6.13}$$

Thus again the effect of any initial error will be damped if $h < \frac{2}{|\lambda|}$ and unbounded otherwise.

In this case we say that Forward Euler is *conditionally stable*, that is, the method is stable only if h is sufficiently small.

One can repeat the above analysis for the initial value problem

$$y'(t) = f(t, y(t)).$$

Then when $\frac{\partial}{\partial y} f(t, y) < 0$, the stability condition for Forward Euler is

$$h < \frac{2}{|\frac{\partial}{\partial y} f(t, y(t))|}.$$

With this we can see that the maximum stable timestep will depend on the solution and will change as $y(t)$ evolves.

Suppose now that we look at a backward difference approximation to the derivative. Thus we look at

$$\frac{y_{n+1} - y_n}{h} = f(t_{n+1}, y_{n+1}) \tag{4.6.14}$$

a first order approximation (locally second order). Rewriting (4.6.14) gives

$$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1}). \tag{4.6.15}$$

This gives an implicit scheme referred to as the Backward Euler method (or sometimes implicit Euler method). If we now apply Backward Euler to the test equation

$$y'(t) = -\lambda y(t), \quad \lambda > 0$$

then we obtain

$$y_{n+1} = \frac{y_n}{1 + \lambda h}. \tag{4.6.16}$$

Doing the same perturbation analysis as before, we get the equation for error propagation as

$$e_{n+1} = \frac{e_n}{1 + \lambda h} \tag{4.6.17}$$

where $e_n = y_n^p - y_n$ and where y^p is the solution computed by perturbing y_0.

Equation (4.6.17) gives

$$e_{n+1} = \frac{e_0}{(1 + \lambda h)^{n+1}} \tag{4.6.18}$$

so for $\lambda > 0$ any error introduced at $t = 0$ will be damped out, that is, become exponentially small after a large number of steps. This happens for any timestep and so this algorithm is unconditionally stable. We remark that in general implicit methods tend to have better stability properties than explicit methods.

Example 4.6.1 *Consider the Trapezoidal or Crank-Nicolson Rule*

$$y_{n+1} = y_n + \frac{h}{2}[f(t_n, y_n) + f(t_{n+1}, y_{n+1})]. \tag{4.6.19}$$

which is a second order implicit method. Applying this to the test equation

$$y'(t) = -\lambda y(t), \quad \lambda > 0$$

gives

$$y_{n+1} = y_n - \frac{\lambda h}{2}[y_{n+1} + y_n].$$

following the usual steps we obtain

$$\begin{aligned} e_{n+1} &= \frac{1-\frac{1}{2}\lambda h}{1+\frac{1}{2}\lambda h} e_n \\ &\cdots \\ &= \left(\frac{1-\frac{1}{2}\lambda h}{1+\frac{1}{2}\lambda h}\right)^{n+1} e_0. \end{aligned}$$

This implies that Crank-Nicolson is unconditionally stable. □

Example 4.6.2 *Consider now Modified Euler:*

$$\begin{aligned} y^*_{n+1} &= y_n + hf(t_n, y_n) \\ y_{n+1} &= y_n + \tfrac{h}{2}(f(t_n, y_n) + f(t_{n+1}, y^*_{n+1})) \end{aligned}$$

with the test equation

$$y'(t) = -\lambda y(t), \quad \lambda > 0.$$

We obtain

$$y_{n+1} = y_n \left[1 - \lambda h + \frac{(\lambda h)^2}{2}\right].$$

Again, following the usual steps we obtain

$$\begin{aligned} e_{n+1} &= \left[1 - \lambda h + \tfrac{(\lambda h)^2}{2}\right] e_n \\ &\cdots \\ &= \left[1 - \lambda h + \tfrac{(\lambda h)^2}{2}\right]^{n+1} e_0 \end{aligned}$$

Therefore, for stability, we require that

$$\left|1 - \lambda h + \frac{(\lambda h)^2}{2}\right| < 1$$

which, after some algebra, can be shown to be satisfied if

$$\lambda h < 2.$$

Hence we have the same stability condition as for Forward Euler. □

Finally, we remark that we can construct a higher-order unconditionally stable method by extending the backward difference ideas seen previously.

Example 4.6.3 *We can use three points to construct a second order approximation to a derivative via*

$$\frac{\frac{3}{2}y_{n+1} - 2y_n + \frac{1}{2}y_{n-1}}{h} = y'(t_{n+1}) + O(h^2). \tag{4.6.20}$$

This gives a second order backward approximation and allows us to approximate the solution to the equation $y'(t) = f(t, y(t))$ by

$$y_{n+1} = \frac{2}{3}hf(t_{n+1}, y_{n+1}) + \frac{4}{3}y_n - \frac{1}{3}y_{n-1}. \tag{4.6.21}$$

Note that this backward difference formula uses 3 time levels and so is a multistep method. This means that we must use a single step method to start off.

Applying (4.6.21) to $y'(t) = -\lambda y(t)$ gives the equation

$$\left(\frac{3}{2} + \lambda h\right) y_{n+1} - 2y_n + \frac{1}{2}y_{n-1} = 0.$$

The typical perturbation analysis then results in

$$\left(\frac{3}{2} + \lambda h\right) e_{n+1} - 2e_n + \frac{1}{2}e_{n-1} = 0.$$

This is a second order recurrence equation with constant coefficients and is solved by setting $e_n = \mu^n$ and solving for μ. In this case μ would have to satisfy

$$\left(\frac{3}{2} + \lambda h\right) \mu^2 - 2\mu + \frac{1}{2} = 0$$

which gives

$$\mu = \frac{2 \pm \sqrt{4 - 2(\frac{3}{2} + \lambda h)}}{3 + 2\lambda h}.$$

Since $e_n = \mu^n$ we must have $|\mu| < 1$ for stability (note that since μ is complex then the absolute value signs denote the magnitude of a complex number). After some algebra we can show that

$$|\mu| < 1 \text{ for all } \ h > 0$$

which implies that our multistep method is unconditionally stable. □

Technical Note: Suppose we have the recurrence

$$a\epsilon_{n+1} + b\epsilon_n + c\epsilon_{n-1} = 0.$$

Substitution of $\epsilon_n = \mu^n$ into the equation gives

$$a\mu^2 + b\mu + c = 0.$$

If $b^2 - 4ac = 0$ then this quadratic equation has only one root $\mu = \mu^*$. In this case the solution to the recurrence is given by a linear combination of $(\mu^*)^n$ and $n(\mu^*)^n$.

4.7 Conversion of High Order ODEs to First Order Systems

Most ODE software is based on algorithms for solving a system of first order ODEs. Let $\vec{y} = [y^{(1)}, y^{(2)}, ..., y^{(m)}]^t$. Let $\vec{f} = [f^{(1)}, f^{(2)}, ..., f^{(m)}]^t$. We can denote a system of first order ODEs by

$$\frac{d\vec{y}}{dt} = \frac{d\vec{f}}{dt} , \tag{4.7.1}$$

or, writing this out in component form

$$\begin{aligned} \frac{dy^{(1)}}{dt} &= f^{(1)}(t, y^{(1)}, y^{(2)}, ..., y^{(m)}) \\ &... \\ \frac{dy^{(m)}}{dt} &= f^{(m)}(t, y^{(1)}, y^{(2)}, ..., y^{(m)}) . \end{aligned} \tag{4.7.2}$$

Note that in this context $y^{(m)}$ refers to the $m'th$ component of the vector $\vec{y}$, and $f^{(m)}$ refers to the $m'th$ component of the vector $\vec{f}$.

Consider the ODE

$$\frac{d^3x}{dt^3} = a(x)\frac{d^2x}{dt^2} + b(x)\frac{dx}{dt} + c(x,t) . \tag{4.7.3}$$

This is a *third order* ODE since the highest derivative appearing in equation (4.7.3) is of third order. In order to transform this into a system which looks like equation (4.7.2), we use the following idea. Let

$$\begin{aligned} w &= \frac{dx}{dt} \\ u &= \frac{dw}{dt} = \frac{d^2x}{dt^2} \end{aligned} \tag{4.7.4}$$

so that equation (4.7.3) becomes

$$\begin{aligned} \frac{dx}{dt} &= w \\ \frac{dw}{dt} &= u \\ \frac{du}{dt} &= a(x)u + b(x)w + c(x,t) . \end{aligned} \tag{4.7.5}$$

We can write this system in the form of equation (4.7.2) in the following way. Set

$$\begin{aligned} y^{(1)} &= x \\ y^{(2)} &= w \\ y^{(3)} &= u . \end{aligned} \tag{4.7.6}$$

Then

$$\begin{aligned}
\frac{dy^{(1)}}{dt} &= y^{(2)} \\
\frac{dy^{(2)}}{dt} &= y^{(3)} \\
\frac{dy^{(3)}}{dt} &= a(y^{(1)})y^{(3)} + b(y^{(1)})y^{(2)} + c(y^{(1)}, t). \qquad (4.7.7)
\end{aligned}$$

Thus, in this case we have that the vector dynamics function $\vec{f}$ is

$$\begin{aligned}
f^{(1)} &= y^{(2)} \\
f^{(2)} &= y^{(3)} \\
f^{(3)} &= a(y^{(1)})y^{(3)} + b(y^{(1)})y^{(2)} + c(y^{(1)}, t) \ . \qquad (4.7.8)
\end{aligned}$$

4.8 Stiff Differential Equations

At this stage we have seen that explicit methods are easy to use and are typically only conditionally stable. On the other hand implicit methods have the disadvantage that one needs to solve a non-linear equation at each step (implying more work per step) but are typically unconditionally stable. Both methods allow for high order approximations.

It is sometimes the case that an explicit Runge-Kutta method (with automatic step-size control) will end up taking very small timesteps, even though the solution is not changing very much. In this case the timestep is limited by stability considerations.

Informally, we say that a problem is *stiff* if the timesteps required for a stable explicit method are much smaller than are required for accurately determining those aspects of the problem which are of interest.

So how does stiffness come about? If for example we are using ODE45 (an explicit Runge-Kutta method with error controlled timestep selector) then the following would occur for a stiff problem. After an initial rapid transient error had died out, then we would expect to be able to increase the stepsize. However, if a large timestep is attempted then the error becomes large due to instability. Since the observed error is large then the timestep will be cut. It will turn out that the timestep history will be very erratic with many repeat timesteps. The computation will grind for long periods, although apparently the solution is not changing. In this case the solution is to make use of a solver specifically designed for stiff equations (typically backward difference methods).

Stiffness often manifests itself when solving systems of differential equations. In this case some variables may have very fast transients while others may have slow transients (this happens, for example, in systems resulting from slow chemical reactions). Stiff solvers look for exactly such properties, that is, they tend to look for low and high transients and handle them efficiently. In such cases the rule of thumb is that when in doubt use a stiff solver.

4.9 Exercises for ODEs

1. True or false: With an unconditionally stable method, one can take arbitrarily large time steps in numerically solving a stable ODE within a given accuracy.

2. Verify for the following system of equations

$$\begin{array}{rcl} Y_1' & = & Y_2 \\ Y_2' & = & -x^2 Y_1 - x Y_2 \end{array} .$$

that $Y_1(x) = y(x)$, where $y(x)$ satisfies the second order equation

$$y''(x) + x\ y'(x) + x^2\ y(x) = 0$$

3. Write the following third order differential equation as a system of three first-order equations.

$$y'''(t) + \sin(t)y''(t) - g(t)y'(t) + g(t)y(t) = f(t)$$

4. State the following problem in first order form. Differentiation is respect to t.

$$\begin{array}{rcl} u''(t) + 3v'(t) + 4u(t) + v(t) & = & t \\ v''(t) - v'(t) + u(t) + v(t) & = & \cos(t) \end{array}$$

5. Apply both the Forward Euler and Modified Euler methods to the IVP

$$\begin{cases} y'(t) = -5y(t) \\ y(0) = 5 \end{cases}$$

Show the computation schemes for both methods and estimate the step size for each case that ensures stable computations.

6. Suppose you are using an ODE solver to compute an approximate solution to the equation $y' = f(t, y)$. At some point t_i you have an approximate solution y_i. Using the solver, you compute estimates y_{i+1}^h and $y_{i+1}^{h/2}$ using steps h and $h/2$, where $h = 0.01$. Note that the second estimate involves applying the ODE solver twice, first to get to $t_{i+\frac{1}{2}}$, and then again to get from $t_{i+\frac{1}{2}}$ to t_{i+1}. The method you are using is a second order method with third order local truncation error. Suppose $y_{i+1}^h = 3.269472\cdots$ and $y_{i+1}^{h/2} = 3.269374\cdots$. That is, $\|y_{i+1}^h - y_{i+1}^{h/2}\| \approx 10^{-4}$.

Derive an *estimate* for the local truncation error at the point t_{i+1}. Show carefully how you arrived at your estimate.

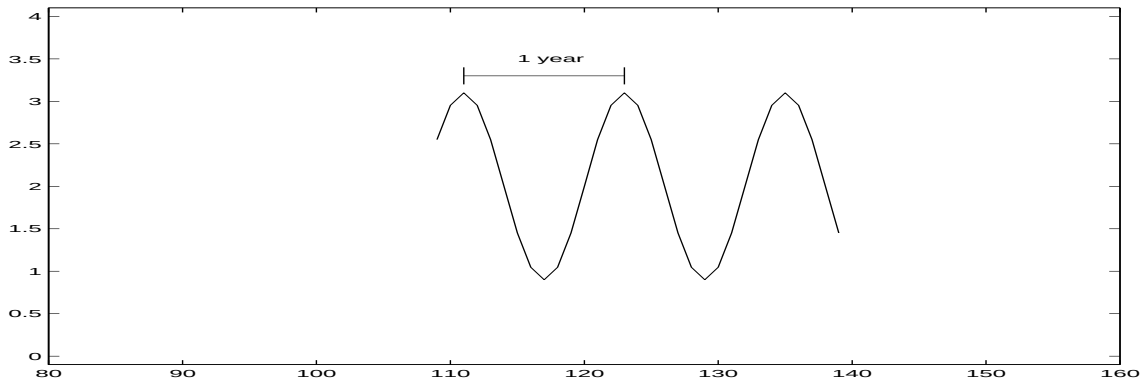
Chapter 5

Discrete Fourier Transforms

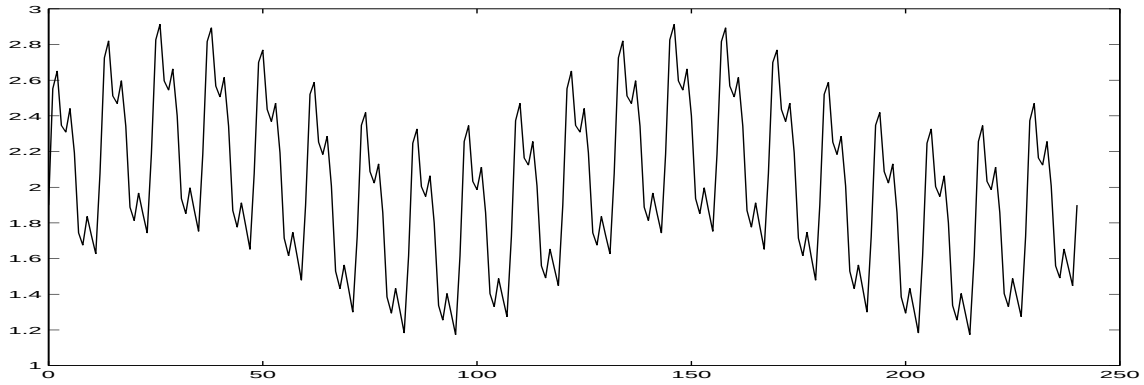
In this chapter we will introduce a tool, the Discrete Fourier Transform, which transforms a finite set of data in one or more dimensions into a second finite set of data. The transformation is reversible and gives an alternate view of information which may be more useful for certain applications. In this case the primary motivation comes from compression of sound and images, and processing of signals (which are usually sound or images).

5.1 Introduction to Fourier Analysis

Suppose one has a table of values which change with equally spaced time intervals, say $f_0, \ldots, f_{N-1}$, and one wishes to analyze the data for trends. As an example, one could have data from the last 20 years for the monthly spot price of orange juice (so 240 equally spaced pieces of information). In this example, one could expect some price fluctuations that are of a cyclic nature, for example based on seasonal supply and demand, a low price in the fall and a higher price in the spring. This would imply that the prices vary according to



which represents the graph of the function $f(t) = a + b\sin(\frac{2\pi}{12}t)$ for some constants a, b. However, the actual data would most likely be of the form



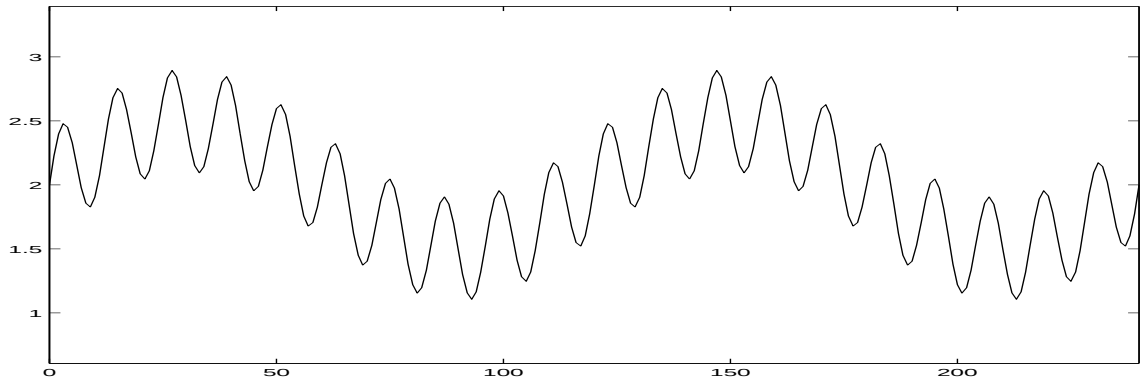
due to such factors as imports from the southern hemisphere, weather fluctuations caused by sunspots or El Nino phenomena, and so on. These factors might also occur on a regular predictable pattern but with alternate cycle periods. As such then we need to see how much the data cycles once every 240 months, or twice every 240 months and so on. If $f(t)$ is the function of orange prices then we are interested in representing this function in the form

$$f(t) = a_0 + a_1 \cos(q \cdot t) + b_1 \sin(q \cdot t) + a_2 \cos(2q \cdot t) + b_2 \sin(2q \cdot t) + \cdots \qquad (5.1.1)$$

where $q = \frac{2\pi}{T}$ with $T = 240$, the time interval of the data. If something such as sunspot activity was the only additional factor beyond seasonal fluctuations that entered into our equation, and if these occurred at say regular 10 year intervals, then our function might be of the form

$$f(t) = a_0 + b_2 \sin(2q \cdot t) + b_{20} \sin(20q \cdot t)$$

which looks like



Fourier analysis for discrete or continuous data involves the conversion of time or spatial information into frequency information via function representations of the form (5.1.1).

Some issues that will be discussed in the following sections include:

a) How to determine coefficients a_k and b_k for data represented in functional form $f(t)$.

b) What to do (and how to do it) in the case where the input consists of discrete data, $f_0, \ldots, f_{N-1}$, rather than continuous data.

c) How to do the discrete computation fast.

d) Applications of Fourier analysis.

5.2 Fourier Series

Let us first consider the case of continuous data, where we are given a function $f(t)$ and a period T. We assume that our function is periodic with period T, that is, it has the property that

$$f(t \pm T) = f(t) \tag{5.2.1}$$

Note that $g(t) = \cos(\frac{2\pi kt}{T})$ and $h(t) = \sin(\frac{2\pi kt}{T})$ have such a property for every integer k. For example,

$$h(t+T) = \sin\left(\frac{2\pi k(t+T)}{T}\right) = \sin\left(\frac{2\pi kt}{T} + 2\pi\right) = h(t).$$

We are interested in representing $f(t)$ in terms of a sum of trig functions

$$f(t) = a_0 + \sum_{k=1}^{\infty} a_k \cos\left(\frac{2\pi kt}{T}\right) + \sum_{k=1}^{\infty} b_k \sin\left(\frac{2\pi kt}{T}\right) \tag{5.2.2}$$

Each a_k, b_k represents the amount of *information* in a harmonic of period $\frac{T}{k}$ or frequency $\frac{k}{T}$.

For simplicity let us assume that we have a function defined on $t \in [0, 2\pi]$, with period $T = 2\pi$, so that we are looking for a representation

$$f(t) = a_0 + \sum_{k=1}^{\infty} a_k \cos(kt) + \sum_{k=1}^{\infty} b_k \sin(kt). \tag{5.2.3}$$

Then we can find formulas for the a_k and b_k in terms of the input function. Indeed, the functions $(1, \cos kt, \sin kt)$ have the property that

$$\begin{aligned}
\int_0^{2\pi} \cos kt \sin k't \; dt &= 0 \; ; \text{ for all } k, k' \\
\int_0^{2\pi} \cos kt \cos k't \; dt &= 0 \; ; \; k \neq k' \\
\int_0^{2\pi} \sin kt \sin k't \; dt &= 0 \; ; \; k \neq k' \\
\int_0^{2\pi} \sin kt \; dt &= 0 \\
\int_0^{2\pi} \cos kt \; dt &= 0.
\end{aligned} \tag{5.2.4}$$

(this is the same as saying that the functions $(1, \cos kt, \sin kt)$ are *orthogonal* on $[0, 2\pi]$). From equations (5.2.3-5.2.4) we can determine the coefficients a_k, b_k.

$$\begin{aligned} a_0 &= \frac{\int_0^{2\pi} f(t)\ dt}{2\pi} \\ a_k &= \frac{\int_0^{2\pi} f(t)\cos kt\ dt}{\int_0^{2\pi} \cos^2 kt\ dt} \\ b_k &= \frac{\int_0^{2\pi} f(t)\sin kt\ dt}{\int_0^{2\pi} \sin^2 kt\ dt} \end{aligned} \tag{5.2.5}$$

For example, a formula for a_ℓ is determined by multiplying both sides of equation (5.2.3) by $\cos(\ell t)$, integrating both sides from 0 to 2π and then taking the integral inside the summations to be done on a term by term basis. This gives

$$\int_0^{2\pi} f(t)\cos(\ell t)dt = \int_0^{2\pi} a_0\cos(\ell t)dt + \sum_{k=1}^{\infty}\int_0^{2\pi} a_k\cos(kt)\cos(\ell t)dt + \sum_{k=1}^{\infty}\int_0^{2\pi} b_k\sin(kt)\cos(\ell t)dt.$$

so

$$\int_0^{2\pi} f(t)\cos(\ell t)dt = a_\ell\int_0^{2\pi}\cos(\ell t)\cos(\ell t)dt = a_\ell\cdot\pi$$

and hence our formula for a_ℓ.

We can write the *Fourier series* (5.2.3) more compactly if we use Euler's formula:

$$e^{i\cdot\theta} = \cos(\theta) + i\sin(\theta) \tag{5.2.6}$$

where $i = \sqrt{-1}$, something which will become particularly convenient when we consider the discrete case. Using

$$e^{-i\cdot\theta} = \cos(\theta) - i\sin(\theta)$$

we get that

$$\cos(\theta) = \frac{e^{i\cdot\theta} + e^{-i\cdot\theta}}{2} \text{ and } \sin(\theta) = \frac{e^{i\cdot\theta} - e^{-i\cdot\theta}}{2i}.$$

Consequently, we can write

$$f(t) = \sum_{k=-\infty}^{\infty} c_k e^{ikt} \tag{5.2.7}$$

where c_k are in general now complex numbers. The correspondence with the c_k and the a_k, b_k of representation (5.2.2) is given by

$$c_0 = a_0, \quad c_k = \frac{a_k}{2} - i\cdot\frac{b_k}{2} \text{ and } c_{-k} = \frac{a_k}{2} + i\cdot\frac{b_k}{2} \text{ for } k > 0.$$

Note that

$$|c_0| = |a_0|, \quad |c_k| = |c_{-k}| = \frac{1}{2}\sqrt{a_k^2 + b_k^2} \text{ for } k > 0.$$

We can obtain a formula for c_ℓ by noting that

$$\begin{aligned} \int_0^{2\pi} e^{ikt} e^{-i\ell t}\, dt &= 0 \ ; \ k \neq \ell \\ &= 2\pi \ ; \ k = \ell. \end{aligned} \tag{5.2.8}$$

If we multiply both sides of (5.2.7) by $e^{-i\ell t}$ and integrate term by term we obtain the formula

$$c_\ell = \frac{1}{2\pi} \int_0^{2\pi} e^{-i\ell t} f(t)\, dt. \tag{5.2.9}$$

For typical functions $f(t)$, there is a small amount of information in the high frequency harmonics. Therefore, we can approximate any input *signal* $f(t)$ by

$$f(t) \simeq \sum_{k=-M}^{M} c_k e^{i\ kt} \tag{5.2.10}$$

for M not too large. In an electrical signal, the magnitude of the c_j represents the amount of *power* in the frequency k/T. High frequencies often represent noise in a signal. By suppressing high frequency components of a signal or image (filtering), we can often produce a clean image. On the other hand, high frequency components in a digital image can represent edges, so that by boosting high frequency components, edges are enhanced. Image compression algorithms (e.g., JPEG) utilize the fact that little information is carried in high frequency components to reduce the storage required for a digital image.

5.3 Discrete Fourier Transform (DFT)

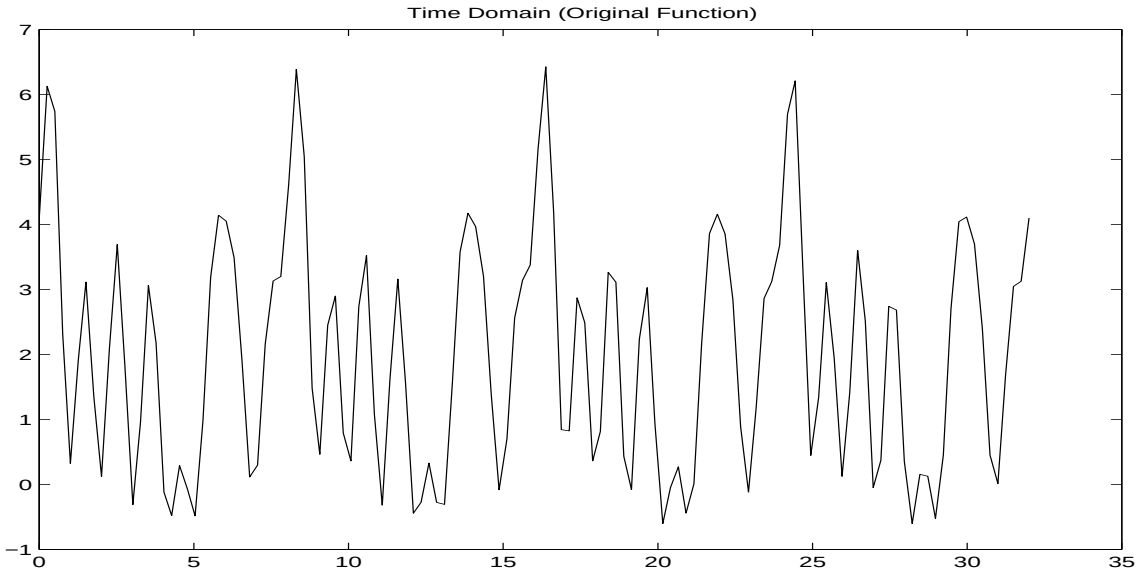
Of course, in practice, we do not have an analytic expression for $f(t)$, the input signal. Usually, we have an input signal sampled with N samples over a period T, with samples being $\Delta t = T/N$ apart. Let us denoted the input signal by $f(n\Delta t) = f_n$, and so the input sampled data is given by the set $f_0, f_1, ..., f_{N-1}$. Note that $f(0) = f(T)$, so that given $f(0)$, $f(T)$ is redundant. Let the discrete sample time be

$$t_n = n\Delta t = \frac{nT}{N} \ ; \ n = 0, 1, ..., N-1 \tag{5.3.1}$$

We can think of our problem as a problem in interpolation. We are given a finite set of points $(t_0, f_0), \ldots, (t_{N-1}, f_{N-1})$ and wish to find constants a_0, a_k, b_k such that

$$a_0 + \sum_{k=0}^{M} a_k \cos\left(\frac{2\pi}{T} kt\right) + \sum_{k=0}^{M} b_k \sin\left(\frac{2\pi}{T} kt\right)$$

(for some M) interpolates these values. For example, the 128 equally spaced samples from 0 to $T = 32$ having the following graph



in fact interpolate the function

$$f(t) = 2.0 + 1.2\cos(4qt) + 0.8\cos(12qt) + 1.1\sin(12qt) + 1.1\cos(16qt) + 0.9\sin(28qt) - \cos(32qt)$$

where $q = \frac{2\pi}{32}$. The problem in this case is how to determine the particular coefficients in front of the sin and cos functions (including of course the coefficients which are 0).

As was the case in the previous section, it is convenient to use exponential rather than trigonometric forms for our expression. Since we have N sample points, we can fit this data exactly if we use N degrees of freedom. Therefore we approximate the input signal $f(t)$ by

$$f(t) \simeq \sum_{k=-N/2+1}^{N/2} c_k e^{\frac{i\,2\pi kt}{T}} \tag{5.3.2}$$

where we will assume from now on that N is even.

For formula (5.3.2) to be useful we need to have a formula for the c_k in terms of the sampled values $f(n\Delta t) = f_n$ where $t_n = n\Delta t = \frac{nT}{N}$. Then

$$f(n\Delta t) = f_n = \sum_{k=-N/2+1}^{N/2} c_k e^{i\,\frac{2\pi nk}{N}}. \tag{5.3.3}$$

For computational purposes (see section on FFT) we wish to write this formula in a more convenient fashion. Recall that we have assumed that $f(t+T) = f(t)$. Thus,

$$\begin{aligned} f_n &= \sum_{k=-N/2+1}^{N/2} c_k e^{i\,\frac{2\pi nk}{N}} \\ &= \sum_{k=0}^{N/2} c_k e^{i\,\frac{2\pi nk}{N}} + \sum_{k=-N/2+1}^{-1} c_k e^{i\,\frac{2\pi nk}{N}}. \end{aligned} \tag{5.3.4}$$

Letting $j = N + k$ in the second term in equation (5.3.4), we get

$$\begin{aligned}\sum_{k=-N/2+1}^{-1} c_k e^{i\frac{2\pi nk}{N}} &= \sum_{j=N/2+1}^{N-1} c_{j-N} e^{i\frac{2\pi n(j-N)}{N}} \\ &= \sum_{j=N/2+1}^{N-1} c_{j-N} e^{i\frac{2\pi nj}{N}} \end{aligned} \tag{5.3.5}$$

since $e^{i\frac{2\pi n(-N)}{N}} = e^{-i2\pi n} = 1$. For convenience, let us define for $j > N/2$

$$c_j = c_{j-N} \; ; \; j = N/2+1, ..., N-1 \tag{5.3.6}$$

that is, we have extended the definition of c_j periodically. By this we mean that $c_{j\pm N} = c_j$. With equation (5.3.6), equation (5.3.5) now becomes

$$\sum_{k=-N/2+1}^{-1} c_k e^{i\frac{2\pi nk}{N}} = \sum_{k=N/2+1}^{N-1} c_k e^{i\frac{2\pi nk}{N}} \tag{5.3.7}$$

so that equations (5.3.4) and (5.3.7) give

$$f_n = \sum_{k=0}^{N-1} c_k e^{i\frac{2\pi nk}{N}} \tag{5.3.8}$$

Or, letting $F_k = c_k$, we can write this as

$$f_n = \sum_{k=0}^{N-1} F_k e^{i\frac{2\pi nk}{N}} \tag{5.3.9}$$

It is convenient to write equation (5.3.8) as

$$f_n = \sum_{j=0}^{N-1} F_j W^{nj} \tag{5.3.10}$$

where $W = e^{\frac{i\,2\pi}{N}}$. A useful property to note is that

$$\sum_{j=0}^{N-1} W^{j(k-l)} = N\delta_{k,l} \tag{5.3.11}$$

where

$$\begin{aligned}\delta_{k,l} &= 0 \; k \neq l \\ &= 1 \; k = l. \end{aligned} \tag{5.3.12}$$

Equation (5.3.11) follows from the simple polynomial factorization

$$x^N - 1 = (x-1)\cdot(x^{N-1} + x^{N-2} + \cdots + x + 1)$$

which yields a formula for the sum of a finite geometric series

$$\sum_{j=0}^{N-1} x^j = \frac{x^N - 1}{x - 1}. \tag{5.3.13}$$

So, for $(k \neq l)$ we have

$$\sum_{j=0}^{N-1} W^{j(k-l)} = \frac{W^{(k-l)N} - 1}{W^{k-l} - 1} = 0$$

since $W^{(k-l)N} = 1$ when $k \neq l$ and $k - l$ is an integer.

If we multiply equation (5.3.10) by W^{-nk} and then sum over n we get

$$\begin{aligned}
\sum_{n=0}^{N-1} f_n W^{-nk} &= \sum_{n=0}^{N-1}\sum_{j=0}^{N-1} F_j W^{nj} W^{-nk} \\
&= \sum_{j=0}^{N-1} F_j \sum_{n=0}^{N-1} W^{n(j-k)} \\
&= \sum_{j=0}^{N-1} F_j \delta_{j,k} N \\
&= F_k N
\end{aligned} \tag{5.3.14}$$

or, rewriting equation (5.3.14) and equation (5.3.10) we obtain the discrete Fourier transform (DFT) pair

$$\begin{aligned}
f_n &= \sum_{j=0}^{N-1} F_j W^{nj} \\
F_k &= \frac{1}{N}\sum_{n=0}^{N-1} f_n W^{-nk}
\end{aligned} \tag{5.3.15}$$

Example 5.3.1 *Let's compute the DFT of the following data*

$$f_n = \cos\left(\frac{2\pi n}{N}\right) \qquad n = 0, 1, ..., N-1.$$

The coefficients of the DFT are given by

$$F_k = \frac{1}{N}\sum_{n=0}^{N-1} \cos\left(\frac{2\pi n}{N}\right) W^{-nk}.$$

Recall that we can write cosine in terms of e (see equation (5.2.6)) and this will enable us to relate the cosine term to $W = e^{i2\pi/N}$. *Write*

$$F_k = \frac{1}{N}\sum_{n=0}^{N-1}\frac{1}{2}\left(e^{i\frac{2\pi n}{N}} + e^{-i\frac{2\pi n}{N}}\right)e^{-i\frac{2\pi nk}{N}} = \frac{1}{2N}\sum_{n=0}^{N-1}e^{i\frac{2\pi n}{N}(1-k)} + \frac{1}{2N}\sum_{n=0}^{N-1}e^{-i\frac{2\pi n}{N}(1+k)}.$$

Now, when $k = 1$ *the expression reduces to*

$$\frac{1}{2N}\sum_{n=0}^{N-1}1 + \frac{1}{2N}\sum_{n=0}^{N-1}e^{-i\frac{4\pi n}{N}}.$$

The first summation term amounts to N, *and the second can be recognized as a geometric sum (equation (5.3.13)) with ratio* $e^{-i\frac{4\pi}{N}}$, *which disappears because* $e^{i2j\pi} = 1$ *for any integer* j:

$$\sum_{n=0}^{N-1}e^{-i\frac{4\pi}{N}n} = \frac{e^{-i4\pi} - 1}{e^{-i\frac{4\pi}{N}} - 1} = 0.$$

Thus,

$$F_1 = \frac{1}{2}.$$

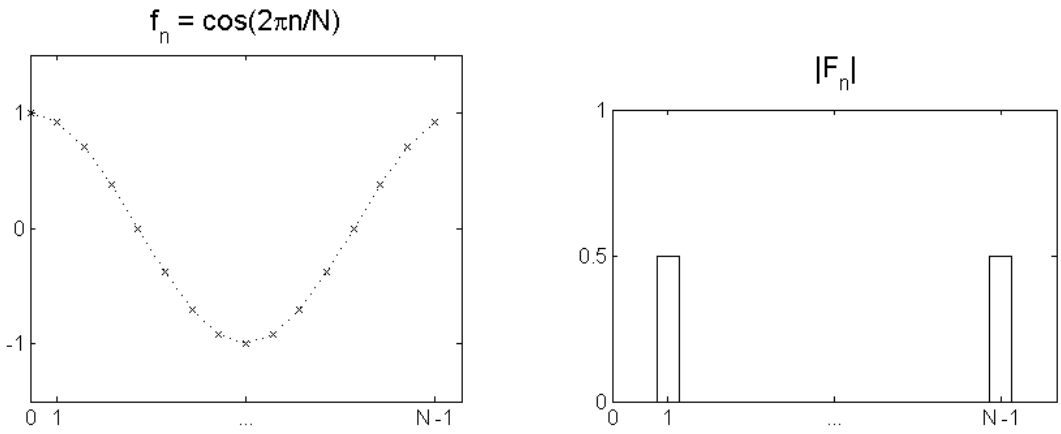
Similarly, when $k = N - 1$, *the first summation term vanishes and the second reduces to* N, *giving*

$$F_{N-1} = \frac{1}{2}.$$

When $k \neq 1$ *and* $k \neq N - 1$, *both sums in the expression for* F_k *can be recognized as geometric sums which vanish in the same way as described above. Therefore,*

$$F_k = \begin{cases} \frac{1}{2} & \text{if } k = 1 \text{ or } k = N-1 \\ 0 & \text{otherwise.} \end{cases}$$

We can present the DFT pair graphically as follows.



The graph on the right shows the modulus of the Fourier coefficients, $|F_n|$. Since the F_n are complex numbers, we use the modulus to get a measure of their size. This graph is called the power spectrum*. (In many books, the power spectrum is defined by the values $|F_n|^2$, but both depict the same qualitative information.) In this case, the size of F_1 reflects the fact that the data has one single pattern.*

□

5.4 Discrete Fourier Transform (DFT) II

In the previous sections we have motivated the definition of the discrete Fourier transform (DFT) by referring to Fourier series of continuous functions. In general and formal terms, the DFT is a reversible transformation of a sequence of N complex numbers. Specifically, it transforms the sequence $\{f_n \in C |\ n = 0, \ldots, N-1\}$, which we will refer to as a data sample, into another sequence of N complex numbers, $\{F_k \in C |\ k = 0, \ldots, N-1\}$, which we refer to as the Fourier coefficients of the sample . Reversible means that if we are given the Fourier coefficients, $\{F_k\}$, there is a simple inverse transformation by which we can reproduce the data sample, $\{f_n\}$. In common applications of the DFT, the $\{f_n\}$ are real-valued observations of some process (e.g. acoustic samples, prices, intensities).

To understand how the DFT can be useful, it is helpful to distinguish between data (e.g. observations) and information (e.g. answers to questions).

- In general, one needs to process data somehow to gain desired information.
- In principle, any information available from the data sample is also available from its Fourier coefficients, since each sequence can be determined from the other.
- In practice, some types of information are more accessible from the Fourier coefficient sequence than from the data sample directly.

Let $W = e^{2\pi i/N}$ be the Nth root of unity introduced in the previous section. Then the Fourier coefficient, F_k, is defined by

$$F_k = \frac{1}{N} \sum_{n=0}^{N-1} f_n \, W^{-nk}. \tag{5.4.1}$$

One might reasonably assume that this definition is meant to apply for $0 \leq k \leq N-1$. However, in fact, it applies for all integer values of k. Mathematically, it is true and useful to think of $\{F_k\}$ as a doubly-infinite sequence. However, only N of the terms are independent; the rest are directly related to them. Let us see why.

The sequence $\{F_k\}$ is doubly infinite, periodic

Assume that $\{F_k |\ k = 0, \ldots, N-1\}$ have been computed by (5.4.1). Hence, we have N elements in our data sample. It is a basic property of the set of integers that any

integer k can be uniquely expressed as $k = mN + p$ for some pair of integers m and p with $0 \le p \le N-1$. This is the meaning of '$k \equiv p \pmod{N}$'. Now each term of (5.4.1) contains W^{-k} and

$$W^{-k} = e^{-2k\pi i/N} = e^{-2mN\pi i/N} e^{-2p\pi i/N} = W^{-p} .$$

Thus

$$F_k \;=\; \frac{1}{N} \sum_{n=0}^{N-1} f_n \, W^{-nk} \;=\; \frac{1}{N} \sum_{n=0}^{N-1} f_n W^{-np} \;=\; F_p \; . \tag{5.4.2}$$

Hence we can regard the definition (5.4.1) of $\{F_k\}$ as defining a doubly infinite and periodic sequence ($-\infty < k < \infty$) that is determined by the subsequence with $0 \le k \le N-1$ (mathematically, any subsequence of length N works).

The property : $F_k = \overline{F_{N-k}}$ for f_n real

We now observe another property of the N-root of unity W, and observe that it implies a special kind of symmetry among the Fourier coefficients when the data sample values, $\{f_n\}$, are all real-valued.

For $1 \le k \le N-1$, note that $N-k$ also lies between 1 and $N-1$. Thus,

$$\begin{aligned} W^{N-k} &= e^{2\pi i(N-k)/N} \\ &= e^{2\pi i} e^{-2\pi ik/N} \\ &= W^{-k} \; . \end{aligned}$$

Consequently, if the data values $\{f_n\}$ are real (so that $f_n = \overline{f_n}$), then

$$F_k = \frac{1}{N} \sum_{n=0}^{N-1} f_n \, W^{-kn} = \frac{1}{N} \sum_{n=0}^{N-1} f_n \, W^{(N-k)n} = \frac{1}{N} \sum_{n=0}^{N-1} \overline{f_n} \, \overline{W^{-(N-k)n}} = \overline{F}_{N-k} \tag{5.4.3}$$

recalling that the overline notation indicates the complex conjugate (see §B.1), and exploiting the property that $\overline{W^r} = W^{-r}$. Hence, there is a conjugate symmetry property in the Fourier coefficients of a real-valued data sample. In other words, when the values in the data sample $\{f_n\}$ are real, we get $F_k = \overline{F_{N-k}}$ for $1 \le k \le N-1$. For example, if $N = 9$, then we know that $F_1 = \overline{F_8}$, $F_2 = \overline{F_7}$, etc.

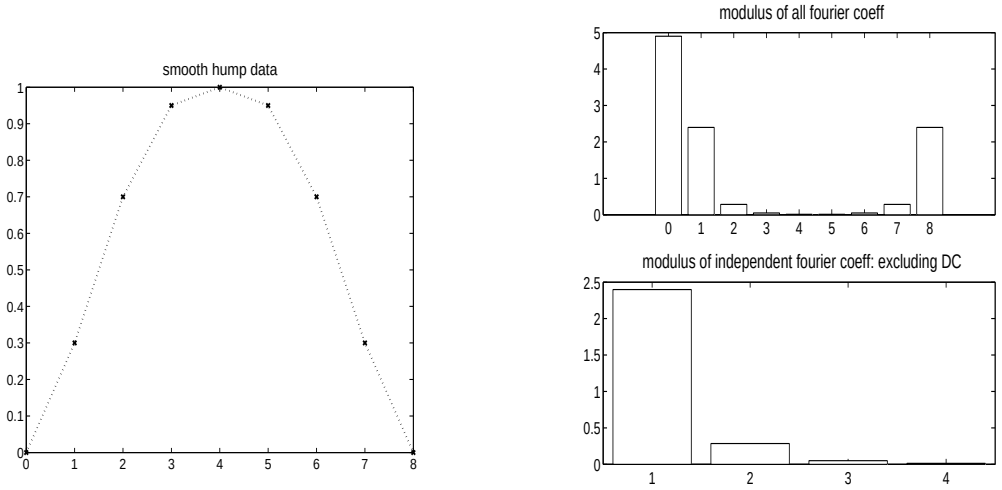
Here are two examples for $N = 9$.

Example 5.4.1 : A Smooth Hump

Consider the following data. For simplicity, we ignore the $\frac{1}{N}$ scaling factor in the Fourier coefficients (it does not affect the analysis here).

n	0	1	2	3	4	5	6	7	8
f_n	0	.3	.7	.95	1.	.95	.7	.3	0
F_n	4.9	-2.25+.82i	-.22+.18i	.03-.04i	-.002+.013i	-.002+.013i	.03+.04i	-.22-.18i	-2.25-.82i

Observe that $F_k = \overline{F_{9-k}}$ for $k = 1, 2, 3, 4$; this is the conjugate symmetry expressed in (5.4.3), and has nothing to do with the fact that in this case f itself happens to be symmetrical. Note that the relation also holds for $k = 0$. This comes from the fact that the Fourier coefficients are periodic so that $F_0 = F_9 = \overline{F_9}$ (since it is real-valued and equals its own conjugate). This coefficient has a special name, the direct current (DC) value. We can present this example graphically.



Fourier coefficients for smooth hump data sample

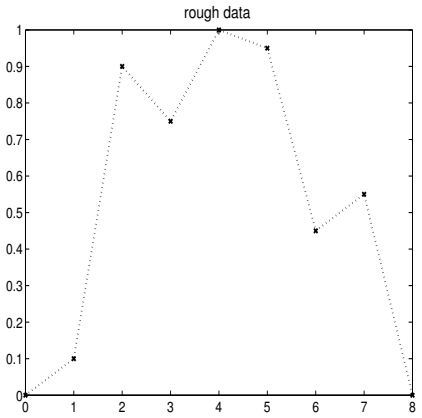
The graphs on the right show the modulus of the Fourier coefficients, $|F_n|$ (the power spectrum).The upper graph shows all 9 coefficients but this is redundant because of the conjugate symmetry. The lower graph shows only the coefficients for $n = 1, 2, 3, 4$. We do not show $|F_0|$ because it contains a particularly simple piece of information about the data sample, that is, F_0 is the sum of the data sample values.

Example 5.4.2 : A Rough Hump

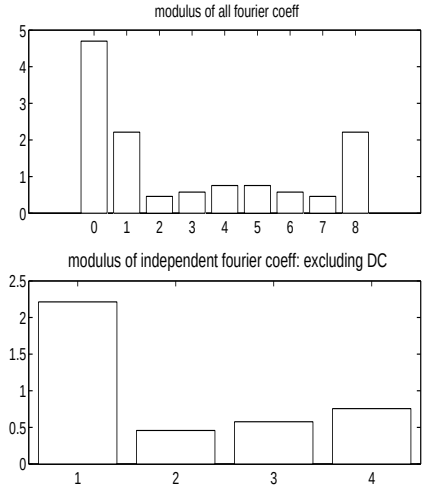
Consider the following data and Fourier coefficients (again, ignoring the scaling factor).

n	0	1	2	3	4	5	6	7	8
f_n	0	.1	.9	.75	1.	.95	.45	.55	0
F_n	4.7	-2.10+.69i	-.45+.07i	-.55-.17i	.755+.020i	.755-.020i	-.55+.17i	-.45-.07i	-2.10-.69i

Graphically, we can present this example as



Data sample for rough hump example



Fourier coefficients for rough hump data sample

Even though these data samples do not have many points, they illustrate how the Fourier coefficients can quantify some aspects of the pattern present in the data sample. They can provide some numerical information about what we can easily see qualitatively from the graphs of data samples, but not so easily see by reading their tables. The extent to which the data has one pattern (e.g. one hump) in both cases is reflected in the size of F_1*, compared to* $|F_k|$ *for* $k = 2, 3, 4$*. In another example shown in Figure 5.1 (on page 91 of these notes), the data sample has a pattern that occurs about four times (i.e. 4 humps). This is reflected in the fact that* F_4 *is the largest coefficient, except for* F_0*.*

The extent to which the data varies smoothly with n *is reflected by way the Fourier coefficients decrease in size as* n *increases. In the smooth hump case, we see that* $|F_{k+1}|/|F_k|$ *is a small fraction for* $k = 2, 3$*. In the case of the rough hump, these ratios are bigger than 1.*

5.5 Inverse Discrete Fourier Transform (IDFT)

We mentioned above that there is a simple transformation of the Fourier coefficients that recovers the data sample values. If the F_n are defined by (5.4.1) , then this transformation is

$$f_n = \sum_{k=0}^{N-1} F_k W^{nk}. \tag{5.5.1}$$

This can be established by looking at a matrix algebra view of (5.4.1), and using the orthogonality of (B.1.12). Let f be the column vector of the data samples, and let F be the corresponding column vector of Fourier coefficients. Then (5.4.1) can be written

$$F = Mf \tag{5.5.2}$$

where M is an $N \times N$ matrix with jth column $= \frac{1}{N}\overline{W(j)}$ of (B.1.12). Now, (B.1.12) states that

$$\overline{M}^t \, M = \frac{1}{N} \, I \tag{5.5.3}$$

where I is the $N \times N$ identity matrix. In other words,

$$M^{-1} = N\,\overline{M}^t.$$

So, from (5.5.2), we can conclude that

$$f = N\,\overline{M}^t\,F.$$

If we write this out componentwise, we find that it is (5.5.1).

5.5.1 Lack of standardization

Unfortunately, there is lack of a standard definition of the exact formula used to define the Fourier transform, and consequently, to carry out the inverse transformation. The differences do not affect the basic role of the transform as an alternative representation of a sequence of real numbers. But they do result in different numerical values for the coefficients of the transform.

One of the differences comes from choosing one of two basic nth roots of unity, $(e^{\pm i\,2\pi/N})$, that can be used to define a transform. We will use U for either. The other source of differences is associated with the choice of a scale factor for the forward transform. We will use the symbol sc for it. The value of sc will determine the scale factor needed for the inverse transform, which must be $1/(Nsc)$.

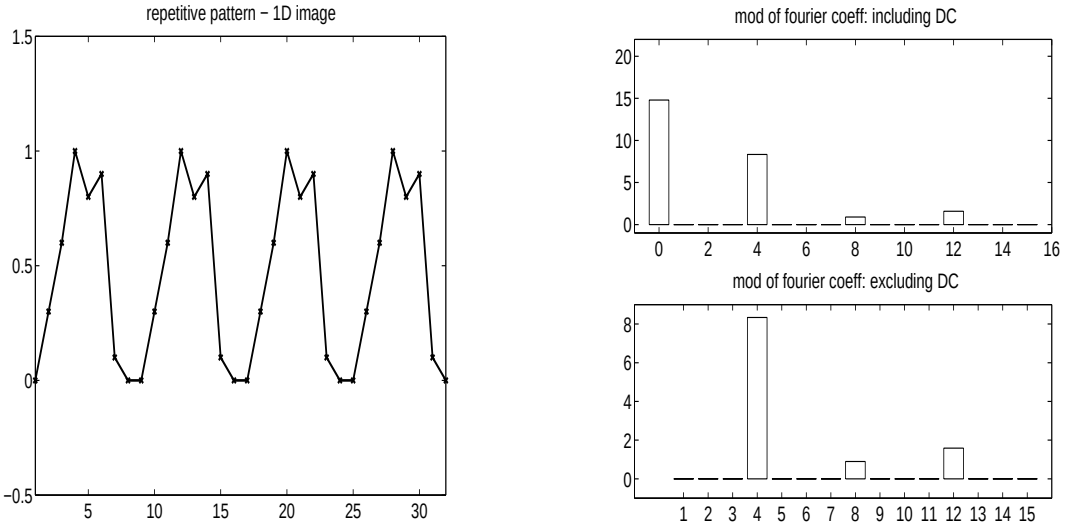
Using these parametric symbols, the common definitions of the transform and inverse transform can be written

$$\begin{aligned} F_n &= sc\sum_{k=0}^{N-1} f_k(\overline{U}^n)^k \qquad (5.5.4)\\ f_n &= \frac{1}{Nsc}\sum_{k=0}^{N-1} F_k(U^n)^k. \end{aligned}$$

In §5.8 of these notes, we use $U = e^{2\pi i/N}$ and $sc = 1/N$. Matlab and Python use $U = e^{2\pi i/N}$ and $sc = 1$. Moreover, Python's array indexing starts at 0, while Matlab's array indexing starts at 1. This can cause some minor confusion between the mathematics and the implementation. To store a data sample $\{f_n|\ n = 0 \ldots N-1\}$ in a Matlab array, x, it stores the value of f_{n-1} in $x(n)$. The Fourier coefficients are also stored in this fashion.

5.6 1-D image compression

On the left of Figure 5.1, we show a plot of the intensities of a 32-number 1D image (actually, it is a small pattern that is repeated 4 times). The pattern can be specified with 7 numbers so there are really only 7 "pieces" of real number information in the image. On the upper-right of Figure 5.1, we show plots of the modulus of the Fourier coefficients $|F_k|$, $k = 0, \ldots, 15$. We can see that only four are non-zero, and one of them



1-D image x Size of Fourier Coefficients

Figure 5.1: 1-D analog of scaled image data and its Fourier transform

is F_0 which is real. So F has the information of 7 real numbers. Because the pattern has a Fourier frequency of 4, $F_1 = F_2 = F_3 = 0$. On the lower-right of Figure 5.1, we show a plot of the non-DC part of the Fourier transform; this contains the actual pattern information and can be plotted on a more readable scale without the DC component, $|F_0|$.

If we create a new pattern, y, which is somewhat like x, by setting

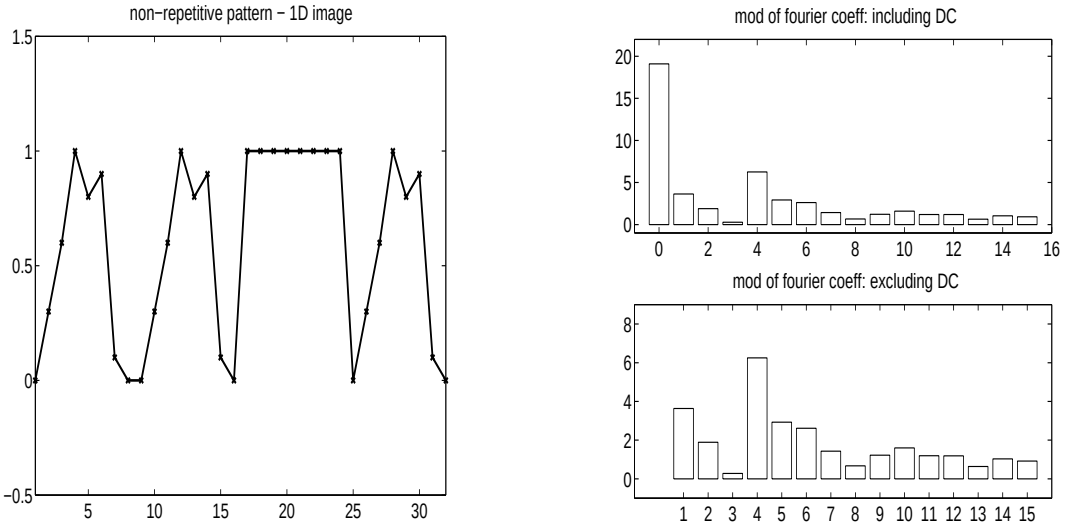
$$y(1:16) = x(1:16); \;\; y(17:24) = 1.0; \;\; y(25:32) = x(25:32), \tag{5.6.1}$$

then y has some of the pattern of x but is, in fact, non-repetitive. Figure 5.2 below shows a plot of the data y and the size of its Fourier coefficients. Note that F_1, F_2, F_3 are not zero for this image. However, note that the coefficient of Fourier frequency 4 is still the second largest, indicating a strong resemblance to a pattern that repeats four times in the total sequence.

Image y has a number of small-valued Fourier coefficients. How important is the information stored in them? If we zero out the four Fourier coefficients with a modulus less than 1, and construct a new image vector from the inverse transform of the reduced set of Fourier coefficients, we get the compressed image shown in Figure 5.3.

5.7 2-D image compression

The data for an image is an array, X, of pixel data. The image specified by X is displayed as a rectangle of very small squares, i.e. pixels. The square at position (i, j) is coloured according to $X(i, j)$. The pixel data may be of several types. The type that we will discuss is double precision floating point numbers in the range $0 \leq x \leq 1$; it is called *scaled image* data. The image can be displayed in a figure window by the command



1-D image y Size of Fourier Coefficients

Figure 5.2: Non-repetitive data and its Fourier transform

`imshow(X)`. You can also specify a colormap using the argument `cmap=gray`. When a gray scale is used to display the array, the square at position (i, j) is set to a shade of gray determined by $X(i, j)$ with $X(i, j) = 0$ being white and $X(i, j) = 1$ being black.

Figure 5.4(a) shows a 256×256 pixel gray scale image of a photo of Montreal. Part (b) of the figure shows a compressed version of the same image, using only 15% of the Fourier coefficients. This compression is achieved by the following method:

1. Take the 2D Fourier transform of the image array (the 2D DFT is discussed later in §5.12).

2. Compress the transformed array by replacing small Fourier coefficients by 0.

3. Reconstruct the compressed image array using the inverse 2D Fourier transform of the compressed data.

The number of nonzero Fourier coefficients dropped from about 65K to about 10K.

One goal of image compression is to facilitate efficient transmission of images. One way to achieve this is to transmit only the non-zero Fourier coefficients of the image, i.e. transmit a compressed transform of the image. However, for a significant image, even the compressed transform is a large file. So image transmission protocols use transforms of sub-blocks of an image. In other words, they break the image into pieces, and apply the compression to each small piece. We will use 16×16 pixel sub-blocks in this discussion.

The 2-D Fourier transform of an array $X(16, 16)$ of scaled image data is described in §5.12 as a complex array $\{F_{k,l}\}$ indexed by $0 \leq k, l \leq 15$. The coefficient $F_{0,0}$ is the DC component of the 2-D Fourier transform.

Figure 5.6 shows the sizes of the Fourier coefficients for a 16×16 sub-block of Figure 5.4 with the DC component set to zero (so the pattern dependent coefficients can be seen more easily). The compressed transform is shown on the right.

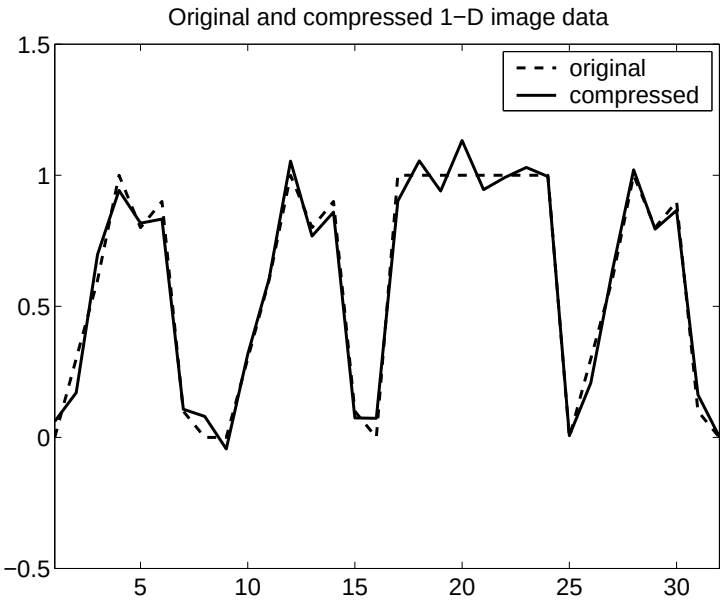


Figure 5.3: Comparison of original (unrepetitive) image and compressed image

5.8 Fast Fourier Transform

We would like to compute

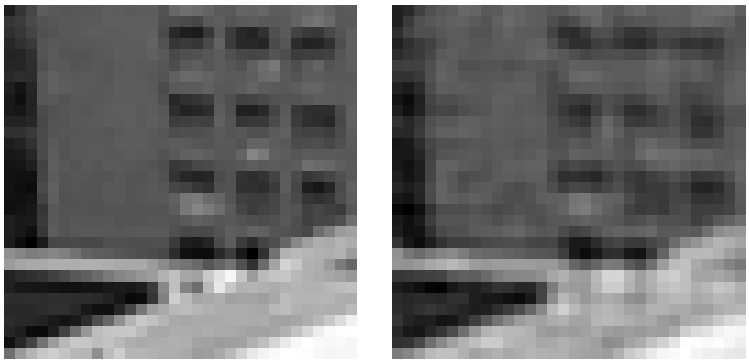
$$F_k \;=\; \frac{1}{N}\sum_{n=0}^{N-1} f_n\, W^{-nk},$$

for $k = 0, \ldots, N-1$, where $W = e^{2\pi i/N}$ is a complex Nth root of unity. Note that we can do this in the standard way at a cost of $O(N^2)$ complex floating point operations.

In this section we show how to compute these N quantities $F_0, \ldots, F_{N-1}$ in $O(N \log_2 N)$ operations using a divide and conquer approach. We assume that $N = 2^m$ for some m.

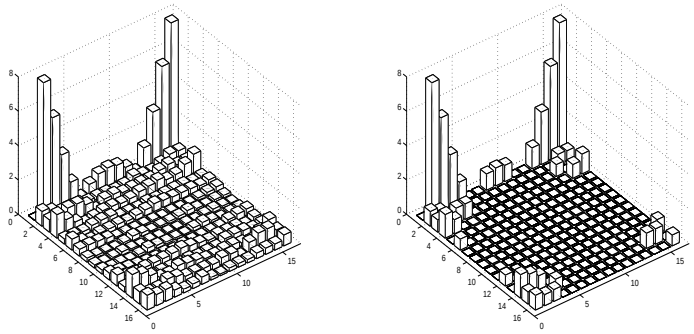
(a) Original (b) Compressed by 85%

Figure 5.4: Images of Montreal



(a) Original (b) Compressed by 85%

Figure 5.5: A 32×32 sub-block of the images in Figure 5.4



(a) Original (b) Compressed by 85%

Figure 5.6: Magnitudes of the Fourier coefficients for a 16×16 sub-block of the image in Figure 5.4. Subfigure (a) shows the original Fourier coefficients, and (b) shows the coefficients after compression.

If this is not the case, then the input signal is padded with zeros (note that this does not effect the sums).

We can split the sums into two parts,

$$F_k = \frac{1}{N} \sum_{n=0}^{N/2-1} f_n W^{-nk} + \frac{1}{N} \sum_{n=N/2}^{N-1} f_n W^{-nk} . \tag{5.8.1}$$

Letting $m = n - N/2$, equation (5.8.1) becomes

$$\begin{aligned} F_k &= \frac{1}{N}\sum_{n=0}^{N/2-1} f_n W^{-nk} + \frac{1}{N}\sum_{m=0}^{N/2-1} f_{m+N/2} W^{-k(m+N/2)} \\ &= \frac{1}{N}\sum_{n=0}^{N/2-1} f_n W^{-nk} + \frac{1}{N}\sum_{n=0}^{N/2-1} f_{n+N/2} W^{-nk} W^{-kN/2} \\ &= \frac{1}{N}\sum_{n=0}^{N/2-1} \left(f_n + W^{-kN/2} f_{n+N/2}\right) W^{-nk}. \end{aligned} \tag{5.8.2}$$

Noting that $W^{-kN/2} = e^{-ik\pi} = (-1)^k$, the even coefficients become

$$F_{2k} = \frac{1}{N}\sum_{n=0}^{N/2-1} \left(f_n + f_{n+N/2}\right) W^{-2nk} \qquad\qquad k = 0, \ldots, N/2-1, \tag{5.8.3}$$

while the odd coefficients become

$$\begin{aligned} F_{2k+1} &= \frac{1}{N}\sum_{n=0}^{N/2-1} \left(f_n - f_{n+N/2}\right) W^{-n(2k+1)} \\ &= \frac{1}{N}\sum_{n=0}^{N/2-1} \left[\left(f_n - f_{n+N/2}\right) W^{-n}\right] W^{-2nk} \qquad k = 0, \ldots, N/2-1\ . \end{aligned} \tag{5.8.4}$$

If we recombine our sample into two half-length sequences, $\{g_n\}$ and $\{h_n\}$, using

$$\begin{aligned} g_n &= \frac{1}{2}(f_n + f_{n+N/2}) \qquad n = 0, \ldots, N/2-1 \\ h_n &= \frac{1}{2}(f_n - f_{n+N/2}) W^{-n} \quad n = 0, \ldots, N/2-1\ , \end{aligned} \tag{5.8.5}$$

then equations (5.8.3-5.8.4) become

$$F_{2k} = \frac{2}{N}\sum_{n=0}^{N/2-1} g_n W^{-2nk} \tag{5.8.6}$$

$$F_{2k+1} = \frac{2}{N}\sum_{n=0}^{N/2-1} h_n W^{-2nk}, \tag{5.8.7}$$

for $k = 0, \ldots, N/2-1$. Also,

$$W^{-2kn} = e^{-\frac{2\pi i}{N}2kn} = \left(e^{-\frac{2\pi i}{N/2}}\right)^{kn}\ ,$$

so that W^2 is the $Wfactor$ for an input signal of length $N/2$ (that is, it is the $(N/2)$-th complex root of unity). Thus, equations (5.8.6-5.8.7) can be regarded as

$$\begin{aligned} F_{even} &= \text{DFT}\ (g_n)\ ;\ N/2\ points \\ F_{odd} &= \text{DFT}\ (h_n)\ ;\ N/2\ points. \end{aligned}$$

Therefore, we have converted the problem of computing a single DFT of N points into the computation of two DFT's of $N/2$ points. We can then reduce each of the $N/2$ length DFT's into two $N/4$ length DFT's, and so on. There will be $\log_2 N$ of these stages. Each stage requires $O(N)$ complex floating point operations, so the complexity of the FFT is $O(N \log_2 N)$.

5.9 FFT Algorithm (Butterfly)

For the DFT pair:

$$\begin{aligned} F_k &= \frac{1}{N} \sum_{n=0}^{N-1} f_n W^{-nk} \\ f_n &= \sum_{k=0}^{N-1} F_k W^{kn}, \end{aligned}$$

the algorithm below takes as input complex data

$$f_0, f_1, ..., f_{N-1}$$

with $N = 2^m$ and computes

$$f'_k = \frac{1}{N} \sum_{n=0}^{N-1} f_n U^{nk}$$

The algorithm overwrites the original array f with f'. However, the output array is in bit-reversed order, so it must be later unscrambled. Note that if

$$U = W^{-1} = e^{-\frac{i\ 2\pi}{N}} \tag{5.9.1}$$

then this is a forward FFT. If

$$U = W = e^{\frac{i\ 2\pi}{N}} \tag{5.9.2}$$

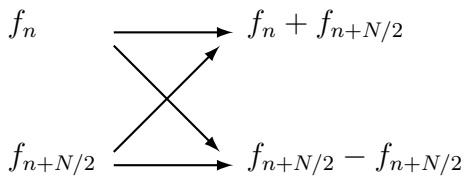
and then f'_k is multiplied by N, then this is a reverse FFT (inverse FFT).

FFT Pseudo-Code

$N = 2^m$
$W = e^{i2\pi/N}$
FOR $k = 1, ..., m$
    For $j = 1, ..., 2^{k-1}$
        $l := (j-1) * N$
        For $n = 0, ..., N/2 - 1$
            $wfactor := (W^{-n})^{2^{k-1}}$

$temp := wfactor * (f_{n+l} - f_{n+l+N/2})$
$f_{n+l} := \frac{1}{2} * (f_{n+l} + f_{n+l+N/2})$
$f_{n+l+N/2} := \frac{1}{2} * temp$
EndFor
EndFor
$N := N/2$
EndFor

Each step of the FFT computes values using f_n and $f_{n+N/2}$ according to (5.8.5) which can be visualized as follows (ignoring the factors):



This diagram resembles the wings of a butterfly, hence the name of the algorithm.

This algorithm requires $O(N \log_2 N)$ (complex) operations, compared to $O(N^2)$ operations for a naive implementation of the DFT.

Given the input $f_0, ..., f_{N-1}$, we want to compute the DFT $F_0, .., F_{N-1}$. The above butterfly algorithm overwrites the original array $f_0, ..., f_{N-1}$ with $f'_0, ..., f'_{N-1}$. Where do we find $F_0, .., F_{N-1}$? Consider the simple case of an input signal of length 8. If we examine the output array, we find (ignoring the $1/N$ factor)

$$\begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \end{bmatrix} \xrightarrow{\text{(Butterfly Operations)}} \begin{bmatrix} F_0 \\ F_4 \\ F_2 \\ F_6 \\ F_1 \\ F_5 \\ F_3 \\ F_7 \end{bmatrix} \tag{5.9.3}$$

so, the output array contains $F_0, ..., F_{N-1}$, but in scrambled order. Let's call this output array $X_0, ..., X_7$. Let's write down the array index for X in binary

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \end{bmatrix} = \text{(binary array index)} \begin{bmatrix} X_{(000)} \\ X_{(001)} \\ X_{(010)} \\ X_{(011)} \\ X_{(100)} \\ X_{(101)} \\ X_{(110)} \\ X_{(111)} \end{bmatrix} \tag{5.9.4}$$

Now, writing down the value of F in each location in the X array, but expressing the index in binary, we see

$$\text{(binary array index)} \quad \begin{bmatrix} X_{(000)} \\ X_{(001)} \\ X_{(010)} \\ X_{(011)} \\ X_{(100)} \\ X_{(101)} \\ X_{(110)} \\ X_{(111)} \end{bmatrix} = \begin{bmatrix} F_{(000)} \\ F_{(100)} \\ F_{(010)} \\ F_{(110)} \\ F_{(001)} \\ F_{(101)} \\ F_{(011)} \\ F_{(111)} \end{bmatrix} = \begin{bmatrix} F_0 \\ F_4 \\ F_2 \\ F_6 \\ F_1 \\ F_5 \\ F_3 \\ F_7 \end{bmatrix} \tag{5.9.5}$$

This ordering is simply a consequence of sorting an N length array into an array of length $N/2$ containing all the even indices, and a second array of length $N/2$ containing all the odd indices (in increasing order). We then repeat this operation for each $N/2$ length array, and so on (which is what the butterfly does). So, in general, we can determine what value of $F_0, ..., F_{N-1}$ is in $X_0, ..., X_{N-1}$, by simply expressing the output array X index as an $\log_2 N$ digit binary number, reversing the bit sequence, and then converting back to decimal. Consequently, the complete FFT algorithm consists of $\log_2 N$ butterfly operations and then a bit reversal reordering.

Example 5.9.1 *Consider the input data,*

$$\mathbf{f} = (1, 2, 3, 4, 1, 2, 3, 4) \quad (N = 8)$$

By applying (5.8.5) we obtain the half-length sequences,

$$\mathbf{g} = (1, 2, 3, 4), \quad \mathbf{h} = (0, 0, 0, 0).$$

Now, we recurse on these two sequences until we reach the base case (a single element), each time overwriting the original array, $\mathbf{f}$*. The elements are unscrambled in the final step. Here is the sequence of operations:*

$$\mathbf{f} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \rightarrow \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} 2 \\ 3 \end{bmatrix} \begin{bmatrix} -1 \\ i \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} \rightarrow \begin{matrix} [\ 2.5\ ] \\ [-0.5\ ] \\ [-0.5+0.5i] \\ [-0.5-0.5i] \\ [\ 0\ ] \\ [\ 0\ ] \\ [\ 0\ ] \\ [\ 0\ ] \end{matrix} \Longrightarrow \begin{bmatrix} 2.5 \\ 0 \\ -0.5+0.5i \\ 0 \\ -0.5 \\ 0 \\ -0.5-0.5i \\ 0 \end{bmatrix} = \mathbf{F}$$

There are five independent Fourier coefficients ($n = 0, 1, 2, 3, 4$). In terms of the magnitude of the coefficients, the largest, F_0, is simply the average of the data values. The next largest is F_2 (with $|F_2| = \frac{1}{\sqrt{2}}$) which reflects the fact that the data contains a repeating pattern (each unique data value is observed twice*).*

5.10 Aliasing

Let us go back to our original representation of a periodic signal

$$f(t) = \sum_{k=-\infty}^{\infty} c_k e^{i \frac{2\pi k t}{T}} \tag{5.10.1}$$

where T is the period, and equation (5.10.1) is exact. Suppose $t_n = \frac{Tn}{N}$. Then (5.10.1) becomes

$$f(t_n) = f_n = \sum_{k=-\infty}^{\infty} c_k e^{i \frac{2\pi k n}{N}}. \tag{5.10.2}$$

A plot of $|c_k|^2$ would give the *exact* power spectrum. Now, the coefficients of the DFT, F_k are defined by

$$f_n = \sum_{k=-N/2+1}^{N/2} F_k e^{i \frac{2\pi k n}{N}}. \tag{5.10.3}$$

Note that the index k in equation (5.10.3) runs from $[-N/2+1, ..., +N/2]$ so that it corresponds with equation (5.10.2). A plot of $|F_k|^2$ would show the computed *approximate* power spectrum. How closely does F_k correspond to c_k, for $k \in [-N/2+1, +N/2]$?

We can determine the precise relationship between F_k and c_k as follows. Recall the *orthogonality* relation

$$\sum_{p=0}^{N-1} e^{i \frac{2\pi p(l-k)}{N}} = N\delta_{k,l} \tag{5.10.4}$$

which can be written (by shifting the index) as

$$\sum_{p=-N/2+1}^{N/2} e^{i \frac{2\pi p(l-k)}{N}} = N\delta_{k,l}. \tag{5.10.5}$$

It therefore follows from equations (5.10.5) and (5.10.3) that

$$F_l = \frac{1}{N} \sum_{n=-N/2+1}^{N/2} f_n e^{-i \frac{2\pi n l}{N}} \tag{5.10.6}$$

Substituting equation (5.10.2) into (5.10.6) gives

$$\begin{aligned} F_l &= \frac{1}{N} \sum_{n=-N/2+1}^{N/2} e^{-i \frac{2\pi n l}{N}} \sum_{k=-\infty}^{\infty} c_k e^{i \frac{2\pi k n}{N}} \\ &= \sum_{k=-\infty}^{\infty} c_k \frac{1}{N} \sum_{n=-N/2+1}^{N/2} e^{i \frac{2\pi n(k-l)}{N}} \end{aligned} \tag{5.10.7}$$

Since $k = -\infty, .., \infty$ in equation (5.10.7), equation (5.10.5) must be modified to give

$$\sum_{p=-N/2+1}^{N/2} e^{i\,\frac{2\pi p(l-k)}{N}} = N(\delta_{k,l} + \delta_{k,l+N} + \delta_{k,l-N} + \delta_{k,l+2N} + ...). \quad (5.10.8)$$

Consequently, equations (5.10.8) and (5.10.7) combine to give

$$F_l = c_l + c_{l+N} + c_{l-N} + c_{l+2N} + c_{l-2N} + ... \quad (5.10.9)$$

Equation (5.10.9) has the following interpretation: if the original signal contains frequencies with complex frequency $\frac{|p|}{T} > \frac{N}{2T}$ ($\frac{N}{2T}$ is the *Nyquist* frequency), then the power in this frequency is *aliased* down to a lower frequency. In other words a plot of $|F_k|^2$ will be misleading, since some the power at a given k may actually be coming from a higher frequency. This effect can result in poor images captured by digital cameras. The cure for this problem is to sample at a higher rate, or to filter the signal before digitization.

5.11 Correlation Function

Consider two real periodic input arrays $y_i, z_i; i = 0, ..., N-1$. The correlation function ϕ_n is defined as

$$\phi_n = \frac{1}{N} \sum_{i=0}^{i=N-1} y_{i+n} z_i \quad . \quad (5.11.1)$$

Suppose the maximum of ϕ occurs at $n = p$. Then, this tells us that if we shift the array z by p positions to the right, then this gives us the best match (the maximum positive correlation) between arrays y and z. We can regard z as the reference signal, and we can imagine sliding z to the right by p units in order to get the best match with y.

A naive evaluation of equation (5.11.1) would require N^2 flops. However, we can do this using FFTs in $O(N \log N)$ operations.

Let

$$\begin{aligned} y_{l+n} &= \sum_{m=0}^{N-1} Y_m W^{m(l+n)} \\ z_l &= \sum_{r=0}^{N-1} Z_r W^{rl} \quad , \end{aligned} \quad (5.11.2)$$

where $Y = FFT(y)$ and $Z = FFT(z)$. Let $\Phi = FFT(\phi)$, i.e.

$$\Phi_k = \frac{1}{N} \sum_{n=0}^{N-1} \phi_n W^{-nk} \quad . \quad (5.11.3)$$

Substituting definition (5.11.1) into equation (5.11.3) and then using equation (5.11.2) gives

$$\begin{aligned}
\Phi_k &= \frac{1}{N^2}\sum_{n=0}^{N-1}\sum_{m=0}^{N-1}\sum_{r=0}^{N-1}\sum_{l=0}^{N-1} Y_m Z_r W^{-n(k-m)} W^{l(r+m)} \\
&= \frac{1}{N^2}\sum_{n=0}^{N-1}\sum_{m=0}^{N-1}\sum_{r=0}^{N-1} Y_m Z_r W^{-n(k-m)} \sum_{l=0}^{N-1} W^{l(r+m)} \\
&= \frac{1}{N^2}\sum_{n=0}^{N-1}\sum_{m=0}^{N-1}\sum_{r=0}^{N-1} Y_m Z_r W^{-n(k-m)} \delta_{r,N-m} N \\
&= \frac{1}{N}\sum_{n=0}^{N-1}\sum_{m=0}^{N-1} Y_m Z_{N-m} W^{-n(k-m)} \\
&= \frac{1}{N}\sum_{m=0}^{N-1} Y_m Z_{-m} \sum_{n=0}^{N-1} W^{n(m-k)} \\
&= \frac{1}{N}\sum_{m=0}^{N-1} Y_m Z_{-m} \delta_{m,k} N \\
&= Y_k Z_{-k} \\
&= Y_k \overline{Z_k} \quad , \qquad (5.11.4)
\end{aligned}$$

where the last step follows since z is real (by conjugate symmetry).

So, we can compute the FFT of the correlation function using equation (5.11.4). We then apply the inverse FFT to obtain the actual correlation function (5.11.1).

Note that it is often the case that the input arrays are samples of a real signal, which is not necessarily periodic. To avoid *wrap-around pollution*, we typically pad both arrays with zeros to twice their original length.

5.12 A Two Dimensional FFT

In image processing applications, a gray scale image is represented by a 2-D array of gray scale values $f_{n,j}, n = 0, ..., N-1, j = 0, ..., M-1$. In this case, the two dimensional DFT uses two roots of unity, $W_N = e^{2\pi i/N}$ and $W_M = e^{2\pi i/M}$ and is defined as

$$F_{k,l} = \frac{1}{N\,M}\sum_{n=0}^{N-1}\sum_{j=0}^{M-1} f_{n,j} W_N^{-nk} W_M^{-jl} \ . \qquad (5.12.1)$$

Given a function which computes a 1-D FFT, how could you use this to compute the 2-D FFT above?

We can rewrite equation (5.12.1) as

$$\begin{aligned} F_{k,l} &= \frac{1}{N\,M}\sum_{n=0}^{N-1} W_N^{-nk}\sum_{j=0}^{M-1} f_{n,j}W_M^{-jl} \\ &= \frac{1}{N}\sum_{n=0}^{N-1} W_N^{-nk}H_{n,l} \end{aligned} \tag{5.12.2}$$

where $H_{n,l} = \frac{1}{M}\sum_{j=0}^{M-1} f_{n,j}W_M^{-jl}$. Thus $H_{n,l}$ can be computed efficiently by performing one-dimensional FFT's of length M on the rows of the input data array. The final computation of $F_{k,l}$ is carried out by computing one-dimensional FFT's of length N on the columns of $H_{n,l}$. If the complexity of a one-dimensional FFT of length N is $CN\log_2 N$, where C is a constant, then the complexity of a two-dimensional FFT is $CNM(\log_2 M + \log_2 N)$.

5.13 The Continuous Fourier Transform

If we let the number of sample points become infinite, and let the spacing between samples tend to zero, we get the continuous fourier transform

$$F(k) = \int_{-\infty}^{\infty} e^{i\ 2\pi kt} f(t)\ dt \tag{5.13.1}$$

$$f(t) = \int_{-\infty}^{\infty} e^{-i\ 2\pi kt} F(k)\ dk. \tag{5.13.2}$$

For example, looking at equation (5.13.1), let's assume that $f(t) \simeq 0$, for $|t| > T/2$, so that

$$F(k) \simeq \int_{-T/2}^{T/2} e^{i\ 2\pi kt} f(t)\ dt. \tag{5.13.3}$$

If we sample the input signal at $t_m = (mT)/N$, and seek estimates for $F(k)$ only at the discrete points $F(k_n)$, with $k_n = n/T$, then equation (5.13.3) becomes

$$\begin{aligned} F(k_n) = F(n) &= \int_{-T/2}^{T/2} e^{i\ \frac{2\pi nm}{N}} f_m\ dt_m \\ &\simeq \sum_{m=-N/2+1}^{N/2} e^{i\ \frac{2\pi nm}{N}} f_m\ dt_m \end{aligned} \tag{5.13.4}$$

where we have approximated the integral by a discrete sum. Finally, if we let $dt_m = T/N$, we get

$$F(n) = \frac{T}{N}\sum_{m=-N/2+1}^{N/2} e^{i\ \frac{2\pi nm}{N}} f_m. \tag{5.13.5}$$

Similarly, if we seek discrete values $f(t_m)$ where $t_m = (mT)/N$ in equation (5.13.2), and we suppose that $F(k_n) = F_n = 0$ for $|n| > N/2$, then equation (5.13.2) becomes (noting that $k_n = n/T$)

$$\begin{aligned} f(t_m) = f_m &= \int_{k=-N/(2T)}^{N/(2T)} e^{-i\ 2\pi k(mT)/N} F(k)dk \\ &\simeq \sum_{n=-N/2+1}^{N/2} e^{-i\ 2\pi k_n(mT)/N} F(k_n)dk_n \\ &= \frac{1}{T} \sum_{n=-N/2+1}^{N/2} e^{-i\ \frac{2\pi mn}{N}} F_n \end{aligned} \tag{5.13.6}$$

where we have let $dk_n = 1/T$. Finally, the approximate forms of equations (5.13.1-5.13.2) are

$$\begin{aligned} F(n) &= \frac{T}{N} \sum_{m=-N/2+1}^{N/2} e^{i\ \frac{2\pi ni}{N}} f_m \\ f_m &= \frac{1}{T} \sum_{n=-N/2+1}^{N/2} e^{-i\ \frac{2\pi mn}{N}} F_n \end{aligned} \tag{5.13.7}$$

which we recognize as the DFT, apart from the factor $1/T$ which can be absorbed by redefining F_n.

We can also define a two-dimensional continuous fourier transform

$$\begin{aligned} F(k_1, k_2) &= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} e^{i\ 2\pi(k_1 x + k_2 y)} f(x, y)\ dx\ dy \\ f(x, y) &= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} e^{-i\ 2\pi(k_1 x + k_2 y)} F(k_1, k_2)\ dk_1\ dk_2 \end{aligned} \tag{5.13.8}$$

We can consider the DFT to be an approximation to the continuous fourier transform. In image processing applications, it is often convenient to develop an algorithm in terms of the 2-d continuous transform (5.13.8), and then convert to a 2-d DFT in the actual application.

5.14 Exercises for Discrete Fourier Analysis

1. Let $\{f_n\}$, $n = 0, 1, \ldots, N-1$, be N samples of a real signal, and N be even.
   (a) Show that $W^{-nk} + W^{-(N-n)k} = 2\cos(\frac{2\pi nk}{N})$.
   (b) Suppose the signal $\{f_n\}$ is an even function; i.e. $f_n = f_{N-n}$, $n = 1, 2, \ldots, N/2-1$. Show that F_k is real.

2. Suppose the signal $\{f_n\}$ is a square signal, defined as:

$$f_n = \begin{cases} 0 & \text{if } n = 0, 1, \ldots, \frac{N}{4} - 1 \text{ or } n = \frac{3N}{4}, \frac{3N}{4} + 1, \ldots, N - 1, \\ \\ 1 & \text{if } n = \frac{N}{4}, \frac{N}{4} + 1, \ldots, \frac{3N}{4} - 1. \end{cases}$$

Show that $F_{2k} = 0$, $k = 1, 2, \ldots, \frac{N}{2} - 1$.

3. For a given input sequence f_i, why is the first component of its DFT, F_0, always equal to the average of the components of f_i?

4. For a constant c determine the DFT of a signal $f_0 + c, \ldots, f_{N-1} + c$ in terms of the DFT of the signal $f_0, \ldots, f_{N-1}$

5. Calculate the Discrete Fourier Transform of the following periodic time sequences by hand, both using the direct DFT formula.

(i)
$$f[n] = (1, 2, 2, 1) \quad (n = 0, \ldots, 3;\ N = 4)$$

(ii)
$$f[n] = (1, 2, 3, 2) \quad (n = 0, \ldots, 3;\ N = 4)$$

Why is the resulting transform real in this case?

6) **(Parseval's Theorem)** Let f_n, $n = 0, \ldots, N - 1$ be given data values (real or complex) and let F_k, $k = 0, \ldots, N - 1$ be the DFT of f_n. Show that

$$\sum_{k=0}^{N-1} F_k \, \overline{F_k} = \frac{1}{N} \sum_{n=0}^{N-1} f_n \, \overline{f_n} \, .$$

7) Derive an algorithm for determining the FFT of two real signals of the same length by performing a single FFT of a complex signal.

Chapter 6

Numerical Linear Algebra

Consider an $n \times n$ matrix A, a solution vector x and a right hand side vector b. We wish to solve

$$Ax = b \tag{6.0.1}$$

The classical way to solve a linear system of equations is via Gaussian elimination. Typically, solving linear systems of equations $Ax = b$, is first learned via a process of reducing the given system of equations to a new system in which the unknowns, x_i, are systematically eliminated. It may be necessary to re-order the equations to accomplish this, for example by the use of equation pivoting.

However, the standard computer techniques referred to as Gaussian elimination for solving $Ax = b$ are based on factoring A into triangular factors. This view of Gaussian elimination has the following major steps:

1. Compute triangular factors L and U from A
2. Solve $Lz = b$
3. Solve $Ux = z$

Of course this may not be possible without re-ordering the equations. In matrix terms, re-ordering the equations is accomplished by multiplying A and b by a 'permutation' matrix, P, that is solving an equivalent system $PAx = Pb$.

6.1 Solving via Matrix Factorization

In fact it is not difficult to see that a triangular factorization of A is related to the row reduction process. Here the factorization algorithm for factoring $A = LU$, where L is unit lower triangular, and U is upper triangular is presented.

LU factorization (6.1.1)

For $k = 1, .., n$
  For $i = k + 1, ..., n$
    $mult := a_{ik}/a_{kk}$

$a_{ik} := mult$
For $j = k+1, ..., n$
$a_{ij} := a_{ij} - mult * a_{kj}$
EndFor
EndFor
EndFor

The original entries of A have been overwritten by L and U, that is, the strictly lower part of the modified A array contains the strictly lower part of L (the unit diagonal is understood). The upper triangular part of the A array now contains U.

Example 6.1.1 *We will see in the next section and the example below that the LU factorization process is almost identical to Gaussian elimination, the main difference being that we wish to retain the various multiplicative factors. As mentioned in the previous section, we store the multiplicative factors (rather than zero) in place of the elements eliminated by Gaussian elimination.*

Let

$$A = \begin{bmatrix} 10 & -7 & 0 \\ -3 & 2 & 6 \\ 5 & -1 & 5 \end{bmatrix}.$$

First stage of LU factorization of A:

$$\begin{bmatrix} 10 & -7 & 0 \\ -3 & 2 & 6 \\ 5 & -1 & 5 \end{bmatrix} \xrightarrow{R2-(\frac{-3}{10})R1} \begin{bmatrix} 10 & -7 & 0 \\ -0.3 & -0.1 & 6 \\ 5 & -1 & 5 \end{bmatrix} \xrightarrow{R3-(\frac{5}{10})R1} \begin{bmatrix} 10 & -7 & 0 \\ -0.3 & -0.1 & 6 \\ 0.5 & 2.5 & 5 \end{bmatrix}$$

Proceeding with the second (and final) stage of factorization, we obtain

$$\begin{bmatrix} 10 & -7 & 0 \\ -0.3 & -0.1 & 6 \\ 0.5 & -25 & 155 \end{bmatrix}$$

from which we extract the factors

$$L = \begin{bmatrix} 1 & 0 & 0 \\ -0.3 & 1 & 0 \\ 0.5 & -25 & 1 \end{bmatrix} \quad \textit{and} \quad U = \begin{bmatrix} 10 & -7 & 0 \\ 0 & -0.1 & 6 \\ 0 & 0 & 155 \end{bmatrix}.$$

One can easily verify that $LU = A$ in this example.

Note that no extra work or storage is required for factoring $A = LU$ compared to the *standard* elimination procedure. Now, to solve equation (6.0.1), we note that

$$Ax = LUx = b \tag{6.1.2}$$

Let $z = Ux$, then it follows from equation (6.1.2) that

$$\begin{aligned} Lz &= b \\ Ux &= z \end{aligned} \tag{6.1.3}$$

Equations (6.1.3) are easy to solve. Let the entries of $(L)_{ij} = l_{ij}$, where we recall that L is unit lower triangular. Then the forward solve ($Lz = b$) algorithm is

Forward Solve

For $i = 1, ..., n$
    $z_i := b_i$
    For $j = 1, ..., i-1$
        $z_i := z_i - l_{ij} * z_j$
    EndFor
EndFor

Let the entries of $(U)_{ij} = u_{ij}$. Then the back solve ($Ux = z$) algorithm is

Back Solve

For $i = n, ..., 1$
    $x_i := z_i$
    For $j = i+1, ..., n$
        $x_i := x_i - u_{ij} * x_j$
    EndFor
    $x_i := x_i / u_{ii}$
EndFor

Example 6.1.2 *Suppose we wanted to solve the system $Ax = b$ for the matrix A defined in the previous example, and the vector $b = (7, 4, 6)$. We have*

$$L = \begin{bmatrix} 1 & 0 & 0 \\ -0.3 & 1 & 0 \\ 0.5 & -25 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 10 & -7 & 0 \\ 0 & -0.1 & 6 \\ 0 & 0 & 155 \end{bmatrix}.$$

Forward solve (Lz=b):

$$\begin{bmatrix} 1 & 0 & 0 \\ -0.3 & 1 & 0 \\ 0.5 & -25 & 1 \end{bmatrix} \begin{bmatrix} z_0 \\ z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} 7 \\ 4 \\ 6 \end{bmatrix} \implies \begin{cases} z_0 = 7 \\ z_1 = 4 + 0.3(7) = 6.1 \\ z_2 = 6 - 0.5(7) + 25(6.1) = 155 \end{cases}$$

Note that after the forward solve, we get the right hand side that would normally appear when doing Gaussian elimination on $Ax = b$.

Similarly, the backward solve (Ux=z) yields the vector $x = (0, -1, 1)$ which can easily be verified as the solution to $Ax = b$.

One of the advantages of factoring $A = LU$ is that we can solve for multiple right hand sides simply by doing multiple forward and back solves. Can you think of an efficient way to form A^{-1} given that you have already found $A = LU$?

6.1.1 Gaussian Elimination ≡ Matrix Factorization

In this section, we show how Gaussian elimination and LU factorization are closely related. The elementary equation reduction form of Gaussian elimination can be summarized as follows.

First step of the reduction:

For $i = 2, ..., n$ (replace equation i by)

$$\text{equation } i \leftarrow \text{equation } i \text{ - } (a_{i,1}/a_{1,1}) \text{ (equation 1)}$$

End for

The result is a new system with

1. a subsystem of n-1 equations in unknowns $(x_2, x_3, ...x_n)$
2. a single equation involving $(x_1, x_2, ...x_n)$

The solution of the new and old systems are the same.

The next step of reduction, which is the final step, uses the first step recursively.

a) Apply steps (1) and (2) to solve the smaller subsystem (1) for $(x_2, x_3, ...x_n)$, Do this, and step b), recursively for k = 2, 3, ... , n-1 of equation reduction.

b) Solve (2) for x_1, using $(x_2, x_3, ...x_n)$ from step a).

After k invocations of the recursion of the reduction process,we have

1. a subsystem of n-k equations in unknowns $(x_{k+1}, \ldots x_n)$
2. a subsystem of k equations. In this subsystem, the first equation involves all the unknowns($x_1, x_2, ...x_n$). For $j = 2 \ldots k$ the jth equation does not involve unknowns $x_1, \ldots x_{j-1}$

How does the recursion end?

What if x_1 doesn't appear in equation 1, i.e $a_{1,1} = 0$? Or if the same thing happens in trying to apply subsequent reductions?

Row Reduction via Matrix Multiplication

The equation reduction process described above can be given in terms of matrix multiplications.

Step 1 of the equation reduction process,

For $i = 2, ..., n$

(replace row i by)

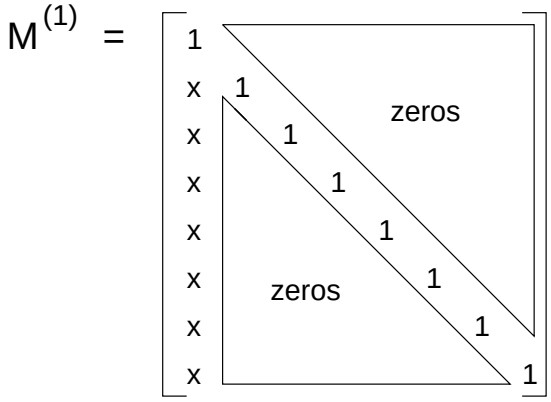
row $i \leftarrow$ row i -$(a_{i,1}/a_{1,1})$ (row 1)

End for

can be expressed as a matrix multiplication, using an $n \times n$ matrix, $M^{(1)}$

$$M^{(1)}A^{(1)} = A^{(2)}$$

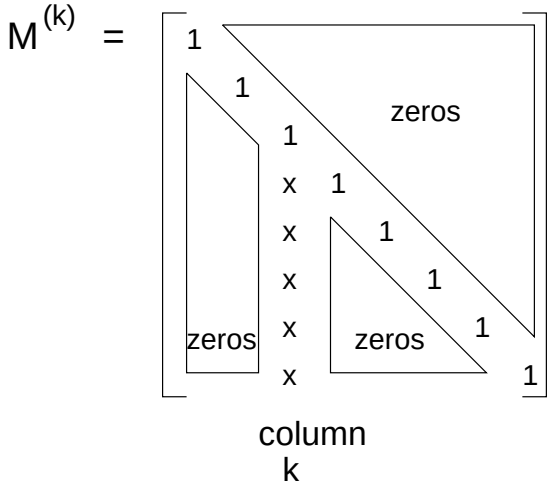
where we let $A^{(1)}$ denote the original matrix. The pattern of entries in $M^{(1)}$ is as shown, where 'x' in the first column is a number.



To pick a specific case, do you see why $M^{(1)}_{3,1} = -a^{(1)}_{3,1}/a^{(1)}_{1,1}$? In general, $M^{(1)}_{k,1} = -a^{(1)}_{k,1}/a^{(1)}_{1,1}$ for $k = 2 \ldots n$.

Thus, the kth step of the row reduction process can be described as a matrix multiplication in essentially the same way. That is, it can be expressed as shown in (6.1.4) using $n \times n$ matrix $M^{(k)}$ with the pattern as shown, for $k = 1, 2, 3, \ldots n-1$.

$$M^{(k)}A^{(k)} = A^{(k+1)} \tag{6.1.4}$$



Note that the comment made above for the first step of equation reduction have their analogs for the kth step, and also note that we have assumed $A^{(k)}_{k,k} \neq 0$.

The above discussion has recast the process of equation reduction into matrix notation and expressed it in terms of matrix multiplications. We now want to manipulate this form of expression to describe some algebraic facts about solving $Ax = b$.

Set $U = A^{(n)}$. This is the standard designation of this upper triangular matrix in the context of Gaussian elimination. We see that the matrix description of the kth reduction step shown at (6.1.4) can be broken into two steps:

$$M^{(k)}A^{(k)} = A^{(k+1)} \quad \texttt{and} \quad M^{(k)}b^{(k)} = b^{(k+1)}. \tag{6.1.5}$$

The first of these implies

$$A^{(k)} = (M^{(k)})^{-1}A^{(k+1)} \tag{6.1.6}$$

and the second implies

$$b^{(k)} = (M^{(k)})^{-1}b^{(k+1)}. \tag{6.1.7}$$

If we look at (6.1.6) for $k = n - 1$, we see that $A^{(n-1)} = (M^{(n-1)})^{-1}A^{(n)}$. In fact,

$$A^{(n-k)} = (M^{(n-k)})^{-1} \dots (M^{(n-1)})^{-1}A^{(n)} \tag{6.1.8}$$

Now each inverse matrix, $(M^{(n-j)})^{-1}$, in (6.1.8) is lower triangular, unit diagonal and the product of these matrices is also lower triangular, unit diagonal. Thus setting $k = n - 1$, introducing L for the product of inverse matrices in (6.1.8), and setting $A^{(n)} = U$, we have

$$A^{(1)} = LU \tag{6.1.9}$$

for lower triangular, unit diagonal matrix L and upper triangular matrix U. Finally, we can conclude from $A = A^{(1)}$ that the matrix of coefficients of the original system of equations can be written as a product of two factors

$$A = LU \tag{6.1.10}$$

one is a lower triangular, unit diagonal matrix L and the other is an upper triangular matrix U.

If we pursued the same development of (6.1.7), we could conclude that

$$b = Lb^{(n)} \tag{6.1.11}$$

for the same matrix, L, that is the factor of A. We can further observe that we can get the solution, x, by solving

$$Ux = b^{(n)} \tag{6.1.12}$$

6.1.2 Cost of Matrix Factorization

The running time of the above algorithms will be proportional to the number of flops. Since most statements consist of *linked triad* operations like

$$z_i := z_i - l_{ij} * z_j$$

usually a multiplication is paired with an addition or subtraction. So, often a measure of work is to simply count the number of *multiply-adds*. Usually, the total number of *multiply-adds* is about one-half the total number of *additions* + *subtractions* + *multiplies* + *divides*. In fact, both operation counts are called *number of flops*. There is no standard agreement on this terminology. We shall adopt the convention of counting the total number of operations, i.e. one multiply-add is two flops. The flop count will include the total number of *adds*+*multiplies*+*divides*+*subtracts*. A simple calculation gives the total number of flops for factoring an $n \times n$ matrix

$$\begin{aligned} \text{LU factor Work} &= \frac{2n^3}{3} + O(n^2) \text{ flops} \\ &= O(n^3) \text{ flops} \end{aligned} \tag{6.1.13}$$

The flop count for a forward or back solve is

$$\begin{aligned} \text{Forward + Back solve Work} &= 2n^2 + O(n) \\ &= O(n^2) \end{aligned} \tag{6.1.14}$$

Obviously, for large n, the factor is more expensive than the forward/back solve.

6.1.3 Pivoting and Factorization

Each of the closely related views of GE given in the opening section can fail if a division by zero is required at some stage. This problem can be reduced if we carry out some form of *row pivoting*.

For the equation reduction view, row pivoting defines a re-ordering of the equations. For the augmented matrix view, it has the effect of re-ordering the rows of $[A^{(k)}, b^{(k)}]$. This operation can be described in matrix terms as a multiplication by a *permutation* matrix. A permutation matrix is a matrix of entries either 0 or 1, with exactly one entry, 1, in each row and column. For example, the permutation matrix that interchanges the second and third rows of a 3×3 matrix is

$$P = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

In the matrix factorization view of Gaussian elimination, (6.1.1), if $a_{kk}^{(k)} = 0$ at some stage, then we examine all entries in the kth column below $a_{kk}^{(k)}$. If

$$\max_{j=k,..,n} |a_{jk}^{(k)}| = |a_{k^*k}^{(k)}|, \tag{6.1.15}$$

then we swap row k^* with row k, and use $a_{k^*k}^{(k)}$ to form the multiplier. Note that at least one of $a_{kk}^{(k)}, a_{k+1,k}^{(k)}, ..., a_{nk}^{(k)} \neq 0$, otherwise the matrix is singular. Evidently then, this produces a re-ordering of the rows of A that could be described by matrix multiplication by a permutation matrix, P. This same re-ordering would have to be applied to the right hand side vector b, to get a new system with the same solution, x, as the original system, $Ax = b$.

Example 6.1.3 *Consider the system $Ax = b$ given by*

$$A = \begin{bmatrix} 10 & -4 & 0 \\ -5 & 2 & 6 \\ 5 & -1 & 5 \end{bmatrix}, \quad b = \begin{bmatrix} 7 \\ 6 \\ 4 \end{bmatrix}.$$

After the first stage of LU factorization, we reach the following state:

$$\begin{bmatrix} 10 & -4 & 0 \\ -0.5 & 0 & 6 \\ 0.5 & 1 & 5 \end{bmatrix}.$$

At this point, division by zero would be encountered because of the element $a_{22} = 0$, unless we use row pivoting. After swapping rows 2 and 3, the factorization is complete. Thus, we have obtained

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ -0.5 & 0 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 10 & -4 & 0 \\ 0 & 1 & 5 \\ 0 & 0 & 6 \end{bmatrix}.$$

where $PA = LU$. The system $Ax = b$ can now be solved via the system $LUx = Pb$. □

Note: If the diagonal element at some stage, $a_{kk}^{(k)}$, is nonzero but *small*, magnification of round-off error can occur. This is because $m = \left| \frac{a_{i,k}^{(k)}}{a_{k,k}^{(k)}} \right|$ is large when $|a_{kk}^{(k)}|$ is small. Assuming this is the case, and that some element, $a_{kj}^{(k)}$, has a round-off error of δ, the operation $m * a_{kj}^{(k)}$ will introduce a large error, $m\delta$, which can lead to a loss of accuracy in the computed solution. In order to limit the growth of round-off error, row pivoting can be carried out *unconditionally* at each step.

In summary, factoring A using row pivoting, followed by re-ordering b and forward and backward solving is arithmetically the same as factoring PA using (6.1.1), followed by solving $Lz = Pb$ and $Ux = z$.

Algorithmically, however, the process of re-ordering is intertwined with the factorization, to produce representations of P, L, and U from A. Thus, standard computer oriented algorithms for solving $Ax = b$ have two stages:

(a) from data A, compute P, L and U such that $PA = LU$.

(b) from P, L, U and b, compute x

The amount of arithmetic required for stage (a) continues to be $\frac{2n^3}{3} + O(n^2)$ flops and for stage (b) is $2n^2 + O(n)$ flops.

Stage (a) is the task of the `lu(A)` command; i.e.

```
[L, U, P] = lu(A)    in Matlab
P, L, U = lu(A)      in Python.
```

6.2 Condition numbers and Norms

The norm of a vector is a measure of its size. Let x be a vector with components

$$x = \begin{bmatrix} x_1 \\ x_2 \\ | \\ x_n \end{bmatrix} \tag{6.2.1}$$

and we define

$$\begin{aligned} \|x\|_1 &= \sum_{i=1}^{n} |x_i| \\ \|x\|_2 &= \left(\sum_{i=1}^{n} x_i^2 \right)^{1/2} \\ \|x\|_\infty &= \max_i |x_i|. \end{aligned}$$

Usually, these *norms* are referred to as the *p-norm* i.e.

$$\|x\|_p \ ; \ p = 1, 2, \infty. \tag{6.2.2}$$

Python computes these vector norms using the `np.linalg.norm` command.

6.2.1 Properties of Norms

$$\begin{aligned} \|x\| &= 0 \rightarrow x_i = 0, \forall i \\ \|\alpha x\| &= |\alpha| \|x\| \ ; \ \alpha = \text{ scalar} \\ \|x + y\| &\leq \|x\| + \|y\| \end{aligned}$$

6.2.2 Matrix Norms

Similar to vector norms, we can define a matrix norm for an $n \times n$ matrix A by

$$\|A\| = \max_{\|x\| \neq 0} \frac{\|Ax\|}{\|x\|}$$

for any $\|\|_p$ norm. It can be shown that (see any linear algebra text)

$$\begin{aligned} \|A\|_1 &= \max_j \sum_{i=1}^{n} |a_{ij}| \\ &= \text{ max absolute column sum} \\ \|A\|_\infty &= \max_i \sum_{j=1}^{n} |a_{ij}| \\ &= \text{ max absolute row sum.} \end{aligned}$$

For the $\|\|_2$ norm, we have to consider the eigenvalues of A. Recall that A has an *eigenvalue* λ (a scalar) associated with a nonzero vector x if

$$Ax = \lambda x$$

where the vector x is the *eigenvector* associated with the eigenvalue λ. If $\lambda_i, i = 1, .., n$ are the eigenvalues of $A^t A$, then

$$\|A\|_2 = \max_i |\lambda_i|^{1/2}.$$

Python also computes these matrix norms using the `np.linalg.norm` command.

Note that for any $n \times n$ matrices A and B, and and n length vector x, we have

$$\begin{aligned}
\|A\| &= 0 \text{ iff } a_{ij} = 0; \forall i, j \\
\|\alpha A\| &= |\alpha| \|A\| \ ; \ \ \alpha = \text{ scalar} \\
\|A + B\| &\leq \|A\| + \|B\| \\
\|Ax\| &\leq \|A\| \|x\| \\
\|AB\| &\leq \|A\| \|B\| \\
\|I\| &= 1 \ ; \ I = \text{ identity matrix.}
\end{aligned}$$

6.2.3 Conditioning

Suppose we perturb the rhs vector b in

$$Ax = b.$$

What happens to x? Let's replace b by $b + \Delta b$, which means that x will change to $x + \Delta x$, which gives

$$A(x + \Delta x) = b + \Delta b. \tag{6.2.3}$$

Since if x is the exact solution, then $Ax = b$, so that equation (6.2.3) becomes

$$\Delta x = A^{-1} \Delta b. \tag{6.2.4}$$

Noting that $Ax = b$, so that

$$\|b\| \leq \|A\| \|x\|, \tag{6.2.5}$$

and from equation (6.2.4) we obtain

$$\|\Delta x\| \leq \|A^{-1}\| \|\Delta b\|. \tag{6.2.6}$$

Now, equation (6.2.5) gives

$$\|x\| \geq \frac{\|b\|}{\|A\|} \quad \text{or} \quad \frac{\|A\|}{\|b\|} \geq \frac{1}{\|x\|}. \tag{6.2.7}$$

Equations (6.2.6) and (6.2.7) give

$$\frac{\|\Delta x\|}{\|x\|} \leq \|A\|\|A^{-1}\|\frac{\|\Delta b\|}{\|b\|}. \tag{6.2.8}$$

Note that the above is true for any $\|\cdot\|_p$ norm.

Let $\kappa(A) = \|A\|\|A^{-1}\|$ be the *condition number* of A. So, equation (6.2.8) says that

$$\text{relative change in x} \quad = \quad \text{condition number} \times \text{relative change in b}\ .$$

If $\kappa(A)$ is large, then the problem *may* be ill-conditioned. The word *may* is used since the bound is not very sharp, and may grossly overestimate the problem. On the other hand, if $\kappa(A)$ is small (near one), then, for sure, the problem is well-conditioned.

Now, suppose we perturb the matrix elements A, i.e.

$$(A + \Delta A)(x + \Delta x) = b, \tag{6.2.9}$$

then, assuming $Ax = b$, i.e. x is the exact solution, equation (6.2.9) implies

$$\begin{aligned} A\Delta x &= -\Delta A(x + \Delta x) \\ \Delta x &= -A^{-1}\left[\Delta A(x + \Delta x)\right], \end{aligned} \tag{6.2.10}$$

which then gives

$$\|\Delta x\| \leq \|A^{-1}\|\|\Delta A\|\|x + \Delta x\|, \tag{6.2.11}$$

or

$$\begin{aligned} \frac{\|\Delta x\|}{\|x + \Delta x\|} &\leq \|A^{-1}\|\|A\|\frac{\|\Delta A\|}{\|A\|} \\ &= \kappa(A)\frac{\|\Delta A\|}{\|A\|}. \end{aligned} \tag{6.2.12}$$

Once again the condition number appears. So, a measure of the sensitivity to changes in A or b is the condition number. Note that

- $\kappa(A) \geq 1$
- $\kappa(\alpha A) = \kappa(A)\ ;\ \alpha =$ scalar.

6.2.4 The residual

The accuracy of the computed solution is sometimes measured by the size of the *residual*, i.e. let the computed solution to $Ax = b$ be $x + \Delta x$ (it's not exact). Let

$$r = b - A(x + \Delta x), \tag{6.2.13}$$

i.e. if the residual $r = 0$, then $\Delta x = 0$, and we have the exact solution. From equation (6.2.13) we have

$$A(x + \Delta x) = b - r.$$

Similar analysis as in equations (6.2.3-6.2.8) gives

$$\frac{\|\Delta x\|}{\|x\|} \leq \kappa(A)\frac{\|r\|}{\|b\|}. \tag{6.2.14}$$

Thus, a small residual does not necessarily imply a small error, if $\kappa(A)$ is large. For example, it is unavoidable that

$$\frac{\|r\|}{\|b\|} \simeq \epsilon_{machine}\,,$$

then equation (6.2.14) implies that

$$\frac{\|\Delta x\|}{\|x\|} \leq \kappa(A)\,\epsilon_{machine}\,,$$

which may not be small if the problem is poorly conditioned.

For implementation in floating point arithmetic, Gaussian elimination with pivoting is as stable and accurate as any other method. This is one of the reasons that it is so commonly used. It is known that Gaussian elimination with pivoting produces a computed solution $\hat{x}$ which satisfies

$$(A+E)\,\hat{x} = b\,,$$

where $\|E\| = \epsilon_{machine}\,\|A\|$, i.e. Gaussian elimination with pivoting solves a *nearby* problem exactly. From equation (6.2.12), we have

$$\frac{\|x-\hat{x}\|}{\|\hat{x}\|} \leq \kappa(A)\,\epsilon_{machine}\,.$$

Note that conditioning is a property of the problem, not a property of the equation solving algorithm.

6.3 Exercises for Numerical Linear Algebra

1. True or false: If x is any n-vector, then $||x||_1 \geq ||x||_\infty$.
2. True or false: If $||A|| = 0$, then $A = 0$.
3. Derive the algorithm to compute the UL decomposition of a matrix A, namely,

   $$A = UL$$

   where U and L are the upper and lower triangular matrices, respectively. We assume that no zero pivots will occur in this case, so no pivoting strategy is needed.
4. Assume that you are given an LU factorization of an $n \times n$ matrix A. Show how to use this to solve

   $$A^2\vec{x} = \vec{b}$$

   with computational complexity $O(n^2)$.

5. Assume that you are given the decomposition $A = LU$ where A is an $n \times n$ matrix. Describe how you can use this decomposition to solve the system $A^T \vec{x} = \vec{b}$ with computational complexity $O(n^2)$.

6. Classify each of the following matrices as well-conditioned or ill-conditioned, and provide a brief explanation for you answer in each case.

$$\begin{bmatrix} 10^{10} & \\ & 10^{-10} \end{bmatrix} \quad \begin{bmatrix} 10^{10} & \\ & 10^{10} \end{bmatrix} \quad \begin{bmatrix} 10^{-10} & \\ & 10^{-10} \end{bmatrix}$$

7. Show that the following two definitions of condition number are equivalent:

$$\kappa(A) = \|A\|_2 \|A^{-1}\|_2; \qquad \kappa(A) = \frac{\max_x \frac{\|Ax\|_2}{\|x\|_2}}{\min_x \frac{\|Ax\|_2}{\|x\|_2}}$$

8. The LU decomposition for the following famous $n \times n$ tridiagonal matrix

$$\begin{bmatrix} 2 & -1 & 0 & & \cdots & 0 & 0 \\ -1 & 2 & -1 & & & 0 & 0 \\ 0 & -1 & 2 & -1 & & 0 & 0 \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & \ddots & \ddots & \\ 0 & 0 & 0 & & -1 & 2 & -1 \\ 0 & 0 & 0 & & & -1 & 2 \end{bmatrix}_{n \times n}$$

has a very simple form $A = LU$ where

$$L = \begin{bmatrix} 1 & 0 & & \cdots & 0 & 0 \\ l_2 & 1 & 0 & & & 0 \\ 0 & l_3 & 1 & & & \\ & & \ddots & \ddots & & \\ 0 & 0 & & & 1 & 0 \\ 0 & 0 & & & l_n & 1 \end{bmatrix} \qquad U = \begin{bmatrix} u_1 & -1 & & \cdots & 0 & 0 \\ 0 & u_2 & -1 & & & 0 \\ 0 & 0 & u_3 & -1 & & \\ & & & \ddots & \ddots & \\ 0 & 0 & & & u_{n-1} & -1 \\ 0 & 0 & & & 0 & u_n \end{bmatrix}$$

Derive the recurrence relations for l_i and u_i. If n is large, Show that u_i monotonically decreases and approaches a limit. Provide a better estimate of this limit.

What you can say about the sequence l_i?

9. The following matrix has an interesting sparsity pattern.

$$S_n = \begin{bmatrix} * & * & & & & & * \\ * & * & * & & & & \\ & * & * & * & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & * & * & * \\ * & & & & & * & * \end{bmatrix}_{n \times n}$$

Design an LU factorization algorithm for this matrix which avoids operating on and creating new zero elements as much as possible. Present your code and the resulting factorization. If we have an $n \times n$ matrix with this sparsity pattern, what is the operation count for the factorization and solution?

Chapter 7

Google Page Rank

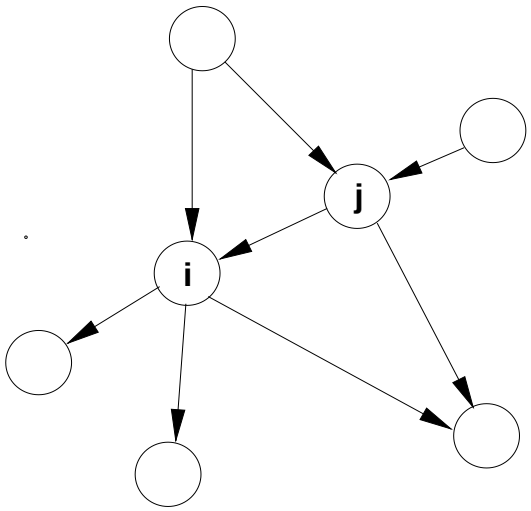
It is well known that various search engines use different techniques in order to give their "best" ranking for a given search query. This implies that a given search query will often give different results for different search engines. One can see this simply by running a few search examples on msn.com, yahoo.com or google.com.

The Google search engine is recognized for the quality of its search results. In this section, we discuss the algorithm used by Google called the Page Rank Algorithm to rank Web pages in importance.

7.1 Introduction

When one uses Google to search web pages for key words a set of pages is returned, each ranked in order of its importance. The question we try to answer in this section is how this ranking is obtained.

We can model the structure of the Web by a directed graph. The nodes in the graph represent web pages. A directed arc is drawn from node j to node i if there is link from page j to page i. Let $deg(j)$ be the outdegree of node j, that is, the number of arcs leaving node j. A typical graph is shown below. In this case $deg(j) = 2$ while $deg(i) = 3$.



The basic page rank idea is as follows. A link from web page j to web page i can be viewed as a vote on the importance of page i by page j. We will assume that all outlinks are equally important (this could be changed easily), so that the importance conferred on page i by page j is simply $1/deg(j)$ (assuming that there is a link from $j \to i$). But this simply gives some idea of the local importance of i, given that we are visiting page j. To get some idea of the global importance of page i, we have to have some idea of the importance of page j. This requires that we examine the pages which point to page j, and then we need to determine the importance of these pages, and so on.

Let us consider the hypothetical concept of a *random surfer*. This surfer selects each page in the Web in turn. From this initial page, the surfer then selects at random an outlink from this initial page, and visits this page. Another outlink is selected at random from this page, and so on. The surfer keeps track of the number of times each page is visited. After K visits, the surfer then begins the process again, by selecting another initial page. This algorithm is presented in (7.1.1). We assume that there are R pages in the web. There are at least two major problems with algorithm (7.1.1). Can you spot the problems?

Random Surfer Algorithm

$$
\begin{array}{l}
Rank(m) = 0 \ , \ m = 1, ..., R \\
\text{For } m = 1, ..., R \\
\quad j = m \\
\quad \text{For } k = 1, ..., K \\
\quad\quad Rank(j) = Rank(j) + 1 \\
\quad\quad \text{Randomly select outlink } l \text{ of page } j \\
\quad\quad j = l \\
\quad \text{EndFor} \\
\text{EndFor} \\
Rank(m) = Rank(m)/(K * R) \ , \ m = 1, ..., R
\end{array}
\tag{7.1.1}
$$

If K in algorithm (7.1.1) was sufficiently large, then we would get a good estimate of the importance of each page on the Web. However, this would be a very expensive algorithm, first because the number of web pages, R, is very large, and second because K needs to be large as well to ensure that we are getting a good sample of all the paths in the Web. We clearly need to do something smarter here.

7.2 Representing Random Surfer by a Markov Chain Matrix

We can solve the random surfer problem more efficiently using some numerical linear algebra. Let P be a matrix of size $R \times R$, where R is the number of pages in the Web. Note that P is a very large matrix! Let P_{ij} be the probability that the random surfer visits page i, given that he is at page j. Thus

$$
\begin{aligned}
P_{ij} &= \frac{1}{deg(j)} \ , \quad \text{if there is a link } j \to i \\
&= \quad 0 \ , \quad \text{otherwise.}
\end{aligned}
\tag{7.2.1}
$$

There are, however, two problems associated with letting the random surfer's movements be governed with the probability matrix P as it stands.

7.2.1 Dead End Pages

What happens if page i has no outlinks? In this case, once our random surfer visits page i, he will have no way of leaving this page. To avoid this problem, we suppose that the random surfer *teleports* to another page at random, if he encounters a dead end page. We suppose that this teleportation moves the surfer to any other page in the Web with

equal probability. More precisely, let $\mathbf{d}$ be an R-dimensional column vector such that

$$\begin{aligned} [\mathbf{d}]_i &= 1 \ , \ \text{if } deg(i) = 0 \\ &= 0 \ , \ \text{otherwise} \end{aligned} \tag{7.2.2}$$

and let $\mathbf{e} = [1, 1, ..., 1]^t$ be the R-dimensional column vector of ones. Then we define a new matrix P' of transition probabilities to include the teleportation property, as follows:

$$P' = P + \frac{1}{R}\mathbf{e} \cdot \mathbf{d}^t \ . \tag{7.2.3}$$

7.2.2 Cycling Pages

Suppose that page j contains only a link to page i, and page i contains only a link to page j. This will trap our random surfer in an endless cycle. We will again use the teleportation idea to make sure that the random surfer can escape from the boredom of visiting the same pages in cyclic fashion. Let $0 < \alpha < 1$ (Google uses $\alpha = .85$), and define a new matrix of probabilities

$$M = \alpha P' + (1 - \alpha) \cdot \frac{1}{R}\mathbf{e} \cdot \mathbf{e}^t \ . \tag{7.2.4}$$

What does equation (7.2.4) do? Essentially, we are saying that the surfer moves to other pages based on the usual idea of selecting an outlink at random, and, in addition, the surfer can also teleport any other page on the web with equal probability. We weight the usual outlink probabilities by α, and the teleportation probability by $(1-\alpha)$, so that the total probability of visiting the next page on the web is one. Once again, we can interpret M_{ij} as the probability that the random surfer will move from page j to page i. Note that from the definition of M we have that $0 < M_{ij} \leq 1$, and

$$\sum_i M_{ij} = 1 \, ; \tag{7.2.5}$$

i.e., each column sums to 1 ($colsum(M) = 1$). In other words, given that the random surfer is at page j, then the probability that he will end up at some new page is one.

The matrix M defined by equation (7.2.4) is referred to as the *Google Matrix*.

7.3 Markov Transition Matrices

Let us be a bit more formal here.

Definition 7.3.1 *A matrix Q is a Markov matrix if*

$$0 \leq Q_{ij} \leq 1 \quad \text{and} \sum_i Q_{ij} = 1 \quad . \tag{7.3.1}$$

Obviously, matrix M in equation (7.2.4) is a Markov matrix. Let $[\mathbf{p}]_i$ represent the probability that the random surfer is at page i so that $\mathbf{p}$ is an R dimensional column vector. We could, for example, specify that initially

$$[\mathbf{p}]_i = \frac{1}{R} \quad , \quad \forall i \tag{7.3.2}$$

that is, the random surfer visits each page initially with equal probability.

Definition 7.3.2 *A vector* $\mathbf{q}$ *is a probability vector if*

$$0 \leq [\mathbf{q}]_i \leq 1 \quad \text{and} \sum_i [\mathbf{q}]_i = 1 \quad . \tag{7.3.3}$$

Given that, at hop n, the random surfer is in the state represented by the probability vector $\mathbf{p}^n$, that is, the probability that the surfer is at page i is $[\mathbf{p}^n]_i$, then the probability vector at state $n+1$ is

$$\mathbf{p}^{n+1} = M\mathbf{p}^n \ . \tag{7.3.4}$$

We should verify that $\mathbf{p}^{n+1}$ is a probability vector. Since M is a Markov matrix, and $\mathbf{p}^n$ is a probability vector, then we have that

$$[\mathbf{p}^{n+1}]_i \geq 0 \quad , \quad \forall i \tag{7.3.5}$$

and as well

$$\begin{aligned} \sum_i [\mathbf{p}^{n+1}]_i &= \sum_i \sum_j M_{ij} [\mathbf{p}^n]_j \\ &= \sum_j [\mathbf{p}^n]_j \sum_i M_{ij} \\ &= \sum_j [\mathbf{p}^n]_j \\ &= 1 \quad . \end{aligned} \tag{7.3.6}$$

Thus $\mathbf{p}^n$ a probability vector implies then $\mathbf{p}^{n+1}$ is also a probability vector for each n.

7.4 Page Rank

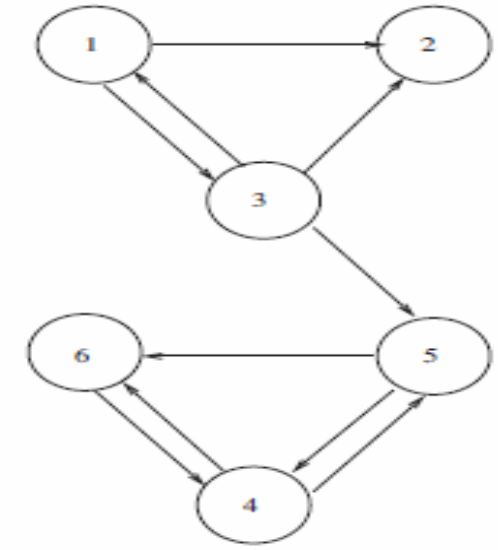
Given the assumption above, we can now precisely state the algorithm for determining the ranking of each page in the Web. Let

$$\mathbf{p}^0 = \frac{1}{R}\mathbf{e} \tag{7.4.1}$$

(that is, the random surfer visits each page initially with equal probability). Then the rank of page i is given by $[\mathbf{p}]_i^\infty$ where

$$\mathbf{p}^\infty = \lim_{k\to\infty} (M)^k \mathbf{p}^0 \ . \tag{7.4.2}$$

Example 7.4.1 *Consider the small web given by*



Then the P matrix for the web is given by

$$P = \begin{bmatrix} 0 & 0 & \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & 1 \\ 0 & 0 & \frac{1}{3} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \end{bmatrix}$$

with the d vector being

$$d = [0, 1, 0, 0, 0, 0]^T.$$

In this case the Q matrix is

$$\begin{aligned} Q &= P + \frac{1}{6} e d^T \\ &= \begin{bmatrix} 0 & \frac{1}{6} & \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{6} & \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{6} & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{6} & 0 & 0 & \frac{1}{2} & 1 \\ 0 & \frac{1}{6} & \frac{1}{3} & \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{6} & 0 & \frac{1}{2} & \frac{1}{2} & 0 \end{bmatrix} \end{aligned}$$

where we recall that $e = [1, 1, 1, 1, 1, 1]^T$. For a general α we then have

$$\begin{aligned} M &= \alpha Q + \frac{(1-\alpha)}{6} e e^T \\ &= \begin{bmatrix} \frac{1}{6} - \frac{1}{6}\alpha & \frac{1}{6} & \frac{1}{6}\alpha + \frac{1}{6} & \frac{1}{6} - \frac{1}{6}\alpha & \frac{1}{6} - \frac{1}{6}\alpha & \frac{1}{6} - \frac{1}{6}\alpha \\ \frac{1}{3}\alpha + \frac{1}{6} & \frac{1}{6} & \frac{1}{6}\alpha + \frac{1}{6} & \frac{1}{6} - \frac{1}{6}\alpha & \frac{1}{6} - \frac{1}{6}\alpha & \frac{1}{6} - \frac{1}{6}\alpha \\ \frac{1}{3}\alpha + \frac{1}{6} & \frac{1}{6} & \frac{1}{6} - \frac{1}{6}\alpha & \frac{1}{6} - \frac{1}{6}\alpha & \frac{1}{6} - \frac{1}{6}\alpha & \frac{1}{6} - \frac{1}{6}\alpha \\ \frac{1}{6} - \frac{1}{6}\alpha & \frac{1}{6} & \frac{1}{6} - \frac{1}{6}\alpha & \frac{1}{6} - \frac{1}{6}\alpha & \frac{1}{3}\alpha + \frac{1}{6} & \frac{5}{6}\alpha + \frac{1}{6} \\ \frac{1}{6} - \frac{1}{6}\alpha & \frac{1}{6} & \frac{1}{6}\alpha + \frac{1}{6} & \frac{1}{3}\alpha + \frac{1}{6} & \frac{1}{6} - \frac{1}{6}\alpha & \frac{1}{6} - \frac{1}{6}\alpha \\ \frac{1}{6} - \frac{1}{6}\alpha & \frac{1}{6} & \frac{1}{6} - \frac{1}{6}\alpha & \frac{1}{3}\alpha + \frac{1}{6} & \frac{1}{3}\alpha + \frac{1}{6} & \frac{1}{6} - \frac{1}{6}\alpha \end{bmatrix} \end{aligned}$$

while for $\alpha = .85$ *we have*

$$M = \begin{bmatrix} \frac{1}{40} & \frac{1}{6} & \frac{37}{120} & \frac{1}{40} & \frac{1}{40} & \frac{1}{40} \\ \frac{9}{20} & \frac{1}{6} & \frac{37}{120} & \frac{1}{40} & \frac{1}{40} & \frac{1}{40} \\ \frac{9}{20} & \frac{1}{6} & \frac{1}{40} & \frac{1}{40} & \frac{1}{40} & \frac{1}{40} \\ \frac{1}{40} & \frac{1}{6} & \frac{1}{40} & \frac{1}{40} & \frac{9}{20} & \frac{7}{8} \\ \frac{1}{40} & \frac{1}{6} & \frac{37}{120} & \frac{9}{20} & \frac{1}{40} & \frac{1}{40} \\ \frac{1}{40} & \frac{1}{6} & \frac{1}{40} & \frac{9}{20} & \frac{9}{20} & \frac{1}{40} \end{bmatrix}. \tag{7.4.3}$$

Repeated multiplication by the matrix M *gives the sequence*

$$v, \quad Mv, \quad M^2v, \quad M^3v, \quad \ldots, \quad M^{10}v$$

which is found to be

$$\begin{bmatrix} 0.16667 \\ 0.16667 \\ 0.16667 \\ 0.16667 \\ 0.16667 \\ 0.16667 \end{bmatrix}, \begin{bmatrix} 0.09583 \\ 0.16666 \\ 0.11944 \\ 0.26111 \\ 0.16666 \\ 0.19027 \end{bmatrix}, \begin{bmatrix} 0.08245 \\ 0.12318 \\ 0.08934 \\ 0.28118 \\ 0.19342 \\ 0.23041 \end{bmatrix}, \begin{bmatrix} 0.06776 \\ 0.10280 \\ 0.07749 \\ 0.32051 \\ 0.18726 \\ 0.24415 \end{bmatrix}, \ldots, \begin{bmatrix} 0.05205 \\ 0.07428 \\ 0.05782 \\ 0.34797 \\ 0.19975 \\ 0.26810 \end{bmatrix}.$$

Thus, for example, the ranking of importance of the pages in this small web are $4, 6, 5, 2, 3, 1$.

So, essentially, we start with $\mathbf{p}^0$ defined by equation (7.4.1) and then repeatedly multiply by M, to give us $\mathbf{p}^\infty$. There are two obvious questions

- Does iteration (7.4.2) converge?
- If yes, how fast does this iteration converge?

7.5 Convergence Analysis

In order to analyze the convergence of iteration (7.4.2), we have to review some basic linear algebra. (Maybe you haven't seen this before, but it's easy). Suppose we have a special nonzero vector $\mathbf{x}$ such that

$$Q\mathbf{x} = \lambda\mathbf{x} \tag{7.5.1}$$

where λ is a (possibly complex) scalar. The special vector $\mathbf{x}$ is an *eigenvector* of the matrix Q, with *eigenvalue* λ.

Note that equation (7.5.1) is equivalent to finding a nonzero solution of the linear system of equations $(\lambda I - Q)x = 0$ and so λ an eigenvalue of Q is equivalent to the matrix $\lambda I - Q$ being singular. This in turn implies that eigenvalues are roots of the *characteristic polynomial* $\det(\lambda I - Q)$ of Q.

Example 7.5.1 *Returning to the small web from the previous example we saw that*

$$M = \begin{bmatrix} \frac{1}{40} & \frac{1}{6} & \frac{37}{120} & \frac{1}{40} & \frac{1}{40} & \frac{1}{40} \\ \frac{9}{20} & \frac{1}{6} & \frac{37}{120} & \frac{1}{40} & \frac{1}{40} & \frac{1}{40} \\ \frac{9}{20} & \frac{1}{6} & \frac{1}{40} & \frac{1}{40} & \frac{1}{40} & \frac{1}{40} \\ \frac{1}{40} & \frac{1}{6} & \frac{1}{40} & \frac{1}{40} & \frac{9}{20} & \frac{7}{8} \\ \frac{1}{40} & \frac{1}{6} & \frac{37}{120} & \frac{9}{20} & \frac{1}{40} & \frac{1}{40} \\ \frac{1}{40} & \frac{1}{6} & \frac{1}{40} & \frac{9}{20} & \frac{9}{20} & \frac{1}{40} \end{bmatrix} \tag{7.5.2}$$

with a ranking vector computed via the power method yielding

$$[0.05205, \quad 0.07428, \quad 0.05782, \quad 0.34797, \quad 0.19975, \quad 0.26810]^T \tag{7.5.3}$$

after 10 *iterations.*

One can find that the lone eigenvector for eigenvalue 1 *for* M *is then seen to be*

$$v = [\frac{3080}{59569}, \quad \frac{4389}{59569}, \quad \frac{3420}{59569} \quad \frac{1184000}{3395433}, \quad \frac{9560}{47823}, \quad \frac{16000}{59569}]^T \tag{7.5.4}$$

which with floating point numbers becomes

$$v \approx [0.05170, \quad 0.07367, \quad 0.05741, \quad 0.34870, \quad 0.19990, \quad 0.26859]^T. \tag{7.5.5}$$

7.5.1 Some Technical Results

We are going to very briefly derive some technical properties of the eigenvalues of a Markov matrix here. If you are not a linear algebra fan, I suggest you skip to Section 7.5.2.

Theorem 7.5.2 *Every Markov matrix* Q *has* 1 *as an eigenvalue.*

Proof: The eigenvalues of Q and Q^t are the same (since Q and Q^t have the same characteristic polynomials). Since $Q^t\mathbf{e} = \mathbf{e}$, we have that $\lambda = 1$ is an eigenvalue of Q^t. Thus $\lambda = 1$ is an eigenvalue of Q.

Theorem 7.5.3 *Every (possibly complex) eigenvalue* λ *of a Markov matrix* Q *satisfies*

$$|\lambda| \leq 1 \quad . \tag{7.5.6}$$

Thus 1 *is the largest eigenvalue of* Q*.*

Proof: This result actually follows from the Gershgorin Circle Theorem (a result that can be found in linear algebra texts) applied to the eigenvalues of Q^t. Since the eigenvalues of Q and Q^t are the same, the theorem follows.

Definition 7.5.4 *A Markov matrix* Q *is a positive Markov matrix if*

$$Q_{ij} > 0 \quad , \quad \forall i, j \quad . \tag{7.5.7}$$

Theorem 7.5.5 *If Q is a positive Markov matrix, then there is only one linearly independent eigenvector of Q with $|\lambda| = 1$.*

Proof: See, for example, (Grimmett and Stirzaker, *Probability and Random Processes*, Oxford University Press, 1989.)

Remark 7.5.6 *Theorem 7.5.2 and 7.5.5 can be interpreted as follows. If Q is a positive Markov matrix, then there exists $\mathbf{x}$ such that $Q\mathbf{x} = \mathbf{x}$. Suppose there exist vectors $\mathbf{x}$ and $\mathbf{y}$, such that $Q\mathbf{x} = \mathbf{x}$, $Q\mathbf{y} = \mathbf{y}$. Then $\mathbf{x} = c\mathbf{y}$, where c is a scalar. In other words, the eigenvector having eigenvalue one is unique up to a scaling factor.*

7.5.2 Convergence Proof

Theorem 7.5.7 *If M is a positive Markov matrix, the iteration (7.4.2) converges to a unique vector $\mathbf{p}^\infty$, for any initial probability vector $\mathbf{p}^0$.*

Proof: Let $\mathbf{x}_l$ be an eigenvector of M, corresponding to the eigenvalue λ_l. Suppose that M has a complete set of eigenvectors, in other words, we can represent $\mathbf{p}^0$ as

$$\mathbf{p}^0 = \sum_l c_l \mathbf{x}_l \tag{7.5.8}$$

for some scalars c_l. (We do not have to make this assumption, but it simplifies the proof). Suppose also that we order these eigenvectors so that $|\lambda_1| > |\lambda_2| \geq ...$ so that $\mathbf{x}_1$ corresponds to the unique eigenvector [1] with $\lambda_1 = 1$. Then

$$(M)^k \mathbf{p}^0 = c_1 \mathbf{x}_1 + \sum_{l=2}^{R} c_l (\lambda_l)^k \mathbf{x}_l \ . \tag{7.5.9}$$

From Theorem (7.5.5), we have that $|\lambda_l| < 1$ for all $l > 1$, so that

$$\lim_{k\to\infty} (M)^k \mathbf{p}^0 = c_1 \mathbf{x}_1 \tag{7.5.10}$$

for any $\mathbf{p}^0$. $c_1\mathbf{x}_1$ cannot be identically zero, since $\mathbf{p}^0$ is a probability vector, and hence $\mathbf{p}^\infty$ is a probability vector. Hence $c_1 \neq 0$ and $\mathbf{x}_1$ cannot be the zero vector. Uniqueness follows since if we start the iteration with another probability vector

$$\mathbf{q}^0 = \sum_l b_l \mathbf{x}_l \tag{7.5.11}$$

for some coefficients b_l, then

$$\mathbf{q}^\infty = \lim_{k\to\infty} (M)^k \mathbf{q}^0 = b_1 \mathbf{x}_1 \ . \tag{7.5.12}$$

But, for given $\mathbf{x}_1$, since $\mathbf{q}^\infty, \mathbf{p}^\infty$ are probability vectors, we have $b_1 = c_1$.

[1] To make $\mathbf{x}_1$ unique, we can arbitrarily fix the scaling factor mentioned in Remark 7.5.6. One way to do this would be to require $\sum (\mathbf{x}_1)_i^2 = 1$

Remark 7.5.8 *The speed of convergence of algorithm (7.4.2) is determined by the size of the second largest eigenvalue λ_2 of the Google matrix M, since $|\lambda_l| < |\lambda_2|$, $l > 2$. One can show that the size of $|\lambda_2| \simeq \alpha$ (see Golub and van Loan,* Matrix Computations*, 1996). For example, if $\alpha = .85$, then $|\lambda_2|^{114} = (.85)^{114} \simeq 10^{-8}$. This means that* 114 *iterations of algorithm (7.4.2) for any starting vector should give reasonably accurate estimates for $\mathbf{p}^\infty$.*

The Web is estimated to contain billions of pages. Using algorithm (7.4.2) is estimated to require several days of computation. Note that the ranking vector $\mathbf{p}^\infty$ can be computed and stored independent of the Google query. When a user enters a keyword search, a subset of Web pages containing the keywords is returned. Then, these pages are ranked according to the precomputed vector $\mathbf{p}^\infty$.

Why not just choose α very small, since this would speed up the computation? Consider the case $\alpha = 0$. Then it's easy to see that $[\mathbf{p}^\infty]_i = 1/R$, i.e. all pages are ranked equally. The smaller the value of α, the faster the convergence of algorithm (7.4.2). However small values of α result in less significant ranking information.

7.6 Practicalities

If we assume that $\mathbf{p}^n$ is a probability vector then

$$\begin{aligned} \frac{\mathbf{e}\mathbf{e}^t}{R}\mathbf{p}^n &= \frac{1}{R}\mathbf{e}(\mathbf{e}^t\mathbf{p}^n) \\ &= \frac{\mathbf{e}}{R} \end{aligned} \tag{7.6.1}$$

so that

$$M\mathbf{p}^n = \alpha P'\mathbf{p}^n + (1-\alpha)\frac{\mathbf{e}}{R} \tag{7.6.2}$$

and that

$$\begin{aligned} P'\mathbf{p}^n &= (P + \frac{\mathbf{e}\cdot\mathbf{d}^t}{R})\mathbf{p}^n \\ &= P\mathbf{p}^n + \mathbf{e}\left(\frac{\mathbf{d}^t\mathbf{p}^n}{R}\right) \ . \end{aligned} \tag{7.6.3}$$

Putting these two steps together gives

$$M\mathbf{p}^n = \alpha\left(P\mathbf{p}^n + \mathbf{e}\left(\frac{\mathbf{d}^t\mathbf{p}^n}{R}\right)\right) + (1-\alpha)\frac{\mathbf{e}}{R} \ . \tag{7.6.4}$$

Typically, P is quite sparse (i.e., has few non-zero elements), so that even though M is dense, we can perform the matrix-vector multiply $M\mathbf{p}^n$ in $O(R)$ operations.

7.7 Summary

Given the positive Markov matrix M which represents the structure of the Web, the pages can be ranked using the components of the vector $\mathbf{p}^k$ computed via algorithm (7.7.1) below.

Page Rank Algorithm

$$
\begin{array}{l}
\mathbf{p}^0 = \mathbf{e}/R \\
\text{For } k = 1, ..., \text{ until converged} \\
\qquad \mathbf{p}^k = M\mathbf{p}^{k-1} \\
\qquad \text{If } \max_i |[\mathbf{p}^k]_i - [\mathbf{p}^{k-1}]_i| < tol \text{ then quit} \\
\text{EndFor}
\end{array}
\tag{7.7.1}
$$

7.7.1 Some Other Interesting Points

Instead of defining M as

$$M = \alpha P' + (1-\alpha)\mathbf{e}\mathbf{e}^t/R \,, \tag{7.7.2}$$

Google actually uses

$$M = \alpha P' + (1-\alpha)\mathbf{v}\mathbf{e}^t \tag{7.7.3}$$

where $\mathbf{v}$ is a probability vector, which can be tuned by allowing teleportation to particular pages. That is, Google can intervene to adjust page ranks up or down for commercial considerations.

Google also reports that the Page Rank is updated only every few weeks, due to the cost of computation. There is considerable effort directed to speeding up the convergence of the iterative algorithm (7.7.1). In mathematical terms, algorithm (7.7.1) is known as the power method for finding the eigenvector of M corresponding to the largest eigenvalue.

Chapter 8

Least Squares Problems

8.1 Finding Parameters

Suppose you took a beautiful picture of the Montreal city-scape, but the film was partially over-exposed before you developed it. The resulting photos might contain a bright side and a dark side, or a bright blob in the middle, as shown in figure 8.1.

(a) Original (b) Over-exposed

Figure 8.1: Original and Over-exposed pictures of Montreal

However, not all is lost – we can remove some of the shading artifacts[1]. However, we need to estimate the shading effect accurately first.

Suppose we know the type of shading effects, but we do not know how strong the effect is. Figure 8.2 shows a model in which the observed photo can be decomposed into 3 parts: the true photo, a centred exposure artifact, and a left-right exposure artifact.

[1]In image processing, an "artifact" is an unwanted feature in an image, usually the result of some sort of problem or inaccuracy.

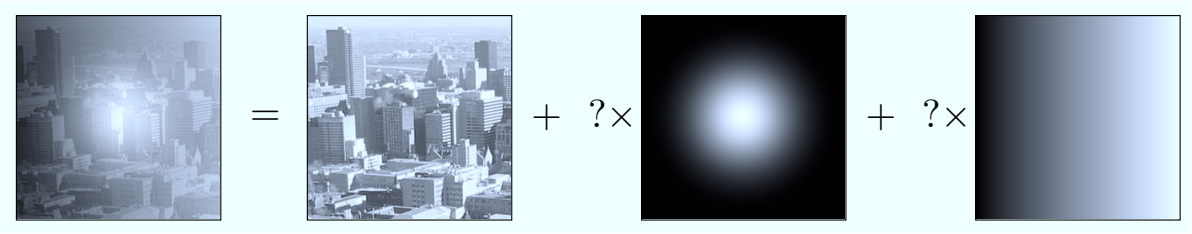


Figure 8.2: Decomposition of over-exposed photo. Although we know the form of the exposure artifacts, the question-marks indicate that we do not know how strong each artifact is.

8.2 Least Squares Fitting

Figure 8.2 illustrates a particular model for representing the observed picture. The form of the exposure artifact components is assumed to be known – it is the *amount* of each component that is unknown. We will estimate how much of each component is present by a process known as *least squares fitting*. In particular, we will model the observed photo as a linear combination of 4 components, illustrated in Figure 8.3.

Figure 8.3: Decomposition of over-exposed photo into 4 components.

To go further with this problem, we will represent an "image object" as a single column vector. The original image is stored as a table of 256×256 numbers. To make it into a column vector, we simply read off all the numbers in the table, going down one column at a time, as in Figure 8.4.

Using the same method, we can convert each of the components in Figure 8.3 to vectors, producing 4 column vectors. Then, Figure 8.3 can be written as a vector equation,

$$y = a_1\beta_1 + a_2\beta_2 + a_3\beta_3 + \epsilon \tag{8.2.1}$$

$$y = \mathbf{A}\beta + \epsilon\ , \tag{8.2.2}$$

where $\mathbf{A} = [a_1|a_2|a_3]$. In this context, the image itself (represented by ϵ) is anything that does not fit the model. In many other least squares fitting problems, we model everything except random noise; in that case, ϵ would represent noise. But in our scenario, the image is what is left after we remove all the exposure artifacts.

The question remains, how do we choose the parameters β_1, β_2 and β_3 to best match the artifacts observed in the image? What we aim to do is find the parameter values

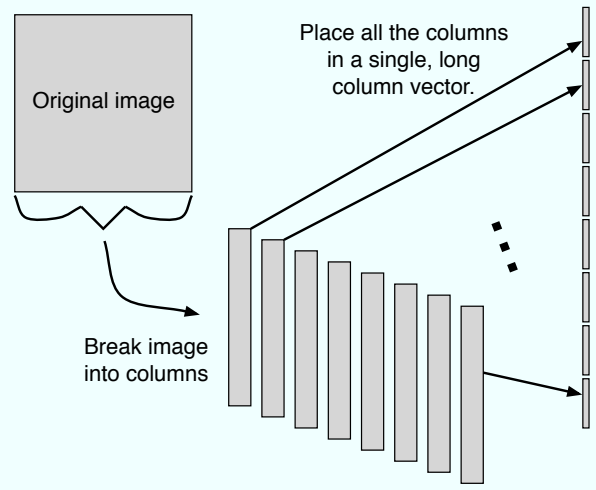


Figure 8.4: Representing an image as a column vector

that minimize some measure of the *residual*. The residual, r, is the vector of differences between the observed data (image), and the model for a given set of parameter values,

$$r = \begin{bmatrix} r_1 \\ \vdots \\ r_m \end{bmatrix} = y - \mathbf{A}\beta \tag{8.2.3}$$

where m is the number of pixels (65,536 for our 256×256 image). Notice that in this case the desired image (ϵ) is actually r, the residual (compare (8.2.3) to (8.2.2)). This is not normally the case, but happens in our scenario. The reason for this is that we are modeling the artifacts so that we can remove them. In essence, we are modeling the part we don't want, so the part we **do** want (the uncorrupted image data) is left in ϵ.

A few words about the dimensions of these matrix expressions. If one were to draw a picture of the matrix expression on the right-hand-side of equation (8.2.3), it might look like Figure 8.5. The long, tall look of this matrix expression is typical of least squares fitting problems. Each row in the matrix expression represents one observation of the system (one opportunity to observe the interplay between the parameters β and the output y). Usually there are many more observations (rows) than there are parameters (columns in $\mathbf{A}$). This makes $\mathbf{A}$ tall and narrow. Since we have so many more observations than fitting parameters, it is highly unlikely that we will be able to find parameter values that fit all the observations exactly. This type of problem is called *overdetermined*. Instead of trying to fit all the observations exactly, we seek parameter values that approximate the observations as "closely as possible". What we mean by that is discussed in the next section.

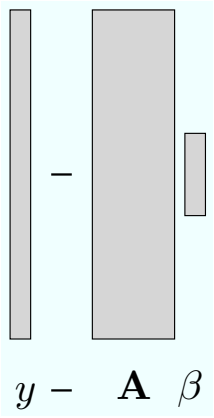


Figure 8.5: Illustration of the matrix expression for calculating the residual vector.

8.2.1 Total squared error

A useful summary of the errors associated with the given choice of β is the square of the (Euclidean) length of r. We will call this measure the **total squared error**, and denote it

$$E(\beta) = \sum_{i=1}^{m} r_i^2 \ . \tag{8.2.4}$$

The total squared error can also be written $E(\beta) = \|r\|^2 = \|y - \mathbf{A}\beta\|^2$. Furthermore, the expression $r^t r$ is the same as $\|r\|^2$ (r^t is the transpose of r). Hence, $E(\beta)$ can also be written

$$E(\beta) = (y - \mathbf{A}\beta)^t (y - \mathbf{A}\beta) \ . \tag{8.2.5}$$

Figure 8.6 shows the results when choosing particular parameter values. Figure 8.6(a) is the residual image corresponding to $\beta = [163.4\ 1.5\ 0.5]^t$, resulting in a total squared error of 29,352, while Figure 8.6(b) corresponds to $\beta = [163.4\ 1.9\ 0.7]^t$, resulting in a total squared error of 14,261. Figure 8.6(b) has a smaller total squared error than (a). It is not surprising that (b) also looks like it has less exposure artifact than (a).

Can we find parameter values β_1, β_2 and β_3 that make a better linear fit to the data than the values tried in Figure 8.6? Here "better" means smaller total squared error.

The answer is 'Yes'. In fact, we are going to discuss how to compute $\bar{\beta}$ that gives the *smallest possible* total squared error. That is, $E(\bar{\beta}) \leq E(\beta)$ for all possible choices of β. The parameters in $\bar{\beta}$ are called the least squares fit parameters for this data. For the example in Figure 8.2, the least squares fit parameters are $\bar{\beta} = [163.4\ 2.047\ 0.7127]^t$ and give a total squared error of $E(\bar{\beta}) = 13,005$.

8.2.2 The Normal Equations

There is a simple method to finding the optimal least squares fit parameters for a linear least squares problem. If the model is a linear function of the parameters (i.e. if the

(a) $E(163.4, 1.5, 0.5) = 29,352$ (b) $E(163.4, 1.9, 0.7) = 14,261$

Figure 8.6: For (a), $\beta = [163.4\ 1.5\ 0.5]^t$ and $E(\beta) = 29,352$. For (b), $\beta = [163.4\ 1.9\ 0.7]^t$ and $E(\beta) = 14,261$.

model can be written as a matrix times the vector of parameters values), then this method is guaranteed to find the least squares solution.

To find the least squares solution, we need to solve the **normal equations**. This section describes what the normal equations are, and where they come from. You will see that it is nothing more than first-year calculus.

Let us illustrate our point with the problem

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \\ a_{51} & a_{52} & a_{53} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} \tag{8.2.6}$$

or in matrix form $Ax = b$ where

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}$$

with A the 5×3 matrix in equation (8.2.6). In general, it will not be possible to find x such that $Ax = b$, so we search for x which minimizes the residual

$$\min_x \|r\|_2 = \|b - Ax\|_2 \tag{8.2.7}$$

In other words, we seek the *least squares* solution to $Ax = b$. Note that in general, for

a least squares problem, we have

$$\begin{aligned} Ax &= b \\ & A = m \times n \\ & x = n \times 1 \\ & b = m \times 1 \\ & m > n (\text{overdetermined}) \end{aligned}$$

Writing out $\|r\|_2^2 = r^t r$ gives

$$\begin{aligned} r_i &= \sum_j a_{ij} x_j - b_i \\ r^t r &= \sum_i r_i r_i \\ &= \sum_i \left[\sum_j a_{ij} x_j - b_i \right] \left[\sum_k a_{ik} x_k - b_i \right] \end{aligned} \tag{8.2.8}$$

In order to minimize (8.2.8), we differentiate with respect to x_l, and set to zero. The result, in matrix notation, is

$$A^t A x = A^t b. \tag{8.2.9}$$

So, x, which is the solution to the minimization problem (8.2.7), is given by the solution to equation (8.2.9).

Equations (8.2.9) are referred to as the *normal equations*. If the columns of A are linearly independent (A has full column rank), then $A^t A$ is nonsingular. Since $A^t A$ is a symmetric matrix, we can save half the work of a usual factor with gaussian elimination (can you write out the pseudo code for this?). So, the work of factoring the normal equations is

$$\text{work} = \frac{n^3}{3} + O(n^2). \tag{8.2.10}$$

But, consider the case when $m = n$, and suppose A is symmetric, positive definite, then we have that

$$\begin{aligned} \lambda_{max}(A^t A) &= \lambda_{max}(A)^2 \\ \lambda_{min}(A^t A) &= \lambda_{min}(A)^2 \end{aligned}$$

where

$$\begin{aligned} \lambda_{max} &= \text{ maximum eigenvalue} \\ \lambda_{min} &= \text{ minimum eigenvalue} \end{aligned}$$

so that $\kappa(A^t A)_2 = \kappa(A)_2^2$, which makes the problem much worse conditioned. So, solving the normal equations may be poorly conditioned in general.

8.2.3 The QR decomposition

Suppose we have the $m \times m$ matrix Q which is the identity matrix, except for entries $Q_{ij}, Q_{ji}, Q_{ii}, Q_{jj}$

$$\begin{aligned} Q &= I; \text{ except for the entries} \\ & Q_{ij} = s \\ & Q_{ji} = -s \\ & Q_{ii} = c \\ & Q_{jj} = c \end{aligned} \tag{8.2.11}$$

Now, imagine multiplying Q times an m length vector

$$Qx = Q \begin{bmatrix} x_1 \\ \vdots \\ \vdots \\ x_i \\ \vdots \\ \vdots \\ x_j \\ \vdots \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} x_1 \\ \vdots \\ \vdots \\ cx_i + sx_j \\ \vdots \\ \vdots \\ -sx_i + cx_j \\ \vdots \\ \vdots \\ x_m \end{bmatrix} \tag{8.2.12}$$

Note that only rows i, j of x have changed. We can make $[Qx]_j = 0$ if we choose

$$c = \frac{x_i}{\sqrt{x_i^2 + x_j^2}} \quad ; \quad s = \frac{x_j}{\sqrt{x_i^2 + x_j^2}} \tag{8.2.13}$$

Because of the form of c, s, we can write

$$c = \cos\theta \quad ; \quad s = \sin\theta \tag{8.2.14}$$

for some θ. Assuming c, s given by equation (8.2.14), then it is easy to see that

$$Q^t Q = QQ^t = I \rightarrow Q^{-1} = Q^t \tag{8.2.15}$$

so that Q is an *orthogonal* matrix.

Note that, if r is any m length vector

$$\begin{aligned} \|Q^t r\|_2^2 &= (Q^t r)^t (Q^t r) \\ &= r^t Q Q^t r \\ &= r^t r \\ &= \|r\|_2^2 \end{aligned} \tag{8.2.16}$$

so that multiplication of any vector by Q^t preserves its length, that is,

$$\|Q^t x\|_2 = \|x\|_2 \tag{8.2.17}$$

Now, let's multiply the overdetermined system $Ax = b$ by a sequence of orthogonal matrices $Q^{(1)}, Q^{(2)}, ..., Q^{(k)}$. Each of the $Q^{(p)}$ is selected to an annihilate a given entry in A, and eventually, we reduce A to upper triangular form. Note that Q of the form (8.2.11) only affects two rows of A. For example, if we let A be the 5×3 system (8.2.6), then we can select entries in rows $4, 5$ of $Q^{(1)}$ so that

$$\begin{aligned} A &= \begin{bmatrix} X & X & X \\ X & X & X \\ X & X & X \\ X & X & X \\ X & X & X \end{bmatrix} \quad ; \quad X = \text{ nonzero entry} \\ Q^{(1)}A &= \begin{bmatrix} X & X & X \\ X & X & X \\ X & X & X \\ X & X & X \\ 0 & X & X \end{bmatrix} \end{aligned} \tag{8.2.18}$$

Next, we select entries in $Q^{(2)}$ in rows $3, 4$, so that a_{41} is set to zero. Note that the zero in position a_{51} is unchanged

$$Q^{(2)}Q^{(1)}A = \begin{bmatrix} X & X & X \\ X & X & X \\ X & X & X \\ 0 & X & X \\ 0 & X & X \end{bmatrix} \tag{8.2.19}$$

Continuing in this way we obtain

$$Q^{(k)}...Q^{(2)}Q^{(1)}A = \begin{bmatrix} X & X & X \\ 0 & X & X \\ 0 & 0 & X \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{8.2.20}$$

Let

$$R = \begin{bmatrix} X & X & X \\ 0 & X & X \\ 0 & 0 & X \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{8.2.21}$$

then we can write

$$Q^{(k)}...Q^{(2)}Q^{(1)}A = R \tag{8.2.22}$$

Let

$$Q^{(k)}...Q^{(2)}Q^{(1)} = Q^t \tag{8.2.23}$$

so that

$$Q^tQ = Q^{(k)}...Q^{(1)}(Q^{(1)})^t...(Q^{(k)})^t = I \tag{8.2.24}$$

which means that Q is orthogonal.

From equations (8.2.22-8.2.23)

$$\begin{aligned} Q^tA &= R \\ A &= Q^{-t}R = QR \end{aligned} \tag{8.2.25}$$

So, equation (8.2.25) implies we have constructed the QR decomposition of A, that is, $A = QR$.

Back to the least squares problem (8.2.6). From equation (8.2.17), we have

$$\|Q^tr\|_2 = \|r\|_2 \tag{8.2.26}$$

So, the least squares problem can be written

$$\min_x \|Ax - b\|_2 = \min_x \|Q^tAx - Q^tb\|_2 \tag{8.2.27}$$

which means that x which minimizes $\|Ax - b\|_2$ also minimizes $\|Q^tAx - Q^tb\|_2$. If $A = QR$, then the least squares problem becomes

$$\begin{aligned} Q^tA &= \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ 0 & r_{22} & r_{23} \\ 0 & 0 & r_{33} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ Q^tb &= \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{bmatrix} \end{aligned} \tag{8.2.28}$$

So equation (8.2.27) becomes

$$\min_x \left\| \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ 0 & r_{22} & r_{23} \\ 0 & 0 & r_{33} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} - \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{bmatrix} \right\|_2 \tag{8.2.29}$$

Thus, no matter what vector x we choose,

$$\|Ax - b\|_2 = \|Q^t(Ax - b)\|_2 \quad \geq \quad \sqrt{c_4^2 + c_5^2}$$

The optimum choice for x is then

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} \\ 0 & r_{22} & r_{23} \\ 0 & 0 & r_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} \tag{8.2.30}$$

since this choice of x makes

$$\|Ax - b\|_2 = \sqrt{c_4^2 + c_5^2}$$

which can't be improved upon. So, the solution to the least squares problem is x which satisfies

$$\begin{aligned} \widehat{R}x &= \widehat{Q^t b} \\ \widehat{R} &= \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ 0 & r_{22} & r_{23} \\ 0 & 0 & r_{33} \end{bmatrix} \\ \widehat{Q^t b} &= \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} \end{aligned}$$

or $x = (\widehat{R})^{-1}\widehat{Q^t b}$, assuming that the columns of A are linearly independent.

A *smart* implementation of this QR algorithm requires about (assuming $m \geq n$) $3mn^2 - 2n^3$ flops. If $m = n$, this is about n^3 flops, which can be compared to $n^3/3$ for the normal equation approach (although we have neglected the cost of forming $A^t A$). Thus, a QR decomposition in the square matrix case is about three times more expensive than factoring the normal equations, but is much more stable since it avoids squaring the condition number.

8.3 Exercises for Least Squares Problems

1. Construct a 4×4 orthogonal matrix W such that

$$W \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} \alpha \\ 0 \\ 0 \\ x_4 \end{bmatrix}$$

where $\alpha = \sqrt{x_1^2 + x_2^2 + x_3^2}$.

Appendix A

Local Error of an ODE Method

Suppose we want to solve

$$\frac{dy}{dt} = f(t, y) \ . \tag{A.0.1}$$

Given an algorithm for advancing the time, e.g. forward Euler,

$$y_{n+1} = y_n + hf(t_n, y_n) \tag{A.0.2}$$

where y_n is an approximation to $y(t_n)$, then the local truncation error is determined using the following approach. Write equation (A.0.2) in the form

$$\begin{aligned} g(y_n, y_{n+1}, t_n, t_{n+1}, ...) &= y_{n+1} - [y_n + hf(t_n, y_n)] \\ &= 0 \end{aligned} \tag{A.0.3}$$

Then, we replace y_n, y_{n+1} (the approximate solutions) by $y(t_n), y(t_{n+1})$ (the exact solutions) in equation (A.0.3)

$$g(y(t_n), y_(t_{n+1}), t_n, t_{n+1}, ...) = y(t_{n+1}) - [y(t_n) + hf(t_n, y(t_n))] \tag{A.0.4}$$

Of course, $g(...)$ in equation (A.0.4) will not be zero, since equation (A.0.2) is only an approximation to the exact solution. The *size* of $g(...)$ is the *local truncation error*.

From Taylor series, expanding about t_n, we have

$$y(t_{n+1}) = y(t_n) + y'(t_n)h + O(h^2) \tag{A.0.5}$$

where $h = t_{n+1} - t_n$. Substituting equation (A.0.5) into equation (A.0.4) gives

$$g(...) = h[y'(t_n) - f(t_n, y(t_n))] + O(h^2) \tag{A.0.6}$$

Of course, from equation (A.0.1) we have

$$y'(t_n) = f(t_n, y(t_n)) \tag{A.0.7}$$

so that equation (A.0.6) becomes

$$g(y(t_n), y(t_{n+1}), ...) = O(h^2) \tag{A.0.8}$$

so that the Local Error of Forward Euler is

$$\text{Local Error}_{F.E.} = O(h^2) \tag{A.0.9}$$

We can generalize this definition to any numerical method for solving ODEs. First, write down the algorithm in the form

$$\begin{aligned} g(y_n, y_{n+1}, ...) &= y_{n+1} - [...] \\ &= 0 \end{aligned} \tag{A.0.10}$$

Then, replace y_n by $y(t_n)$, etc.

$$g(y(t_n), y(t_{n+1}), ...) = y(t_{n+1}) - [...] \tag{A.0.11}$$

Now, start expanding the terms in equation (A.0.11) in a Taylor series, and use equation (A.0.1) to simplify. You will end up with

$$g(y(t_n), y(t_{n+1}), ...) = O(h^p) \tag{A.0.12}$$

which is the local truncation error.

The only possibly tricky bit is that you have to figure out the correct point to use as the base for your Taylor series expansion (i.e. $y(t_n)$, or $y(t_{n+1})$, etc.) You want to use equation (A.0.1) to simplify, so this should guide you,

Appendix B

Complex Numbers and Roots of Unity

B.1 Review of Complex Numbers

In order to proceed we first need to do a quick review of complex numbers since this is the arithmetic domain for these discrete transforms. Denote the set of complex numbers by C. Thus each $z \in C$ can be identified as a pair of real numbers, $z = (a, b)$. If $z = (a, b)$ and $w = (c, d)$ then addition and multiplication is defined for C by

$$z + w = (a + c, b + d) \text{ and } z \cdot w = (ac - bd, ad + cb).$$

Classically, one identifies $(a, 0)$ and $(b, 0)$ as a and b, respectively. If we set $i = (0, 1)$ and use the above identification, then we have the classical representation of a complex number as

$$z = (a, b) = a + bi.$$

Notice that $i^2 = (0, 1) \cdot (0, 1) = (-1, 0) = -1$. In Python, you can enter a complex number using the imaginary `j`, as in `-3 + 2.j`.

If $z = a + bi$ then a is the real part of z, typically denoted by $a = Real(z)$, b is the imaginary part of z, typically denoted by $b = Imag(z)$. The number $\bar{z} = a - bi$ is the *conjugate* of z and the quantity $|z| = \sqrt{a^2 + b^2}$ is called the *modulus* or *norm* of z. Some other useful properties of complex numbers are:

- $z + \bar{z} = 2 \text{ Real}(z)$
- $z \cdot \bar{z} = |z|^2 = a^2 + b^2$.

We can form N-vectors (column vectors) of complex numbers, $u = (u_1, u_2, \ldots, u_N)^t \in C^N$, and define some of the typical arithmetic vector operations, such as

- multiplying u by a number $p \in C$,
- adding two complex vectors, or forming a linear combination, i.e. for $u, v \in C^N$ and $e, f \in C$, $eu + fv$ is a complex vector, and

- calculate the inner product of two complex vectors u and v as

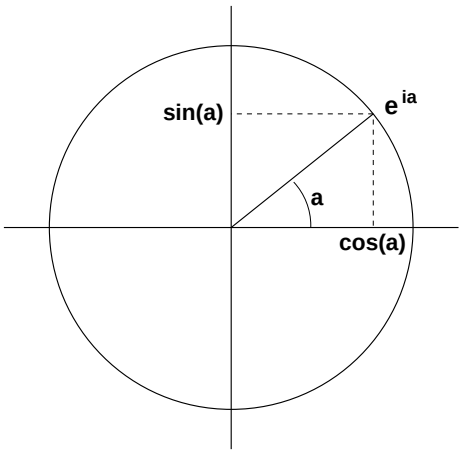
$$(u, v) = \sum_{j=1}^{N} u_j \bar{v}_j \ . \tag{B.1.1}$$

We denote the set of complex N vectors by C^N.

B.1.1 Roots of unity

In the study of the discrete Fourier transform, the exponential function with a pure imaginary argument plays a major role. This is defined as

$$e^{i\theta} \ = \ (\cos(\theta), \sin(\theta)) \ = \ \cos(\theta) + i \sin(\theta) \tag{B.1.2}$$



$e^{ia} = \cos(a) + i \sin(a)$

The figure above (B.1.2) shows the unit circle[1] in the complex plane and the complex number e^{ia}, where a is a real number. It shows how a can be identified with the angle at the origin. From this, it can be seen that e^{ia} is a periodic function of a with period 2π. That is, $e^{i(a+2\pi k)} = e^{ia}$ for *any* integer k.

This exponential function retains the usual property of exponentials, namely the multiplication of two exponentials can be achieved by adding their exponents

$$e^{ia_1} \, e^{ia_2} = e^{i(a_1+a_2)} \ . \tag{B.1.3}$$

It can be seen from equation (B.1.2) that the conjugate of $e^{i\theta}$ is

$$e^{-i\theta} = \cos(\theta) \ - \ i \ \sin(\theta) \ . \tag{B.1.4}$$

Let W be the complex number

$$W = e^{2\pi i/N} \tag{B.1.5}$$

[1] circle of radius 1

for some given integer N. Since $e^{2\pi i} = 1 + 0i$, we have

$$W^N = 1. \tag{B.1.6}$$

Any complex number v such that $v^N = 1$ is called an *Nth root of unity*. For example, $e^{\pi i/4}$ is an 8th root of unity, and so is $e^{-\pi i/4}$. The "square root of -1", $i = e^{\pi i/2}$, is a 4th root of unity. According to (B.1.5), W is an Nth root of unity, and so is $\bar{W} = e^{-2\pi i/N}$ (the conjugate of W).

Example B.1.1 *One can see roots of unity correspond to equally-spaced points on the unit circle in the complex plane. For example, the number $e^{i\pi/4}$ is an 8th root of unity. The corresponding angle is $\frac{\pi}{4}$, which is the same as 45°. Taking steps of 45° around the unit circle, we find the numbers $e^{i\pi/2}$ (which is equal to i; try plugging $a = \frac{\pi}{2}$ into (B.1.2)), $e^{i3\pi/4}$, ..., $e^{i7\pi/4}$, all of which are 8th roots of unity. Figure B.1 plots these values on the unit circle in the complex plane. Notice that in component form we get*

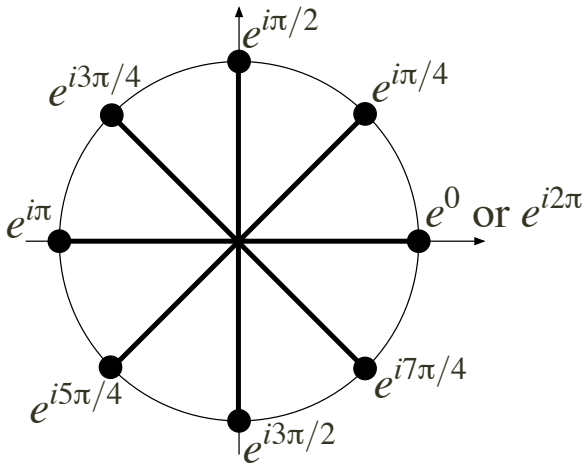


Figure B.1: The 8th roots of unity plotted on the unit circle in the complex plane.

$$\begin{array}{rclrcl} e^{i\pi/4} &=& \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}i, & e^{i\pi/2} &=& i, \\ e^{3i\pi/4} &=& -\frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}i, & e^{i\pi} &=& -1, \\ e^{5i\pi/4} &=& -\frac{1}{\sqrt{2}} - \frac{1}{\sqrt{2}}i, & e^{3i\pi/2} &=& -i, \\ e^{7i\pi/4} &=& \frac{1}{\sqrt{2}} - \frac{1}{\sqrt{2}}i, & e^{2i\pi} &=& 1. \end{array} \tag{B.1.7}$$

Exercises

1. Show that $W^k = e^{2\pi k\, i/N}$ is an Nth root of unity for every integer k.

2. Despite the result in part B.1.1, the set of all Nth roots of unity is finite. Why? *Hint*: Find two integers k_1 and k_2 such that
$$e^{2\pi k_1\, i/N} = e^{2\pi k_2\, i/N}.$$

3. How many different complex numbers are Nth roots of unity?

B.1.2 Orthogonality property

For any $z \in C$, we have the following formula for the sum of a finite geometric series:

$$1 + z + z^2 + \cdots + z^{N-1} \;=\; \frac{z^N - 1}{z - 1}. \tag{B.1.8}$$

Of course (B.1.8) is only valid for $z \neq 1$. When $z = 1$ we have

$$1 + z + z^2 + \cdots + z^{N-1} \;=\; N. \tag{B.1.9}$$

This leads to an important mathematical basis for the usefulness of Fourier transforms.

Let W be the Nth root of unity defined by (B.1.5) and consider the vectors $W(k) \in C^N$, $0 \leq k \leq N-1$, defined by

$$W(k) = (1, W^k, W^{2k}, \ldots, W^{(N-1)k})^t \;. \tag{B.1.10}$$

That is, the jth component, $W(k)_j$, is W^k raised to the power $j-1$.

The complex inner product of the two vectors is given by

$$(W(k), W(j)) = \sum_{n=0}^{N-1} W(k)_n \overline{W(j)}_n. \tag{B.1.11}$$

If $k \neq j$, then

$$\begin{aligned} (W(k), W(j)) \;&=\; 0 \text{ if } k \neq j \\ &=\; N \text{ if } k = j \end{aligned} \tag{B.1.12}$$

To see this, notice that

$$(W(k), W(j)) = \sum_{n=0}^{N-1} W^{(k-j)n} \;. \tag{B.1.13}$$

On the other hand when $k = j$ then $(W(k), W(j)) = N$ by formula (B.1.9). Otherwise, by formula (B.1.8) we have

$$(W(k), W(j)) = \frac{W^{(k-j)N} - 1}{W^{k-j} - 1} = 0$$

because the numerator is zero and the denominator is nonzero. In other words, $W(k)$ and $W(j)$ are 'orthogonal' complex vectors. This is the *Basic Lemma of Fourier Transforms.*

Appendix C

Computerized Tomography (CT)

CT scanners are an example application of the FFT algorithm. In medical applications, a patient is placed in a CT scanner, which consists of an array of x-ray sources (beams) and an array of detectors. The beams are oriented at a specific angle, the data is recorded (which is the projection of the X-ray beams). The beam angle is then changed to a new angle, and the data is recorded again. These measurements are repeated for many different angles. The problem now becomes one of image reconstruction. The image must be reconstructed from this set of projections. This data collection operation is illustrated in Figure C.1.

A typical hospital setup is shown in Figure C.2. A reconstructed image is shown in Figure C.3.

Figure C.1: Data collection for a CT scan.

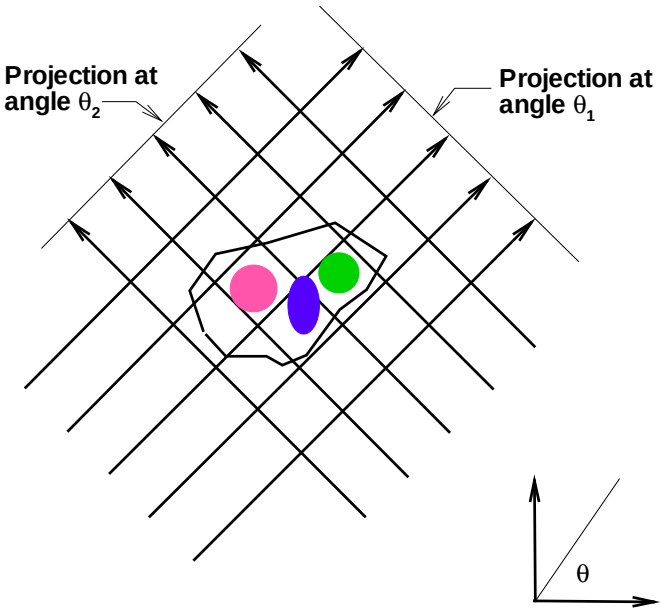
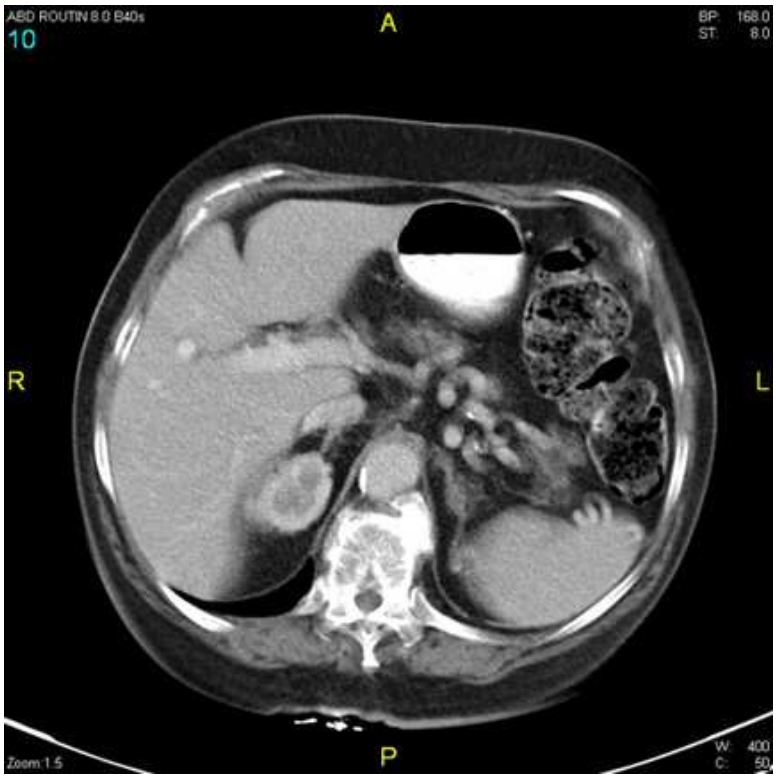


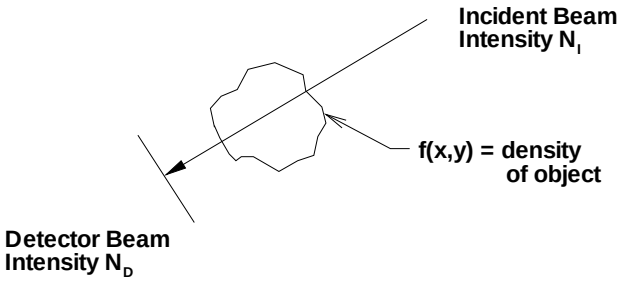
Figure C.2: A CT scanner

Figure C.3: A reconstructed image of an patient's abdomen.



If we look at a two dimensional cross-section, and a single beam, we have the situation shown in Figure C.4.

Figure C.4: A single x-ray beam, with incident intensity N_i and detected intensity N_d. The tomographed object has a density function $f(x, y)$.



If the density of the tissue is $f(x, y)$, then the beam intensity N will be attenuated

$$\frac{dN}{ds} = -f(x, y)N \tag{C.0.1}$$

where s is the distance along the beam. If N_i and N_d are the incident and detected beam intensities, then from equation (C.0.1) we obtain

$$N_d = N_i e^{\left(-\int_{ray} f(x,y)\ ds\right)} \tag{C.0.2}$$

or

$$\int_{ray} f(x, y)\ ds = \log\left(\frac{N_i}{N_d}\right) \tag{C.0.3}$$

so that what we measure at the detector is the *projection* of the image $f(x, y)$ at various projection angles. Given these projections, we want to reconstruct $f(x, y)$. The set of projections is also called the *Radon transform*. Hence the operation of reconstructing the image is the inverse Radon transform.

C.1 Continuous Fourier Transforms

Although we will be using DFTs in all our computations, it will be easier to derive the results using continuous Fourier Transforms. If we let the number of sample points become infinite, and the spacing between sample points go to zero, we get the continuous Fourier transform

$$\begin{aligned} F(k) &= \int_{-\infty}^{\infty} e^{-i\ 2\pi kt} f(t)\ dt \\ f(t) &= \int_{-\infty}^{\infty} e^{i\ 2\pi kt} F(k)\ dk\ , \end{aligned} \tag{C.1.1}$$

and a two dimensional version

$$\begin{aligned} F(k_x, k_y) &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y) e^{-i\ 2\pi(k_x x + k_y y)}\ dx\ dy \\ f(x,y) &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{i\ 2\pi(k_x x + k_y y)} F(k_x, k_y)\ dk_x\ dk_y\ . \end{aligned} \tag{C.1.2}$$

C.2 The Slice Theorem

Consider the 2-d continuous Fourier transform of the image $f(x,y)$

$$F(k_x, k_y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y) e^{-i\ 2\pi(k_x x + k_y y)}\ dx\ dy \tag{C.2.1}$$

Consider

$$\begin{aligned} F(k_x, 0) &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y) e^{-i\ 2\pi(k_x x + 0)}\ dx\ dy \\ &= \int_{-\infty}^{\infty}\left(\int_{-\infty}^{\infty} f(x,y)\ dy\right) e^{-i\ 2\pi k_x x}\ dx \\ &= \int_{-\infty}^{\infty} p(0,x) e^{-i\ 2\pi k_x x}\ dx \end{aligned} \tag{C.2.2}$$

where $p(\theta = 0, x)$ is the projection of $f(x,y)$ at projection angle $\theta = 0$ to the x-axis. Since there is nothing sacred about our choice of co-ordinate system, we can repeat the above argument for any angle θ, and we get the *Fourier Slice Theorem*

Theorem C.2.1 (Fourier Slice Theorem) *The Fourier transform of a parallel projection of an image $f(x,y)$ taken at an angle θ, (we denote the FT of the projection by $P(\theta, t)$), gives a slice of the 2-d Fourier transform $F(k_x, k_y) = FT(f(x,y))$ at an angle θ with the k_x axis.*

This Theorem is illustrated in Figure C.5.

Now, since the measured projections are at discrete locations (there are only a finite number of detectors), we carry out a DFT on the projection data at each projection angle. Consequently, because of the way the slices build up data in the frequency domain, we know the discrete values of the 2-d DFT of the image at discrete locations along circular arcs in the (k_x, k_y) plane, as shown in Figure C.6. In order to reconstruct the image $f(x,y)$, we have to carry out an inverse 2-d DFT, i.e.

$$f(x,y) = IFT(F(k_x, k_y))$$

In order to do this efficiently, we will, of course, use an FFT algorithm. This requires knowledge of the the discrete values of $F(k_x, k_y)$ at equally spaced Cartesian grid locations in the (k_x, k_y) plane. So, we will have to interpolate the slice data onto this grid, before carrying out the 2-d IFFT.

Consequently, a possible image reconstruction algorithm proceeds as follows

Figure C.5: Illustration of the Fourier Slice Theorem: The Fourier transform of the projection of $f(x, y)$ at projection angle θ, is the value of the Fourier transform $F(k_x, k_y) = F.T.(f(x, y))$ along an angle θ to the k_x axis in the frequency domain.

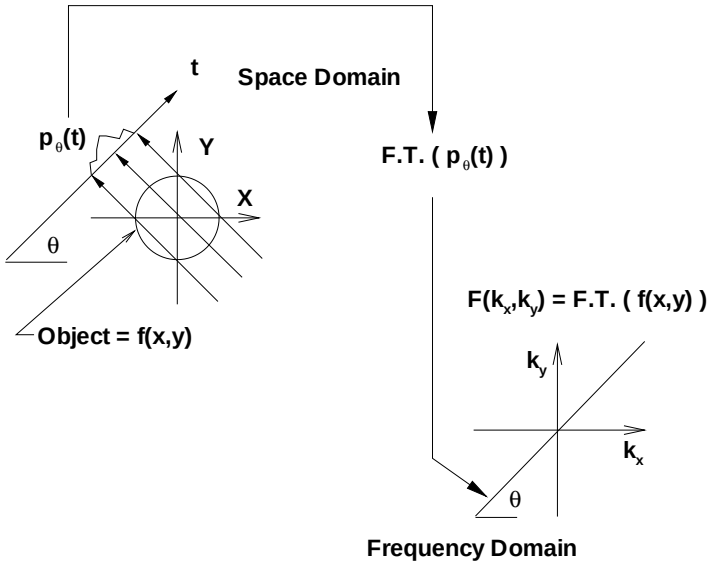
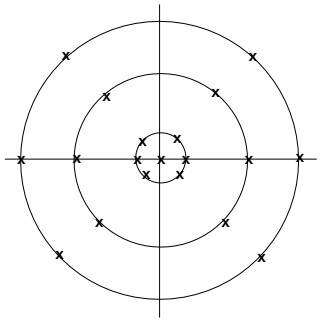


Figure C.6: Since the actual projection data is discrete, we carry out a DFT on the projections. The slice theorem then gives the value of the 2-d DFT at discrete points on circular arcs in the frequency domain.



- At each projection angle θ, carry out the FFT of the projection, which gives a slice of the DFT at angle θ in the frequency domain.
- Interpolate the slice data onto a grid in frequency domain.
- Reconstruct the image by carrying out an inverse FFT on the frequency domain data.

This direct approach does not work very well. Interpolating in the frequency domain is a perilous operation. The actual algorithm used in commercial machines is *filtered backprojection*.

C.3 Filtered Backprojection

This idea avoids the need to interpolate in the frequency domain. Represent the Fourier transform of the image $f(x, y)$ by $F(u, v)$, so that

$$f(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} F(u,v)e^{i2\pi(ux+vy)}\, du\, dv\ . \tag{C.3.1}$$

We now change the variables in the integrand in equation (C.3.1) to polar coordinates

$$\begin{aligned} u &= r\cos\theta \\ v &= r\sin\theta\ , \end{aligned} \tag{C.3.2}$$

with the infinitesimal area $du\ dv = r\ dr\ d\theta$, so that equation (C.3.1) becomes

$$f(x,y) = \int_0^{2\pi}\int_0^{\infty} F(r\cos\theta, r\sin\theta)\ e^{i2\pi r(x\cos\theta+y\sin\theta)} r\ dr\ d\theta\ . \tag{C.3.3}$$

From the Fourier Slice Theorem, we have that for fixed θ

$$F(r\cos\theta, r\sin\theta) = P(\theta, r) \tag{C.3.4}$$

$$P(\theta,r) = \int_{-\infty}^{\infty} p(\theta,t)e^{-i2\pi rt}\, dt\ , \tag{C.3.5}$$

i.e. the Fourier transform of the projection $p(\theta, t)$ at angle θ, so that equation (C.3.3) becomes

$$\begin{aligned} f(x,y) &= \int_0^{2\pi}\int_0^{\infty} P(\theta,r)e^{i2\pi r(x\cos\theta+y\sin\theta)} r\ dr\ d\theta \\ &= \int_0^{2\pi}\left[\int_0^{\infty} rP(\theta,r)e^{i2\pi r(x\cos\theta+y\sin\theta)}\, dr\right] d\theta\ . \end{aligned} \tag{C.3.6}$$

Write this as

$$\begin{aligned} f(x,y) &= \int_0^{\pi}\left[\int_0^{\infty} rP(\theta,r)e^{i2\pi r(x\cos\theta+y\sin\theta)}\, dr\right] d\theta \\ &\quad + \int_{\pi}^{2\pi}\left[\int_0^{\infty} rP(\theta,r)e^{i2\pi r(x\cos\theta+y\sin\theta)}\, dr\right] d\theta \end{aligned} \tag{C.3.7}$$

Letting $\theta' = \theta - \pi$, we can write the second term on the rhs of equation (C.3.7) as

$$\begin{aligned}
&\int_{\pi}^{2\pi} \left[\int_0^{\infty} rP(\theta, r) e^{i2\pi r(x\cos\theta + y\sin\theta)}\, dr \right] d\theta \\
= &\int_0^{\pi} \left[\int_0^{\infty} rP(\theta'+\pi, r) e^{i2\pi r(x\cos(\theta'+\pi) + y\sin(\theta'+\pi))}\, dr \right] d\theta' \\
= &\int_0^{\pi} \left[\int_0^{\infty} rP(\theta'+\pi, r) e^{-i2\pi r(x\cos\theta' + y\sin\theta')}\, dr \right] d\theta' \qquad \text{(C.3.8)}
\end{aligned}$$

Now since $p(\theta+\pi, t) = p(\theta, -t)$, then it follows from equation (C.3.5) that $P(\theta+\pi, r) = P(\theta, -r)$, so that we can write equation (C.3.8) as

$$\begin{aligned}
&\int_{\pi}^{2\pi} \left[\int_0^{\infty} rP(\theta, r) e^{i2\pi r(x\cos\theta + y\sin\theta)}\, dr \right] d\theta \\
= &\int_0^{\pi} \left[\int_0^{\infty} rP(\theta', -r) e^{-i2\pi r(x\cos\theta' + y\sin\theta')}\, dr \right] d\theta' \\
= &\int_0^{\pi} \left[\int_0^{-\infty} rP(\theta, r) e^{i2\pi r(x\cos\theta + y\sin\theta)}\, dr \right] d\theta \\
= &\int_0^{\pi} \left[\int_{-\infty}^{0} -rP(\theta, r) e^{i2\pi r(x\cos\theta + y\sin\theta)}\, dr \right] d\theta \qquad \text{(C.3.9)}
\end{aligned}$$

Combining equations (C.3.7) and (C.3.9) gives

$$f(x,y) = \int_0^{\pi} \left[\int_{-\infty}^{\infty} |r| P(\theta, r) e^{i2\pi r(x\cos\theta + y\sin\theta)}\, dr \right] d\theta\ . \qquad \text{(C.3.10)}$$

Let

$$\hat{p}(\theta, \omega) = \int_{-\infty}^{\infty} |r| P(\theta, r) e^{i2\pi r\omega}\, dr \qquad \text{(C.3.11)}$$

so that equation (C.3.10) becomes

$$f(x,y) = \int_0^{\pi} \hat{p}(\theta, x\cos\theta + y\sin\theta)\, d\theta\ . \qquad \text{(C.3.12)}$$

Consequently, the reconstruction algorithm consists of two steps

- First, the projections $p(\theta, t)$ are filtered used equation (C.3.11), to obtain $\hat{p}(\theta, t)$.
- Then the filtered projections are *backprojected* using equation (C.3.12).

C.3.1 Filtering Operation

The filtering operation in equation (C.3.11) can be done using an FFT. First, we carry out an FFT on each projection to get $P(\theta_j, \omega_k)$ at discrete points (θ_j, ω_k). Then, we multiply each $P(\theta_j, \omega_k)$ by the filter g_k, and then do an inverse FFT to get $\hat{p}(\theta_j, t_n)$.

Let's go through the use of the DFT in a bit of detail here. First recall that in the usual case of a time series, if the input signal is of length T, with N samples of size Δt, then the discrete frequencies k correspond to real frequencies of size k/T. The input signal is considered to be periodic of period T.

Let's translate this into our image application. Assume that

- In the space (time) domain, we assume that the pixels are at unit spacing. This corresponds to a $\Delta t = 1$, with input signal length N.
- The input signal is periodic of period N.
- The discrete frequencies k in this case correspond to real frequencies k/N. This corresponds to real frequencies in the range $[-1/2, +1/2]$, since $k \in [-N/2, +N/2]$.

Since the input signal is periodic of length N, then the Fourier Transform $P(\theta, \omega)$ is

$$P(\theta, \omega) = \int_{-N/2}^{+N/2} p(\theta, t) e^{-i2\pi t\omega} \, dt \ . \tag{C.3.13}$$

Discretize this integral, with $dt \simeq \Delta t = 1$, $t_n = n$, $\omega_k = k/N$, to give

$$P(\theta, \omega_k) \simeq \sum_{n=-N/2+1}^{n=+N/2} p(\theta, t_n) e^{-i2\pi nk/N} \ , \tag{C.3.14}$$

which, apart from the $1/N$ factor, is just the forward DFT.

Similarly, we can discretize the reverse DFT. Note that from our assumptions, $\omega \in [-1/2, +1/2]$, so that

$$p(\theta, t) = \int_{-1/2}^{+1/2} P(\theta, \omega) e^{i2\pi\omega t} \, d\omega \tag{C.3.15}$$

Now, discretize this integral, with $t_n = n$, $d\omega = 1/N$, $\omega_k = k/N$, to get

$$p(\theta, t_n) \simeq \frac{1}{N} \sum_{k=-N/2+1}^{k=N/2} P(\theta, \omega_k) e^{i2\pi kn/N} \ , \tag{C.3.16}$$

which we recognize as the inverse DFT, apart from the factor of $1/N$.

Therefore, the integral in equation (C.3.11) can be approximated as

$$\begin{aligned} \hat{p}(\theta_j, t_n) &= \int_{-1/2}^{+1/2} |\omega| P(\theta_j, \omega) e^{i2\pi\omega t_n} \, d\omega \\ &\simeq \frac{1}{N} \sum_{k=-N/2+1}^{k=N/2} g_k P(\theta_j, \omega_k) e^{i2\pi k/N} \ . \end{aligned} \tag{C.3.17}$$

where the filter function is

$$g_k = |\omega_k| = \frac{|k|}{N} \ . \tag{C.3.18}$$

Note that equations (C.3.14) and (C.3.16) differ from the usual DFTs as defined in these notes, since the $1/N$ factor is multiplied during the reverse transform, rather than during the forward transform. Since we will do a forward transform, a filtering operation, and then a reverse transform, it really does not matter where we multiply by the $1/N$ factor. We can just use any FFT software to do this.

To summarize, we take the FFT of each projection at a fixed angle, multiply each discrete Fourier component by the filter function (C.3.18), then do an inverse FFT to get $\hat{p}(\theta_j, t_n)$. Note that the filter function is defined for $k \in [-N/2+1, +N/2]$, so you have to be careful when applying this to the Fourier components $P_k, k \in [0, N-1]$.

C.3.2 Back Projection

We approximate the integral in equation (C.3.12) at discrete pixel values $f(x_i, y_i)$, using the set of discrete projections θ_m as follows

$$\begin{aligned} f(x_i, y_j) &\simeq \Delta\theta \sum_{m=1}^{m=M} \hat{p}(\theta_m, x_i \cos\theta_m + y_j \sin\theta_m) \\ \Delta\theta &= \frac{\pi}{M} \ , \end{aligned} \tag{C.3.19}$$

where, since $\hat{p}(\theta_m, t_n)$ is only known at discrete values of t_n, we will have to use interpolation. Note that we are assuming that

$$\theta_m \in \left\{ 0, \frac{\pi}{M}, \frac{2\pi}{M}, ..., \frac{(M-1)\pi}{M} \right\} . \tag{C.3.20}$$

Note that if take the projection $\hat{p}(t, \theta_m)$, and map it to a line through the origin, at an angle θ_m to the x-axis, with t being the radial coordinate, then the point $(r, \theta) = (x_i \cos\theta_m + y_j \sin\theta_m, \theta_m)$ is the projection of the point (x, y) onto this line. This makes for a simple and fast implementation of equation (C.3.19). This is illustrated in Figure C.7.

C.3.3 Example Reconstruction

Figure C.8 shows a reconstruction with 90 slices, using backprojection only, no filtering. Note the very poor quality of the reconstruction.

Figure C.9 shows a reconstruction from projections using only 15 projections.

Figure C.7: The backprojection step.

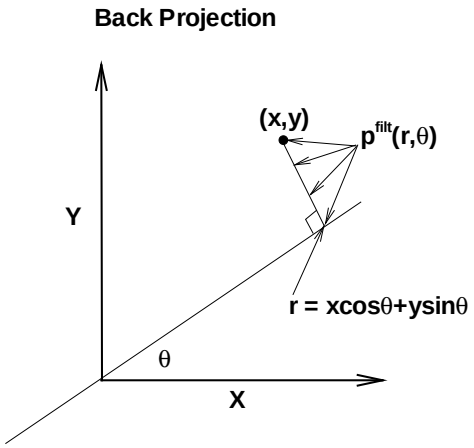


Figure C.8: Example reconstruction from projections using a 90 projections. Back projection only, no filtering.

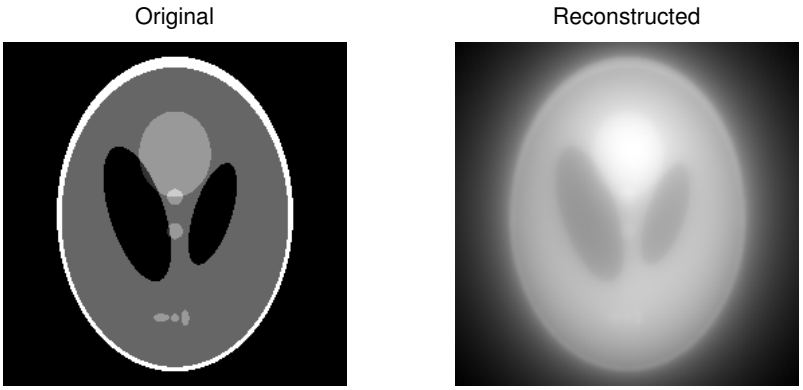


Figure C.9: Example reconstruction from projections using a small number (15) of projections. Note the artifacts which appear in the reconstruction.

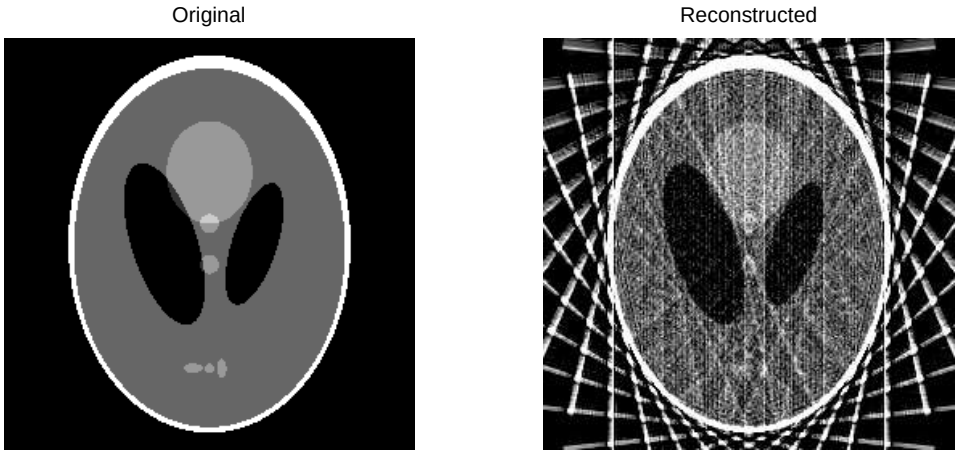
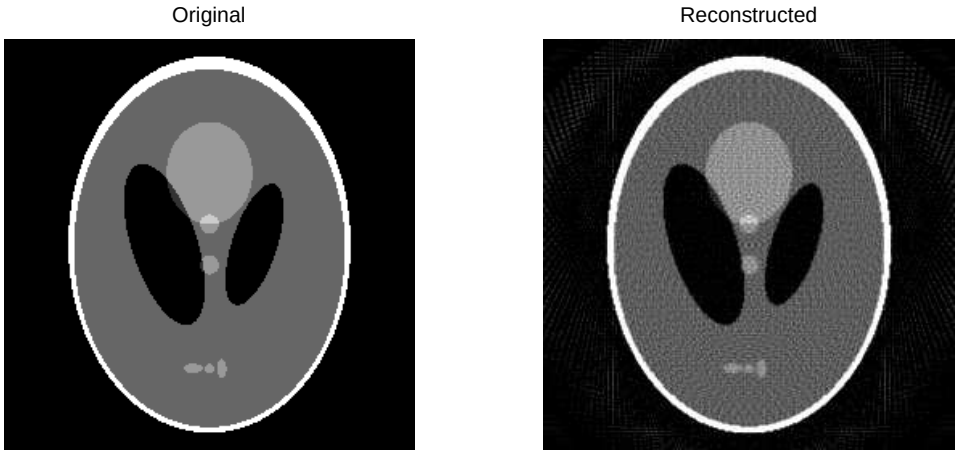


Figure C.10: Example reconstruction from projections using a 90 projections. Note the artifacts in Figure C.9 have disappeared.



Appendix D

Big Oh

We wish to consider a general method of analyzing the behaviour of polynomials in two cases: when $x \to 0$ and when $x \to \infty$. In particular, if we are only interested in the asymptotic behaviour of the polynomial as opposed to the exact value of the function, we may employ the concept of Big-Oh notation.

Definition D.0.1 *Suppose that $f(x)$ is a polynomial in x without a constant term. Then the following are equivalent:*

a) $f(x) = O(x^n)$ *as* $x \to 0$.

b) $\exists\ c > 0, x_0 > 0$ *such that* $|f(x)| < c|x|^n\ \forall\ x$ *with* $|x| < |x_0|$.

c) $f(x)$ *is bounded from above by* $|x|^n$*, up to a constant* c*, as* $x \to 0$.

This effectively means that the dominant term in $f(x)$ is the term with x^n as $x \to 0$, or $f(x)$ goes to zero with order n.

Example D.0.2 *Consider the polynomial* $g(x) = 3x^2 + 7x^3 + 10x^4 + 7x^{12}$*. We say*

$$\begin{aligned} g(x) &= O(x^2) \text{ as } x \to 0 \\ g(x) &\neq O(x^3) \text{ as } x \to 0 \\ g(x) &= 3x^2 + O(x^3) \text{ as } x \to 0 \end{aligned}$$

We note that $g(x) = O(x)$ *as well, but this statement is not so useful because it is not a sharp bound.*

We may also consider the behaviour of a polynomial as $x \to \infty$.

Definition D.0.3 *Suppose that $f(x)$ is a polynomial in x. Then the following are equivalent:*

a) $f(x) = O(x^n)$ *as* $x \to \infty$.

b) $\exists\ c > 0, x_0 > 0$ *such that* $|f(x)| < c|x|^n\ \forall\ x$ *with* $|x| > |x_0|$.

c) $f(x)$ *is bounded from above by* $|x|^n$*, up to a constant* c*, as* $x \to \infty$.

As before, this effectively means that the dominant term in $f(x)$ is the term with x^n as $x \to \infty$, or $f(x)$ goes to infinity with order n.

Example D.0.4 *Consider the polynomial* $g(x) = 3x^2 + 7x^3 + 10x^4 + 7x^{12}$. *We say*

$$\begin{aligned} g(x) &= O(x^{12}) \text{ as } x \to \infty \\ g(x) &\neq O(x^8) \text{ as } x \to \infty \\ g(x) &= 7x^{12} + O(x^4) \text{ as } x \to \infty \end{aligned}$$

Addition and Multiplication of Terms Involving Big-Oh

We can also add and multiply terms using Big-Oh notation, making sure to neglect higher order terms:

$$\begin{array}{rcll} f(x) & = & x + O(x^2) & \text{as } x \to 0 \\ g(x) & = & 2x + O(x^3) & \text{as } x \to 0 \\ \hline f(x) + g(x) & = & 3x + O(x^2) + O(x^3) & \text{as } x \to 0 \\ & = & 3x + O(x^2) & \text{as } x \to 0 \\ f(x) \cdot g(x) & = & 2x^2 + O(x^3) + O(x^4) + O(x^5) & \text{as } x \to 0 \\ & = & 2x^2 + O(x^3) & \text{as } x \to 0 \end{array}$$

Applications to the Taylor Series Expansion

Recall that the Taylor series expansion for a function $f(x)$ around a point x_0 is given by

$$f(x_0 + \Delta x) = \sum_{n=0}^{\infty} \frac{1}{n!} f^{(n)}(x_0)(\Delta x)^n \tag{D.0.1}$$

We may expand this formula and write it in terms of Big-Oh notation as follows:

$$\begin{array}{rcl} f(x_0 + \Delta x) & = & f(x_0) \\ & + & f'(x_0)\Delta x \\ & + & \frac{1}{2} f''(x_0)\Delta x^2 \\ & + & \frac{1}{6} f'''(x_0)\Delta x^3 \\ & + & O(\Delta x^4) \text{ as } \Delta x \to 0 \end{array}$$