CS370

Course Notes for Introduction to Scientific Computation

Professor Keith Geddes

Fall term, 2008

David R. Cheriton School of Computer Science
University of Waterloo

August 28, 2008

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.

Contents

1 Floating Point Number Systems

1.1 Pitfalls in Floating Point Computation

The content of most mathematical courses assumes that all the arithmetic is *exact* when working, for example, over the real number field $\mathbb{R}$. However, problems in scientific computation typically use a computer's floating point number system, a system in which most real numbers have only an approximate representation. In this section we give some examples of computational pitfalls that result from the use of inexact arithmetic.

Example 1

Suppose we would like to compute the value $e^{-5.5}$. One possible method is to use the Taylor series expansion

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ... \tag{1.1}$$

If we use a calculator which carries five significant figures, we get

$$e^{-5.5} = 1.0 - 5.5 + 15.125 - 27.730 + ... \tag{1.2}$$

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

Another method is to note that

$$e^{-x} = \frac{1}{e^x} = \frac{1}{1 + x + \frac{x^2}{2!} + ...} \tag{1.3}$$

so that

$$e^{-5.5} = \frac{1}{1 + 5.5 + 15.125 + ...} \tag{1.4}$$

and again after 25 terms, additional terms no longer change the sum. The correct answer, to five significant digits, is

$$e^{-5.5} = .0040868. \tag{1.5}$$

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

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

Why is the result obtained by the first method so wrong?

1.2 Floating Point Numbers

For most of our discussions of numerical computation, we assume we are using the arithmetic of the mathematically defined real number system, which we denote by $\mathbb{R}$. The number system $\mathbb{R}$ is infinite in two senses:

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

2. $\mathbb{R}$ 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.

Digital computers can represent only finite sets of numbers, so all implementations of algorithms must use approximations to $\mathbb{R}$ and inexact arithmetic. The approximations to $\mathbb{R}$ used by digital computers are known as *floating point number systems.*

To describe floating point number systems, we need to look first at the representation of real numbers as *normalized digital expansions* relative to some chosen base for the number system. As humans, we (usually) use the base-10 number system, i.e. digits $0, 1, 2, \ldots, 9$, also called the decimal digits. For example, the rational number $\frac{73}{3}$ can be represented by the following normalized decimal digit expansion:

$$2.4333333\ldots \times 10^1 = 2 \times 10^1 + 4 \times 10^0 + 3 \times 10^{-1} + 3 \times 10^{-2} + \cdots$$

Digital computers, however, typically use the base-2 number system (binary), which can be expressed using the digits 0 and 1. Some computers use the base-16 number system (hexadecimal), in which the digits are usually expressed as $0, 1, 2, \ldots, 9, A, B, C, D, E$ and F. Note that

$$\frac{73}{3} = 24 + \frac{1}{3} = 16 + 8 + \frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \cdots .$$

So in base 2, $\frac{73}{3}$ can be represented by the normalized binary digit expansion

$$1.1000010101\ldots \times 2^4 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + \cdots$$

while in base 16, $\frac{73}{3}$ can be represented by the normalized hexadecimal digit expansion

$$1.8555555\ldots \times 16^1 = 1 \times 16^1 + 8 \times 16^0 + 5 \times 16^{-1} + 5 \times 16^{-2} + \cdots .$$

The Floating Point Number System $\mathcal{F}(\beta, t, L, U)$

Let β be a positive integer that is to be used as the base for a number system; e.g.

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

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

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

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

$$d_0.d_1d_2d_3\cdots \times \beta^p$$

where

- d_k are base-β digits, i.e. $d_k \in \{0, 1, \ldots, \beta - 1\}$
- 'normalized' means $d_0 \neq 0$
- p is an integer (not necessarily positive).

We call $d_0.d_1d_2d_3\ldots$ the **significand** (also sometimes called the **mantissa**), β the **base**, and p the **exponent**.

Floating point number systems limit the infinite density of $\mathbb{R}$ by retaining a fixed number, t, of digits in the significand (t is called the **precision** of the number system). They limit the infinite extent of $\mathbb{R}$ by allowing only a finite number of integer values for the exponent p, specifically by imposing the restriction $L \leq p \leq U$ for two fixed integer bounds L and U. So every floating point number system is identified by four integer parameters, $\{\beta, t, L, U\}$, which are the base, the precision, and lower and upper bounds on the exponent range. The numbers which can be represented in such a system are precisely those of the form

$$\begin{array}{ll} & \pm\, d_0.d_1d_2...d_{t-1} \times \beta^p \quad \text{for } \; L \leq p \leq U \quad \text{with} \quad d_0 \neq 0 \\ \text{and} & 0 \quad \text{(a special case).}\end{array}$$

There are two standardized floating point number systems that are widely used in the design of computer software and hardware:

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

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

We will denote a floating point number system by $\mathcal{F}(\beta, t, L, U)$ or sometimes simply by $\mathcal{F}$ when the parameters are understood.

1.3 Absolute and Relative Error

When we obtain a computed result, x, and we wish to discuss its relationship with the correct mathematical result, x_{exact}, we can measure either the **absolute error** :

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

or we can measure the **relative error**:

$$Err_{\text{rel}} = \frac{|x_{\text{exact}} - x|}{|x_{\text{exact}}|}\ .$$

Note: When measuring relative error, sometimes the value used in the denominator is the computed value $|x|$ rather than $|x_{\text{exact}}|$ as stated above. In most cases, the difference between the two definitions is insignificant.

For the type of computations performed on a digital computer, the relative error measure is most useful. This is because there is a close relationship between relative error and the number of correct significant digits in the computed result.
Remark: The *significant digits* of a number are all the digits starting with the leftmost nonzero digit.

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

$$0.5 \times 10^{-s} \le \frac{|x_{\text{exact}} - x|}{|x_{\text{exact}}|} < 5.0 \times 10^{-s} \ . \tag{1.6}$$

Consider the situation from Example 1. Two different methods are used to compute a floating point approximation for the value $e^{-5.5}$. The correct value, to five significant digits, is

$$e^{-5.5} = 0.0040868 \ .$$

The value computed by the first method is $x_1 = 0.0026363$, yielding a relative error of

$$Err_{\text{rel}} = \frac{|0.0040868 - x_1|}{0.0040868} \approx 3.5 \times 10^{-1} \, .$$

Based on equation (1.6), we would estimate that x_1 has approximately one significant digit correct (in actual fact, we can see that it has no correct digits).

Next consider the value computed by the second method, $x_2 = 0.0040865$. Its relative error is

$$Err_{\text{rel}} = \frac{|0.0040868 - x_2|}{0.0040868} \approx 0.7 \times 10^{-4} \, .$$

Based on equation (1.6), we would estimate that x_2 has approximately four significant digits correct. We can see that x_2 is indeed correct to four significant digits.

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. For example, suppose that the correct mathematical result for some problem is

$$x_{\text{exact}} = 4.0868 \, .$$

This is the same answer as expected in Example 1 but multiplied by 10^3 (e.g., suppose that the correct result is $1000\, e^{-5.5}$). If we get a computed result of $x_1 = 2.6363$ then its relative error is 3.5×10^{-1}, precisely the same as for x_1 in the preceding example. If we get a computed result of $x_2 = 4.0865$ then its relative error is 0.7×10^{-4}, precisely the same as for x_2 in the preceding example. Again, x_2 has four significant digits correct, and the relative error predicts this independently of the magnitudes of the numbers.

1.4 Relationship between x and its representation $fl(x)$

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 within range).

This maximum relative error measure of machine precision is called **machine epsilon**, denoted here by ε. It is also called the **unit roundoff error** since ε can be defined as the smallest representable number such that, in $\mathcal{F}(\beta, t, L, U)$,

$$1 + \varepsilon > 1 \, .$$

If a computer uses base β arithmetic with t digits in the significand, i.e. $\mathcal{F}(\beta, t, L, U)$, then the value of machine epsilon (or unit roundoff error) is

$$\varepsilon = \frac{1}{2} \beta^{1-t} \, .$$

In general then, the relative error between any nonzero real number x and its floating point representation is bounded by ε. Let $fl(x)$ be the floating point representation of x. Then

$$\frac{|fl(x) - x|}{|x|} \leq \varepsilon \; .$$

This relationship can also be expressed as

$$\begin{aligned} fl(x) - x &= \delta \; x \\ & \quad \text{where } |\delta| \leq \varepsilon \end{aligned}$$

and thus we have

$$fl(x) = x \, (1 + \delta) \tag{1.7}$$

where δ is some value (positive, negative or zero) such that $-\varepsilon \leq \delta \leq \varepsilon$.

The IEEE standard basically says that a single arithmetic operation in $\mathcal{F}$ must be done so that the computed result is rounded to the nearest representable floating point number to the exact real arithmetic result. An **exception** occurs if the exponent is out of range, which leads to a state called **overflow** if the exponent is too large, or **underflow** if the exponent is too small.

Let us denote the floating point addition operator in $\mathcal{F}$ by $\oplus$. Then for adding two floating point numbers $w, z \in \mathcal{F}$, using (1.7), we have

$$w \oplus z = fl(w + z) = (w + z)(1 + \delta) \, .$$

Next we will consider adding three floating point numbers a, b, $c \in \mathcal{F}$ using the floating point addition operation in $\mathcal{F}$.

Exercise

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

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

1.5 Roundoff Error Analysis

Although each floating point arithmetic operation is done with a relative error that is bounded by ε, 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 ε. Consider adding three floating point numbers $a, b, c \in \mathcal{F}$.

How does $(a \oplus b) \oplus c$ differ from the true sum $a + b + c$? I.e., what is the size of the relative error in this sum computed in $\mathcal{F}$? We will do a small exercise in traditional floating point error analysis to answer this question.

First we express $(a \oplus b) \oplus c$ in terms of exact additions with relative error perturbations δ_1 due to the first operation, and δ_2 due to the second operation:

$$\begin{aligned}(a \oplus b) \oplus c &= (a+b)(1+\delta_1) \oplus c = \big((a+b)(1+\delta_1)+c\big)(1+\delta_2)\\ &= \big((a+b+c)+(a+b)\delta_1\big)(1+\delta_2)\\ &= (a+b+c)+(a+b)\delta_1+(a+b+c)\delta_2+(a+b)\delta_1\delta_2\,. \qquad (1.8)\end{aligned}$$

Therefore,

$$|(a+b+c)-((a \oplus b) \oplus c)| \le (|a|+|b|+|c|)\,(|\delta_1|+|\delta_2|+|\delta_1|\,|\delta_2|) \qquad (1.9)$$

where we have slightly increased the upper bound (the right hand side) to make the bound symmetric in the variables a, b and c. In other words, the upper bound as stated is not affected by the order of the terms in the sum.

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

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

is bounded as follows, from (1.9):

$$Err_{\text{rel}} \;\le\; \frac{|a|+|b|+|c|}{|a+b+c|}\,(2\,\varepsilon+\varepsilon^2)\,. \qquad (1.10)$$

Note that (1.10) 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 dominated by $(2\,\varepsilon+\varepsilon^2)$ which is small;
- if $|a+b+c| \ll |a|+|b|+|c|$ then Err_{rel} can be quite large, namely $(2\,\varepsilon+\varepsilon^2)$ multiplied by the "magnification factor" $\frac{|a|+|b|+|c|}{|a+b+c|}$.

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). This phenomenon, known as **cancellation**, can occur when adding a mix of both positive and negative values.

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

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

Then $|a| + |b| + |c| = 20003.1416$ and $a + b + c = 3.1416$. Thus, the relative error bound given by (1.10) is

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

since $\varepsilon = \frac{1}{2}\,10^{-4}$. (Note that since the unit roundoff error ε is a small quantity, ε^2 is very small, and we can approximate $2\,\varepsilon + \varepsilon^2 \approx 2\,\varepsilon$.) This relative error of 0.6×10^0 is quite 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.

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

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

In this case the relative error bound given by (1.10) is

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

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 ε.) 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.

The error bound (1.10) for the case of adding three numbers can be generalized to the case of adding N numbers. Let $x_i \in \mathcal{F}$, $i = 1, \ldots, N$, be N given floating point numbers (i.e. numbers stored in a computer system). If

$$fl\left(\sum_{i=1}^{N} x_i\right)$$

denotes the computed result of adding the N numbers in $\mathcal{F}$, and if $\sum_{i=1}^{N} x_i \neq 0$ then the relative error bound can be expressed as

$$\frac{\left|\sum_{i=1}^{N} x_i - fl\left(\sum_{i=1}^{N} x_i\right)\right|}{\left|\sum_{i=1}^{N} x_i\right|} \leq \frac{\sum_{i=1}^{N} |x_i|}{\left|\sum_{i=1}^{N} x_i\right|}\, 1.01\, N\,\varepsilon\,. \tag{1.11}$$

The appearance of the factor 1.01 in (1.11) is an artificial technicality. We had noted in the roundoff error analysis for adding three numbers that, for practical purposes,

$2\,\varepsilon + \varepsilon^2 \approx 2\,\varepsilon$ since ε is small. Similarly, the analysis leading to the bound (1.11) has employed a simplification such that an expression of the form

$$(N-1)\,\varepsilon + \frac{(N-1)(N-2)}{2}\,\varepsilon^2 + \cdots + \varepsilon^{N-1}$$

has been replaced by an upper bound of the form $1.01\,N\,\varepsilon$, which holds as long as $N < \frac{.01}{\varepsilon}$. This level of detail is of no importance to us here; in applying the bound (1.11) we understand that, for practical purposes, $1.01\,N\,\varepsilon \approx N\,\varepsilon$.

A similar roundoff error analysis for a product of numbers, rather than a sum, shows that the error bound is always approximately $N\,\varepsilon$. This arises because for a product we have

$$\left|\prod_{i=1}^{N} x_i\right| = \prod_{i=1}^{N} |x_i|$$

and therefore,

$$\frac{\left|\prod_{i=1}^{N} x_i - fl\left(\prod_{i=1}^{N} x_i\right)\right|}{\left|\prod_{i=1}^{N} x_i\right|} \leq 1.01\,N\,\varepsilon\,. \tag{1.12}$$

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 small change of size ΔI in one or more input values causes a relatively small change in the mathematically correct output values, then the problem is said to be **well-conditioned**. Otherwise, the problem is said to be **ill-conditioned**. Stated another way, an ill-conditioned problem has output that is very sensitive to slight changes in the input.

Remark 1: This phenomenon is a statement about a mathematical problem, and is not due to floating point errors.

Remark 2: There is always a "sliding scale" for "how well-conditioned" or "how ill-conditioned" a particular problem is.

Example 2

Finding the point of intersection of two lines requires some calculation. Depending on the lines themselves, this calculation may be well-conditioned or ill-conditioned. The lines in Figure 1(a) are nearly perpendicular. A small adjustment to the slope of one of the lines will not move the point of intersection very much, and hence that calculation is well-conditioned. On the other hand, the lines in Figure 1(b) are nearly parallel. A slight adjustment to the slope of one of those lines will move the point of intersection quite a lot. That calculation is ill-conditioned.

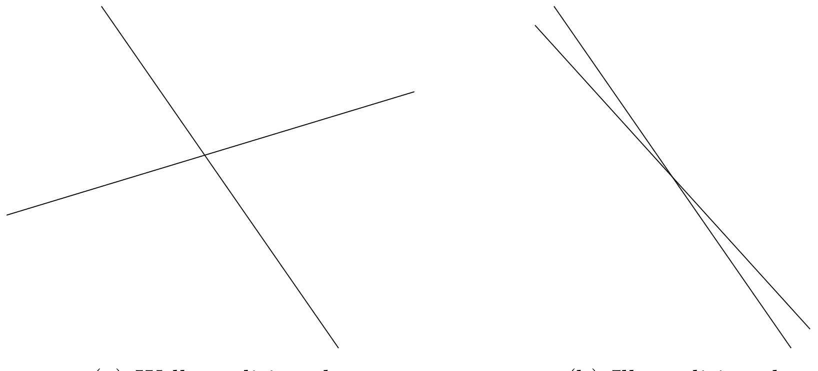


Figure 1: The calculation to find the intersection of two lines can be well-conditioned, as in (a), or ill-conditioned, as in (b).

Looking back to Example 1

Let the problem P be: Given x, compute $f(x) = e^x$. In this case, $I = \{x\}$ and $O = \{e^x\}$.

We can prove that this problem is well-conditioned. To do so, we use some calculus to obtain the following estimate:

$$\frac{|f(x) - f(x + \Delta x)|}{|f(x)|} \approx \frac{|f'(x)|\,|\Delta x|}{|f(x)|}.$$

This comes from the Taylor series expansion

$$\begin{aligned} f(x + \Delta x) &= f(x) + f'(x)\,\Delta x + \frac{1}{2} f''(x)\,\Delta x^2 + O(\Delta x^3) \\ &\approx f(x) + f'(x)\,\Delta x \end{aligned}$$

assuming that $|\Delta x|$ is small.

Expressing the above as a relationship between the *relative* change in input, $\frac{|\Delta x|}{|x|}$, versus the *relative* change in output, we have

$$\frac{|f(x) - f(x + \Delta x)|}{|f(x)|} \approx \frac{|x|\,|f'(x)|}{|f(x)|} \times \frac{|\Delta x|}{|x|}.$$

The *condition number* $\kappa(P)$ of problem P is defined to be the maximum "magnification factor" by which a relative change in the input values may be magnified in the corresponding changes in output values due solely to the mathematical problem (independent of any particular algorithm).

In this example, the analysis tells us that the condition number is

$$\kappa(P) = \frac{|x|\,|f'(x)|}{|f(x)|}$$

for any function $f(x)$. In our example, $f'(x) = f(x)$ so in this particular case we have

$$\kappa(P) = |x| \, .$$

The computational results obtained by the two methods outlined in Example 1 are

$$\begin{aligned} e^{-5.5} &= 1 - 5.5 + 15.125 - \cdots = 0.0026363 \\ e^{-5.5} &= \frac{1}{1 + 5.5 + 15.125 + \cdots} = 0.0040865 \, . \end{aligned}$$

Using a "bad algorithm" (the first method), the computed result was 0.0026363 compared with the correct value which is 0.0040868. In other words, all digits in the computed result are wrong!

This bad computational result cannot be blamed on the problem: the problem as stated is well-conditioned. From above, the condition number for this problem is

$$\kappa(P) = |x| = 5.5 \, .$$

Noting that $\kappa(P) < 10$, we can conclude that roundoff errors (in relative error) of size ε (the unit roundoff error) can lead to relative errors in the output bounded by

$$Err_{\text{rel}} \approx \kappa(P)\, \varepsilon < 10\, \varepsilon \, .$$

For example, if ε is approximately 10^{-5} as in Example 1, then the output might have relative error as large as 10^{-4} meaning that we should expect to have about four significant digits correct (rather than than all five digits correct). This is the worst that can happen due to the conditioning of the problem.

Any larger errors, such as seen in this example, must be blamed on the choice of a bad algorithm. Indeed, Example 1 shows a good algorithm which computes $e^{-5.5}$ to good accuracy. We say that the first computation is using an unstable algorithm and the second computation is using a stable algorithm. The concept of stability may be defined as follows in a general setting.

Consider a problem P with condition number $\kappa(P)$ and suppose that we apply algorithm A to solve problem P. If we can guarantee that the computed output values from algorithm A will have relative errors not too much larger than the errors due to the condition number $\kappa(P)$, then algorithm A is said to be **stable**. Otherwise, if the computed output values from algorithm A can have much larger relative errors, then algorithm A is said to be **unstable**.

We can state the concept of stability more informally as follows. If an algorithm produces inaccurate results, we have to try to determine whether the errors can be blamed on the mathematical problem as stated (i.e. the problem has a large condition number), or, as is often the case, we have chosen an unstable algorithm to solve the problem. In the latter case, we try to find a more stable algorithm; i.e., an algorithm which does not cause a large magnification of errors. Since we always commit roundoff errors during the execution of an algorithm in a floating point number system, it is the magnification of such errors that is of concern to us.

Exercises

1. The numbers in a floating point system $\mathcal{F}(\beta, t, L, U)$ are defined by a base β, a significand length t, and an exponent range $[L,\, U]$. A nonzero floating point number x has the form
$$x = \pm\, d_0.d_1 d_2 \cdots d_{t-1} \times \beta^e\,.$$
Here $d_0.d_1 d_2 \cdots d_{t-1}$ is the significand and e is the exponent. The exponent satisfies $L \le e \le U$. The d_i are base-β digits and satisfy $0 \le d_i \le \beta-1$. Nonzero floating point numbers x are normalized: $d_0 \ne 0$. The floating point number zero is represented by setting all digits in the significand to zero and setting $e = L$.

    (a) What is the largest value of n so that $n!$ can be exactly represented in the floating point number system $\mathcal{F}(2, 5, -10, 10)$? Show your work.

    (b) Suppose that on a base-2 computer, 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 point numbers in a base-2 computer with t-bit significand. How small can $y - x$ be if $x < 8 < y$?

2. Consider a fictitious floating point number system composed of the following numbers:
$$\begin{aligned} S \;=\{\;\; & \pm\, d_1.d_2 d_3 \times 2^{\pm y} :\; d_2, d_3, y = 0 \text{ or } 1, \\ & \text{and } d_1 = 1 \;\text{ unless } d_1 = d_2 = d_3 = 0 \;\;\}\,. \end{aligned}$$
I.e. each number is normalized unless it is the number zero.

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

    (b) Indicate on your plot the regions of OFL (overflow) and UFL (underflow).

    (c) How many elements are contained in S?

    (d) What is the value of ε (machine epsilon)?

3. Using the floating point number system $\mathcal{F}(2, 20, -200, 200)$, represent the distance between the Earth and the Sun (1.5×10^8 kilometers) and the distance between Toronto and Waterloo (110 kilometers). What length does the last bit of the significand represent in each case?

4. Carry out a roundoff error analysis to show that, in a floating point number system, if $ab + c \ne 0$ then
$$\frac{|(ab + c) - ((a \otimes b) \oplus c)|}{|ab + c|} \le \frac{|ab|}{|ab + c|}\varepsilon\,(1 + \varepsilon) + \varepsilon$$
where ε denotes machine epsilon. Justify each inequality that you introduce.

2 Interpolation

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

One approach is to require that our approximate function $g(x)$ should have the property that it *interpolates* the data, that is,

$$g(x_1) = y_1, \ \ldots \ , \ g(x_n) = y_n. \tag{2.1}$$

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

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

interpolates the four points (as shown in Figure 2) and we can evaluate $g(x)$ for other x-values. For example, estimate the value of $g\left(\frac{3}{2}\right)$ just by looking at the graph.

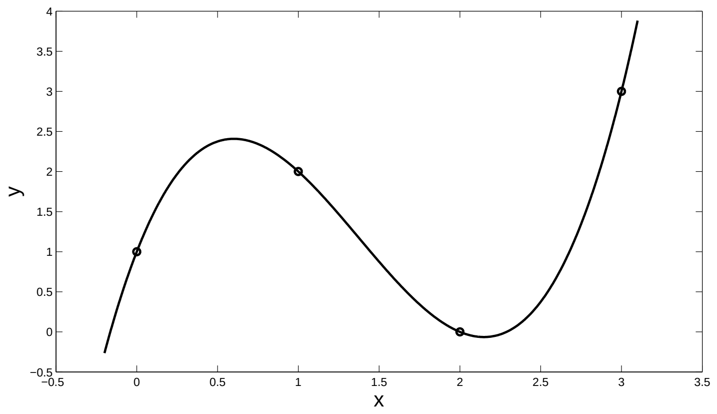


Figure 2: $g(x)$ interpolates the four points $(0, 1)$, $(1, 2)$, $(2, 0)$ and $(3, 3)$

In this section, we discuss several standard ways to construct and evaluate functions of one variable, $g(x)$, that pass through the points $(x_i, y_i), i = 1, 2, \ldots, n$, i.e.

$$g(x_i) = y_i \qquad \text{for } i = 1, \ldots, n \ . \tag{2.2}$$

Any function that satisfies (2.2) is said to **interpolate** the data. In general, the interpolation function $g(x)$ is not unique.

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

- for applications involving a small number of points, i.e. typically n not larger than 5 or 6;
- as a component (conceptually, or explicitly) of a larger computation. E.g. numerical integration, numerical solution of differential equations, or numerical optimization.

The second topic of this chapter is piecewise polynomial interpolation. In that case, $g(x)$ is defined as a different polynomial for each separate subinterval in a specified subdivision of the domain of x values. We discuss two particularly common cases:

- piecewise linear interpolation, in which $g(x)$ is linear on each subinterval;
- cubic splines, in which $g(x)$ is a cubic polynomial 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

In polynomial interpolation, the interpolation function $g(x)$ is chosen to be a single polynomial. Practically speaking, a polynomial is easy to evaluate (e.g., using Horner's rule). Theoretically speaking, it has a nice existence and uniqueness property, as follows.

2.1.1 The Basic Theorem of Polynomial Interpolation

Theorem: Given n points, (x_i, y_i), $i = 1, \ldots, n$ with $x_i \neq x_j$ if $i \neq j$, 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. 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 (discussed in section 2.1.3).

Note that the theorem only claims the existence of an interpolating polynomial. In the next two subsections, we describe two approaches for constructing the interpolating polynomial.

2.1.2 The Vandermonde System

A polynomial of degree (at most) $n-1$ is commonly represented in terms of the standard monomial basis as follows:

$$p(x) = c_1 + c_2 x + \cdots + c_n x^{n-1} \tag{2.3}$$

defined by n coefficients $c_1, ..., c_n$. The most straightforward method to define a polynomial interpolant is via a linear system of equations. Specifically, the interpolation conditions $p(x_i) = y_i$ for $i = 1, \ldots, n$ define a system of n linear equations to be solved for the n unknown coefficients. For example, for the four planar points given previously the interpolating polynomial is of the form $p(x) = c_1 + c_2x + c_3x^2 + c_4x^3$ and the interpolation conditions yield the following linear system of equations:

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

In general, if we are given a set of data $(x_1, y_1), ..., (x_n, y_n)$ and we wish to determine an interpolating polynomial of the form (2.3) then we can set up the linear system $V \cdot \vec{c} = \vec{y}$ where

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

A matrix of the form V is called a *Vandermonde* matrix. Note that all of the data required for defining a Vandermonde matrix is contained in its second column, namely $V_{i,2} = x_i$ for $i = 1, \dots, n$.

The formulation of the interpolation problem into a linear system of equations has both practical and theoretical implications. The theoretical implication is that we can prove the basic theorem in Section 2.1.1 by showing that V is nonsingular if the n values $\{x_i\}$ are distinct. 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 the problem of computing the interpolating polynomial to solving a linear system of equations.

2.1.3 The Lagrange Form

Expressing a polynomial in its standard monomial (or power) basis is common practice and computing with polynomials in this form seems simple. However, there are alternative forms for expressing a polynomial that can be more efficient for certain applications, in particular for polynomial interpolation. The Lagrange form is one such alternative.

Given a set of data (x_i, y_i), $i = 1, \dots, n$, we define the n *Lagrange basis functions* $L_k(x)$, $k = 1, \dots, n$, as follows:

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

It is easy to show that

$$L_k(x_j) = \begin{cases} 0 & j \neq k \\ 1 & j = k \end{cases}. \tag{2.4}$$

Note that $L_k(x)$, for any other real number x, need not be 0 or 1.

Define a polynomial $p(x)$ in terms of the Lagrange basis functions using as coefficients the given $\{y_i\}$ data values as follows:

$$p(x) = y_1 L_1(x) + y_2 L_2(x) + \cdots + y_n L_n(x) \ .$$

Then $p(x)$ is a polynomial of degree $n-1$ that interpolates the n data 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 (interpolating four points) discussed above, we have:

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

and the Lagrange interpolating polynomial is

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

One can check by multiplying out the factors that the polynomial expressed here in Lagrange form is the same polynomial determined previously in standard monomial form, namely

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

Exercises

1. Consider data $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ with $x_1 < x_2 < x_3$ and $y_2 > \max(y_1, y_3)$. Graphically, the middle data point is higher than the two end points. This data is interpolated by a quadratic polynomial, $p_2(x)$. Intuitively, it seems clear that $p_2(x)$ lies above the line joining the two end data points for $x_1 < x < x_3$, and, consequently, there is a maximum value of $p_2(x)$ in this interval.

   We want to show algebraically that this is correct and give a formula for the maximum. The algebra (and programs for doing computations) are simplified by doing some transformations of the data. By rescaling the x-variable, we can assume that $x_1 = 0$ and $x_3 = 1$; we will rename $x_2 = a$ for convenience. Without loss of generality, we can also assume that $y_1 < y_3$ and, by subtracting y_1 from each y_k, that $y_1 = 0$.

   (a) The line interpolating the end points of the normalized data is $p_1(x) = y_3 x$. Show that $p_2(x) > p_1(x)$ for $0 < x < 1$.
   Hint: The Lagrange representation of $p_2(x)$ may be helpful.

(b) Show that the coefficient of x^2 in $p_2(x)$ is negative.

(c) Let $x_{\max}$ be the x-value at which $p_2(x)$ achieves its maximum. Derive a formula for $x_{\max}$ in terms of a, y_2, and y_3.

2. Let $L_k(x), k = 1 \dots n$, be the Lagrange basis functions for $x_k = k$, $k = 1 \dots n$. By considering an appropriately chosen data set $\{y_k\}$ and using the basic theorem in Section 2.1.1, prove that

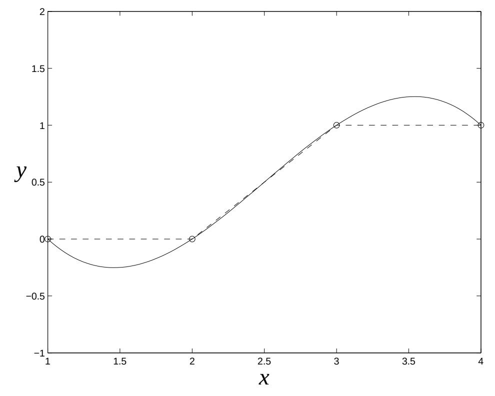
$$\sum_{k=1}^{n} k \ L_k(x) = x$$

for *every* value of x.

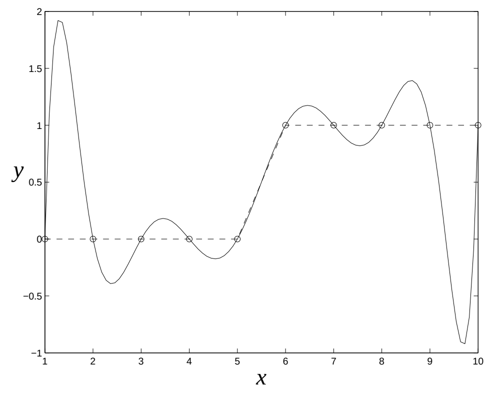
2.2 Piecewise Polynomial Interpolation

Polynomial interpolation may not be suitable when the number of interpolation points gets large. Figure 3 shows how the interpolating polynomial can have wild oscillations in order to pass through the points. For cases like this, piecewise polynomial interpolation might be more appropriate.

Suppose you have n points of interpolation data of the form (x_i, y_i), $i = 1, \dots, n$. Suppose also that the x values are ordered such that $x_i < x_{i+1}$. These x-values define a *partition* of the total interval (x_1, x_n) into $n-1$ subintervals. Authors vary on the terminology for these x_i, which are commonly referred to as the *nodes*, *breakpoints*, or *knots* of the piecewise polynomial.



(a) 4 points



(b) 10 points

Figure 3: Polynomial interpolation can result in wild oscillations when trying to interpolate too many points.

A piecewise polynomial interpolant, $s(x)$, is:

- a function of x for $x_1 \leq x \leq x_n$, (2.5)
- an interpolant, i.e. $s(x_i) = y_i$,
- a polynomial for each subinterval, (x_i, x_{i+1}), usually different on each subinterval,
- continuous for the total interval $[x_1, x_n]$.

The Matlab command `plot(x,y)`, for vector arguments `x` and `y`, creates a plot of the piecewise-linear (degree 1) interpolant of (x_i, y_i), $i = 1, \ldots, n$.

2.2.1 Cubic Spline Interpolants

This is a case in which the polynomial on each subinterval is of degree (at most) 3. Let $S(x)$ be a cubic spline with knots at x_i. Some of the defining characteristics of $S(x)$ are:

1. $S(x)$ is a piecewise polynomial interpolant (see (2.5));
2. $S(x)$ is of degree ≤ 3 between the knots; i.e., it is a cubic polynomial on each subinterval;
3. $S(x)$ is twice continuously differentiable; i.e.,

$$\frac{dS(x)}{dx} \quad \text{and} \quad \frac{d^2S(x)}{dx^2}$$

   are continuous functions for $x_1 \leq x \leq x_n$.

It would be nice if these three conditions completely defined $S(x)$. But, unfortunately, they do not. It is necessary to add additional requirements at the end points, x_1 and x_n, to completely define $S(x)$. Different applications of cubic splines will call for different choices for these end conditions, and this brings some messy technicalities to the discussion of cubic splines. For good results in applications (e.g., graphics), it is important that these details be addressed correctly.

Computing with a cubic spline, $S(x)$, involves two activities:

- computing a representation for $S(x)$ from the data (x_i, y_i);
- evaluating $S(x)$ at one (or more) values of x, i.e., evaluating $S(x)$ using the chosen representation.

2.2.2 Matlab Support for Splines

Matlab's support for interpolation computations in one variable can be thought of as being organized into three layers of increasing technical detail and complexity of use. The simpler layers use the more complex ones and hide their details from the user. It is reasonable to refer to these as 'layers' because these names are overloaded; the function that they perform depends on the form of parameter sequence, i.e. the signature of the command. This can be seen from the `help <cmd>` documentation for each.

The Matlab command `interp1` is the simplest, least specific layer. The primary command of the second layer is the `spline` command. Using different signatures, `spline` can be used to either

(a) combine the two tasks of computing the representation and evaluating S at prescribed arguments, or

(b) just compute the representation.

If the vectors `x` and `y` hold the interpolation data (i.e. the x- and y-coordinates of the interpolation points), and `xeval` holds the x-values at which one wants to evaluate the cubic spline, then task (a) is accomplished by the Matlab command

```
yeval = spline(x,y,xeval)  .
```

The vector `yeval` contains the corresponding values of $S(x)$.

Matlab calls its internal representation of a cubic spline a **ppform**. If Matlab's spline command is called as

```
pp = spline(x,y)
```

then the ppform is returned in variable 'pp'. This variable is not useful directly, but can be passed to Matlab command `ppval`, along with an argument vector at which you want to evaluate the spline, in order to get the corresponding vector of spline values.

The most complex layer is formed by the spline toolbox of Matlab. This layer has many commands which can be listed by entering `help splines`. For example, `csape` is a more sophisticated function to compute cubic splines.

2.2.3 Representation of $S(x)$

There are two basic concepts associated with the representation of cubic splines that we hope to convey. One is that continuity of $S(x)$ and its derivatives across a breakpoint x_k places conditions on the coefficients of the representations in the adjoining intervals $[x_{k-1}, x_k]$ and $[x_k, x_{k+1}]$. This is, of course, common to piecewise polynomial representation generally. The second concept is that for cubic splines these conditions lead to a system of linear equations in which the coefficients of the cubic spline representation are the unknowns. This system must be solved to find values for these coefficients. Fortunately, the system of equations has a special form (tridiagonal), enabling it to be solved very efficiently. Otherwise, cubic splines would be less practical.

A standard representation for a cubic spline is based on defining $S(x)$ as a cubic polynomial $p_i(x)$ on each subinterval $x_i \leq x \leq x_{i+1}$. Figure 4 presents a schematic of the definition.

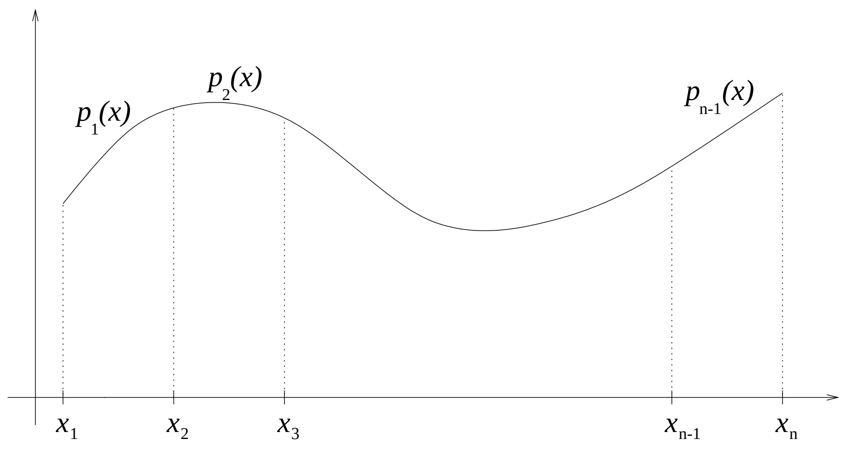


Figure 4: Schematic of piecewise definitions for a cubic spline.

The form selected for representing $p_i(x)$ is not one of the standard polynomial representations that we have discussed. Rather, it is a form which was invented specifically for cubic splines such that the coefficients in this representation can be computed and stored very efficiently. For $i = 1, \ldots, n-1$, we write

$$p_i(x) = a_{i-1}\frac{(x_{i+1}-x)^3}{6h_i} + a_i\frac{(x-x_i)^3}{6h_i} + b_i(x_{i+1}-x) + c_i(x-x_i) \tag{2.6}$$

where $h_i = x_{i+1} - x_i$ is the length of the ith subinterval.

This might suggest that the representation of $S(x)$ would require storing $3n-2$ coefficients (i.e., a_i for $i = 0, \ldots, n-1$, and b_i and c_i for $i = 1, \ldots, n-1$). However, this would be inefficient since some of the information in these coefficients duplicates information in the interpolation data (x_i, y_i), presumably also stored. By applying the interpolation conditions, it is easily shown that the coefficients b_i and c_i can be cheaply computed from a_i and the interpolation data using the formulas

$$b_i = \frac{y_i}{h_i} - \frac{a_{i-1}h_i}{6}\ , \tag{2.7}$$

$$c_i = \frac{y_{i+1}}{h_i} - \frac{a_i h_i}{6}\ . \tag{2.8}$$

Hence, only the n values of a_i need to be computed and stored, in addition to the interpolation data.

Applying the conditions which force continuity of the first derivative across the breakpoints, and specifying boundary conditions at the two end points, leads to a system of linear equations which can be expressed as an $n \times n$ tridiagonal linear system to be solved for the coefficients $a_i, i = 0, \ldots, n-1$.

Applying the conditions which force continuity of the second derivative across the breakpoints yields equations which are identities. In other words, one finds that the form (2.6) has the very special property that $p_i''(x_{i+1}) = p_{i+1}''(x_{i+1})$ for $i = 1, \ldots, n-2$, so that the continuity of the second derivative of $S(x)$ at each interior breakpoint is assured by this special form.

Exercises

1. (a) Show that $p_i''(x_{i+1}) = a_i$. Why is this not sufficient to ensure that $S(x)$ is twice continuously differentiable?

   (b) The requirement that $S(x)$ be continuous means that

$$\lim_{x \to x_i -} S(x) = \lim_{x \to x_i +} S(x) \tag{2.9}$$

   at each internal knot, i.e. for $i = 2, \ldots, n-1$. "$\lim_{x \to x_i -} S(x)$" is the limit of the values of $S(x)$ as $x \to x_i$ while $x < x_i$; it is called the limit from the left. Clearly, $\lim_{x \to x_i -} S(x) = \lim_{x \to x_i -} p_{i-1}(x)$, since if x is near x_i but $x < x_i$ then x is in the $(i-1)$st subinterval.
   Show that the continuity condition, (2.9) and the interpolation condition for $S(x)$ at $x = x_i$ require

$$\lim_{x \to x_i -} p_{i-1}(x) = y_i \tag{2.10}$$

$$\lim_{x \to x_i +} p_i(x) = y_i \; . \tag{2.11}$$

   (c) Show that (2.10) implies (2.8) and that (2.11) implies (2.7).

2. Consider the following alternative representation for a cubic spline, $S(x)$:

$$S(x) = \begin{cases} a + b(x-1) + c(x-1)^2 - \frac{1}{4}(x-1)^2(x-2) & 1 \le x \le 2 \\ e + f(x-2) + g(x-2)^2 + \frac{1}{4}(x-2)^2(x-3) & 2 \le x \le 3. \end{cases}$$

   We also wish our cubic spline to satisfy the boundary conditions:

$$\frac{d^2S}{dx^2}(1) = 0 \; , \quad \frac{d^2S}{dx^2}(3) = 0 \; .$$

   (a) What are the conditions on the coefficients a through g such that $S(x)$ interpolates the points (1,1), (2,1), and (3,0)? Deduce the values of a and e.

   (b) What is the condition on the coefficients such that $S'(x)$ is continuous at $x = 2$?

   (c) Show that enforcing the boundary conditions at $x = 1$ and $x = 3$ leads to $c = -\frac{1}{4}$ and $g = -\frac{1}{2}$.

   (d) Compute the values of b and f from part (a).

   (e) To ensure that $S(x)$ is a cubic spline, what other condition needs to be checked? (It is not necessary to actually verify this condition for the purpose of this exercise.)

3 Planar Parametric Curves

3.1 Introduction to Parametric Curves

This is a short introduction to concepts, examples, and terminology of parametric curves in the x-y plane. Parametric curves are directed curves described by a pair of continuous functions, $x(t)$ and $y(t)$, of a common argument t usually called the parameter, for $a \le t \le b$. For notational convenience, we define the vector function $\vec{P}(t) = (x(t), y(t))$. The curve C is the set of points $\{\vec{P}(t) \mid a \le t \le b\}$.

Some Examples

$$C_1\textbf{:}\ x(t) = \cos(\pi t),\ \ y(t) = \sin(\pi t),\ \ 0 \le t \le 1\,.$$

C_1 is the semi-circle in the upper half plane directed from $(1, 0)$ to $(-1, 0)$.

$$C_2\textbf{:}\ x(t) = \cos(\pi(1-t)),\ \ y(t) = \sin(\pi(1-t)),\ \ 0 \le t \le 1\,.$$

C_2 is C_1 with its direction reversed.

$$C_3\textbf{:}\ x(t) = \cos(\pi t^2),\ \ y(t) = \sin(\pi t^2),\ \ 0 \le t \le 1\,.$$

C_3 is visually the same curve as C_1, but it has a different parameterization. Mathematically, it is a different parametric curve.

$$C_4\textbf{:}\ x(t) = \cos(\pi t),\ \ y(t) = \sin(\pi t),\ \ 0 \le t \le 2\,.$$

C_4 is the entire circle; it is a closed curve, whereas the previous examples are open curves. It is a simple curve, meaning that it does not intersect itself.

$$C_5\textbf{:}\ x(t) = 1 + \tfrac{\cos(2\pi t)}{1+Kt},\ \ y(t) = .8 + .8\tfrac{\sin(2\pi t)}{1+Kt},\ \ 0 \le t \le 2\,.$$

This curve is shown in Figure 5 for a particular value of K. 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; i.e., the vector derivatives of $\vec{P}(t)$ exist for all k:

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

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

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

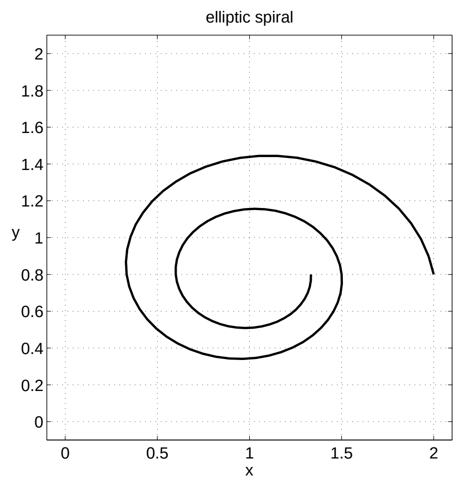


Figure 5: The parameterized curve C_5.

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.

$$\begin{aligned} x(t) = t, \ \ y(t) = 0, \ \ 0 \le t \le 1 \quad ; \\ x(t) = 1, \ \ y(t) = t - 1, \ \ 1 \le t \le 2 \quad ; \\ x(t) = 1 - (t - 2), \ \ y(t) = 1, \ \ 2 \le t \le 3 \quad ; \\ x(t) = 0, \ \ y(t) = 1 - (t - 3), \ \ 3 \le t \le 4 \quad . \end{aligned} \tag{3.1}$$

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

3.2 Interpolating Curve Data by a Parametric Curve

Suppose we are given a sequence of points $(x_i, y_i), \ i = 1, \ldots, n$, that lie on a curve in the plane. 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 for (a) is to regard the row index i as the sampled value of a real-valued parameter, t, whose range is $1 \le t \le n$; i.e., take $t_i = i$.

For step (b), we could use piecewise linear interpolation, giving a function $x_{pl}(t)$ for $0 \le t \le n$. If we did the same thing with the y coordinate data, we would get another function, $y_{pl}(t)$, for $0 \le t \le n$, and together $(x_{pl}(t), y_{pl}(t))$ define a parametric curve passing through the data points. By default, the Matlab plotting command `plot(x,y)` plots the piecewise linear interpolation curve, $(x_{pl}(t), y_{pl}(t))$.

For a smoother interpolating curve, consider using cubic spline interpolation. I.e., 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 in equation (3.1)? How well would using a cubic spline interpolating parametric curve work for this same curve?

For cubic spline interpolation, choosing the parameter values $t_i = i$ can lead to a curve that does not look the way you intended due to distortions caused by this naive choice of parameterization. An ideal parameterization would be one where the parameter values t_i reflect the arc length distance along the curve between successive data points. A good approximation to this ideal is obtained by defining successive parameter values t_i based on the Euclidean distance between points, namely

$$t_{i+1} = t_i + \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}\ . \tag{3.2}$$

3.3 Graphics for Handwriting

Handwriting can be viewed as a series of curve segments in the plane. Here we discuss creating a computer representation of handwriting based on interpolating parametric curves using cubic splines. See Appendix A for an example and a discussion of customizing plots for different purposes.

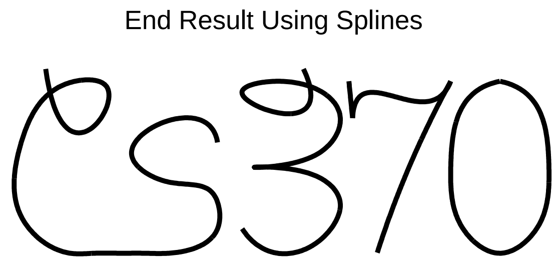
Consider the following Matlab based procedure for representing a single script letter, or a word.

(a) Identify smooth curve segments that make up the script. E.g. the hand-written form of the digit "3" normally has a non-smooth cusp in the middle. It could be broken up into 2 smooth curve segments. A scripted upper-case "C" is normally smooth enough to be a single curve segment, possibly a somewhat complex one. See Figure 6 for the construction of the course title.

(b) For each segment, capture suitable data points (x_i, y_i), $i = 1, \dots, n$ 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 linear 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. This was discussed above in subsection 3.2.

(d) Compute a cubic spline `ppform` structure in Matlab, call it `xpp`, for the interpolating cubic spline of data (t_i, x_i), and a second one, `ypp`, for the data (t_i, y_i). We refer to these two spline functions as $x_{cs}(t)$ and $y_{cs}(t)$.



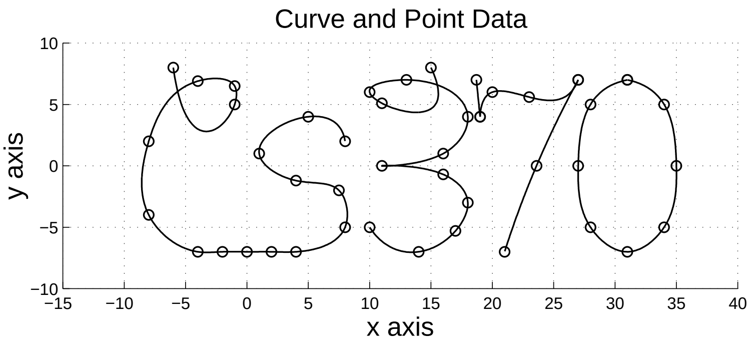


Figure 6: Constructing the CS 370 logo with splines.

(e) Compute a finer partition, $tref_j$ of the parameter interval $t_1 \leq t \leq t_n$.

(f) Create plotting points $(xref_i, yref_i)$ by evaluating the cubic splines at each parameter value in $tref_j$. i.e. $xref_i = x_{cs}(tref_i)$ and $yref_i = y_{cs}(tref_i)$ for $i = 1, \ldots, n$. Finally, plot $(xref_i, yref_i)$.

This process has been followed to produce the CS 370 logo that appears on the title page of these notes. In Figure 6, we show the result 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.

Some details

Steps (a) and (b)
A simple way to accomplish steps (a) and (b) is to write a large version of the letter (or word) on squared graph paper. Identify the segments and mark the data points on the curve segments. Introduce an x axis and a y axis on the paper and read off the coordinates of the sequence of points on each segment. For efficiency, pick a small number of data points to capture the main features of the shape. If the curve segment is closed (e.g. the letter "O"), repeat the first data point as the last one in the sequence.

Step (d)
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.

Step (e)
Below is a Matlab-like algorithm for refining a partition, `t`, by a factor of 3.

```
% refining the partition defined by row array, t, of parameter values
% creates a refined row array, tref, of parameter values
   n = length(t);
   tref = zeros(1,3*(n-1)+1);
   for k = 1:n-1
     i = 3*(k-1)+1;
     dt = t(k+1) - t(k);
     tref(i) = t(k);
     tref(i+1) = t(k)+dt/3 ;
     tref(i+2) = t(k)+2*dt/3;
   end
   tref(3*(n-1)+1) = t(n);
```

What about an arbitrary constant refinement factor (i.e. n instead of 3.)?
Can you write a fully vectorized one (i.e. no `for` loop) ?

4 Numerical Linear Algebra

4.1 From Equation Reduction to Triangular Factorization

"Gaussian elimination" for solving systems of linear equations has several meanings. Commonly, solving a linear systems of equations such as $Ax = b$ 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 (equation pivoting).

However, the standard computer techniques referred to as "Gaussian elimination" for solving $Ax = b$ are based on factoring A into triangular matrices, L and U. That is, these computer algorithms find non-singular $N \times N$ matrices L and U such that $A = LU$. After the matrix factorization, the system can be solved in two steps: first solving $Lz = b$ for z, followed by solving $Ux = z$ for x. The factorization 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. Then, solving the equivalent system $PAx = Pb$ can be done by factoring the matrix PA.

In this section, we show how these views are closely related. To do this, we discuss a common intermediate view that is often used in elementary teaching about solving linear systems of equations, the augmented matrix notation. The role here of the discussion of the augmented matrix procedure is that it enables us to move from manipulating equations as per the equation reduction view to a matrix operations description. This in turn leads to the concepts of matrix factorization.

4.1.1 Reducing a system of equations

The elementary equation reduction form of Gaussian elimination can be summarized as follows. Let $E_1, E_2, \ldots, E_N$ represent the N equations in the system of equations.

Step one:
The first step of the reduction process aims to eliminate x_1 from all the N equations except for E_1.

```
For $n \to 2$ to $N$
        replace $E_n$ by $E_n - \frac{A_{n,1}}{A_{1,1}} E_1$
End for
```

The result is a new system with

1) a subsystem of $N-1$ new equations in unknowns $(x_2, x_3, \ldots, x_N)$, and
2) a single equation involving $(x_1, x_2, \ldots, x_N)$.

The solution of the new and old systems are the same.

The next step of the reduction, which is the final step, uses the first step recursively!

a) Apply "step one" to the smaller subsystem of $N-1$ equations in (1) involving the variables $(x_2, x_3, \ldots, x_N)$. However, instead of eliminating x_1 (which is already gone), eliminate x_2.
Do this step recursively until you arrive at a system with only one equation and one unknown, x_N. Thus, x_N is known.

b) Use the known value of x_N to solve the 2-equation system from the previous recursion step. That is, solve for x_{N-1}. Work your way back through the recursion steps, and eventually solve for x_1 using $(x_2, x_3, \ldots, x_N)$.

After k invocations of the recursion of the reduction process, we have

- a subsystem of $N-k$ equations in unknowns $(x_{k+1}, \ldots, x_N)$, and
- a subsystem of k equations. In this subsystem, the first equation involves all the unknowns $(x_1, x_2, \ldots, x_N)$. For each equation j ($j = 2, \ldots, k$), the unknowns $(x_1, \ldots, x_{j-1})$ have been eliminated.

Question: What if x_1 does not appear in equation 1 (i.e $A_{1,1} = 0$)? Or if the same thing happens in trying to apply subsequent reductions?

4.1.2 Augmented Matrix Notation and Row Reduction

Matrix notation was originally introduced to avoid explicitly writing down the symbols for the unknowns $(x_1, \ldots, x_k)$. This made pencil and paper calculations faster. But matrix notation is also useful for expressing the equation reduction process in terms of matrix multiplications.

Here we define the *augmented* system matrix as the $N \times (N+1)$ matrix

$$\overline{A} = [A, b] \ .$$

Step one of the equation reduction process can be described in terms of the following row reduction operations.

```
For  $n \rightarrow 2$ to $N$
        replace row $n$ by $\left(\text{row } n - \frac{A_{n,1}}{A_{1,1}} \text{ row } 1\right)$
End for
```

This row reduction step can be expressed mathematically as a matrix multiplication by the $N \times N$ matrix $M^{(1)}$,

$$[A^{(1)}, b^{(1)}] = M^{(1)}\overline{A} \ .$$

The pattern of entries in $M^{(1)}$ is as shown in Figure 7, where each 'x' in the first column represents a non-zero number.

To pick a specific case, do you see why $M^{(1)}_{3,1} = \frac{-A_{3,1}}{A_{1,1}}$? And so in general $M^{(1)}_{k,1} = \frac{-A_{k,1}}{A_{1,1}}$ for $k = 2, \ldots, N$.

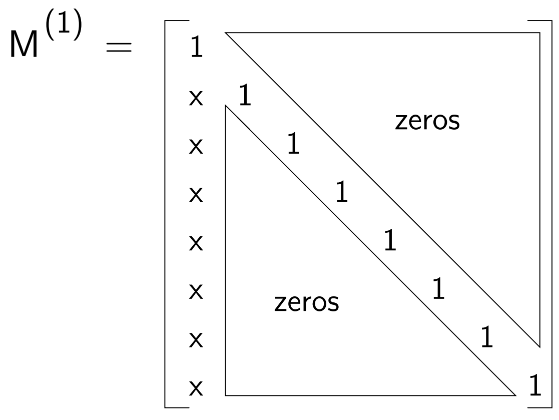


Figure 7: The matrix operator that performs the first step of the reduction.

Terminology:

"$M^{(1)}$ is **lower triangular**" means that $M^{(1)}_{j,k} = 0$ for all $k > j$

"$M^{(1)}$ is **unit diagonal**" means that $M^{(1)}_{k,k} = 1$ for all k

Some comments:

1. Recall that we identified 2 results from "step one" of the equation reduction above:
   - a subsystem of $N-1$ equations involving $(x_2, \ldots, x_N)$. The coefficient matrix for this subsystem is $A^{(1)}_{j,k}$ for $2 < j < N$ and $2 < k < N$ (an $(N-1) \times (N-1)$ submatrix of $A^{(1)}$), and the right -and side is $b^{(1)}_j$ for $2 < j < N$;
   - a single equation involving $(x_1, \ldots, x_N)$. This equation has coefficients $A^{(1)}_{1,k}$ for $1 < k < N$ (the first row of $A^{(1)}$) and right-hand side $b^{(1)}_1$.
2. x satisfies $A^{(1)}x = b^{(1)}$.
3. $M^{(1)}$ depends only on A.
4. $b^{(1)}$ is defined as $M^{(1)}b$.

A BIG IDEA

The kth step of the row reduction process can be described as a matrix multiplication in essentially the same way. It can be expressed using the $N \times N$ matrix $M^{(k)}$ with the pattern shown in Figure 8, for $k = 2, 3, \ldots, N-1$.

$$M^{(k)}[A^{(k-1)}, b^{(k-1)}] = [A^{(k)}, b^{(k)}] \; . \tag{4.1}$$

Note that the comments made above for the first step of equation reduction have their analogs for the kth step, and also note that we have assumed $A^{(k-1)}_{k,k} \neq 0$.

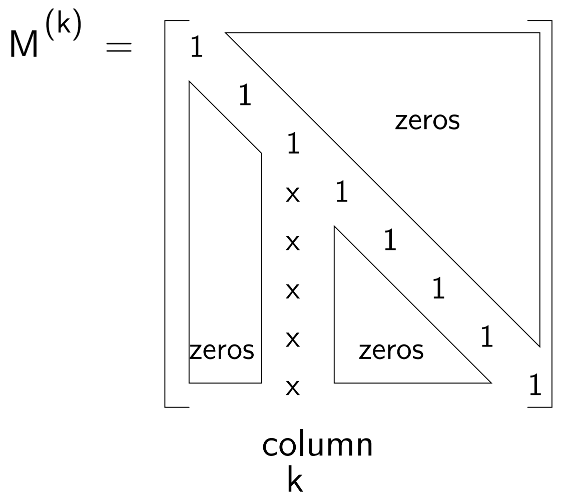


Figure 8: The matrix operator that performs the kth step of the reduction.

When $k = N - 1$,

- the 'subsystem' of equations actually has only one equation, i.e. $A^{(N-1)}_{N,N} x_N = b^{(N-1)}_N$;
- $A^{(N-1)}$ is upper triangular;
- we can solve $A^{(N-1)}x = b^{(N-1)}$ for $x_N, x_{N-1}, \ldots, x_1$ (in that order).

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 new formulation to describe some algebraic facts about solving $Ax = b$. Here are three exercises, the first two of which are useful in this discussion.

Exercises

1. If both B and C are lower triangular and unit diagonal, then so is the product BC.
2. If M is lower triangular and unit diagonal, then so is M^{-1}.
3. Assuming that $A^{(k-1)}_{k,k} \neq 0$

$$M^{(k)}_{j,j} = 1, \ \ 1 \leq j \leq N \tag{4.2}$$

$$M^{(k)}_{k,n} = \frac{-A^{(k-1)}_{k,n}}{A^{(k-1)}_{k,k}}, \ \ k+1 \leq n \leq N \tag{4.3}$$

$$M^{(k)}_{j,n} = 0, \ \ \text{other values of } j, n. \tag{4.4}$$

Define $U = A^{(N-1)}$; this is the standard designation of this upper triangular matrix in the context of Gaussian elimination.

The matrix description of the kth reduction step shown in (4.1) can be broken into two steps:

$$M^{(k)}A^{(k-1)} = A^{(k)} \text{ and } M^{(k)}b^{(k-1)} = b^{(k)} . \tag{4.5}$$

The first of these implies

$$A^{(k-1)} = (M^{(k)})^{-1}A^{(k)} , \tag{4.6}$$

and the second implies

$$b^{(k-1)} = (M^{(k)})^{-1}b^{(k)} . \tag{4.7}$$

If we look at (4.6) for $k = N-1$, we see that $A^{(N-2)} = \left(M^{(N-1)}\right)^{-1} A^{(N-1)}$. In general,

$$A^{(N-k)} = \left(M^{(N-k+1)}\right)^{-1} \left(M^{(N-k+2)}\right)^{-1} \cdots \left(M^{(N-1)}\right)^{-1} A^{(N-1)}. \tag{4.8}$$

Each matrix inverse, $\left(M^{(N-j)}\right)^{-1}$, $j = 1, \ldots, k-1$, in (4.8) is lower triangular and unit diagonal (see Exercise 2 above). Moreover, the product of these matrices is also lower triangular and unit diagonal (see Exercise 1 above). So, setting $k = N-1$, and defining the matrix J as the product of the inverse matrices in (4.8), and setting $A^{(N-1)} = U$, we have

$$A^{(1)} = JU \tag{4.9}$$

where J is a lower triangular and unit diagonal matrix, and U is an upper triangular matrix. Finally, since

$$A = \left(M^{(1)}\right)^{-1} A^{(1)} = \left(M^{(1)}\right)^{-1} J\,U , \tag{4.10}$$

by defining L as the product $\left(M^{(1)}\right)^{-1} J$, the matrix of coefficients of the original system of equations can be written as a product of two factors,

$$A = LU , \tag{4.11}$$

where L is a lower triangular, unit diagonal matrix, and U is an upper triangular matrix.

If we performed the same development on (4.7), we could conclude that

$$b = Lb^{(N-1)} . \tag{4.12}$$

Then, we can get the solution, x, by solving

$$Ux = b^{(N-1)} . \tag{4.13}$$

This triangular factorization is the matrix view of Gaussian elimination. By factoring the system matrix A into upper and lower triangular factors, the solution can easily be extracted.

4.1.3 Matrix factorization form of Gaussian elimination

This view of Gaussian elimination has the following major steps:

1. Compute triangular factors L and U so that $LU = A$.
2. Solve $Lz = b$ (see (4.12) and identify z with $b^{(N-1)}$).
3. Solve $Ux = z$.

What we have shown in the preceding sections is that these steps are not only possible, but are, in fact, closely related to what is done by the elementary equation reduction process. The next section presents and analyzes algorithms for the computations of these three steps.

4.2 LU Factorization

Consider an $N \times N$ matrix A, a solution vector x and a right hand side vector b. We wish to solve

$$Ax = b\ . \tag{4.14}$$

The preceding section gave an overview of how the triangular factorization of A is related to the row reduction process, but not the details of how to carry out the computation. Here we present the algorithm for factoring $A = LU$, where L is unit lower triangular, and U is upper triangular.

```
Algorithm: LU Factorization
    Given an $N \times N$ matrix $A = a_{ij}$
    For $k = 1, \ldots, N$
        For $i = k+1, \ldots, N$
            $mult := a_{ik}/a_{kk}$
            $a_{ik} := mult$
            For $j = k+1, \ldots, N$
                $a_{ij} := a_{ij} - mult * a_{kj}$
            EndFor
        EndFor
    EndFor
```

Note that in the above LU factorization algorithm, the original entries of A are overwritten by L and U. 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. This convention is common among implementations of the LU factorization algorithm because it requires no extra storage.

To solve equation (4.14), we note that

$$Ax = LUx \ = \ b\ . \tag{4.15}$$

If we define $z = Ux$, then it follows from equation (4.15) that

$$\begin{aligned} Lz &= b \,, & (4.16) \\ Ux &= z \,. & (4.17) \end{aligned}$$

Equations (4.16) and (4.17) are easy to solve. Let the entries of $(L)_{ij} = l_{ij}$, where we recall that L is unit lower triangular. We can easily solve $Lz = b$ for z using the forward solve algorithm:

```
Algorithm: Forward Solve
    For $i = 1, \ldots, N$
        $z_i := b_i$
        For $j = 1, \ldots, i-1$
            $z_i := z_i - l_{ij} * z_j$
        EndFor
    EndFor
```

Let the entries of $(U)_{ij} = u_{ij}$. We can easily solve $Ux = z$ for x using the back solve algorithm:

```
Algorithm: Back Solve
    For $i = N, \ldots, 1$
        $x_i := z_i$
        For $j = i+1, \ldots, N$
            $x_i := x_i - u_{ij} * x_j$
        EndFor
        $x_i := x_i / u_{ii}$
    EndFor
```

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 $A = LU$?

4.2.1 How do we count work?

The running time of the above algorithms will be proportional to the number of flops. Notice that most statements consist of *linked triad* operations such as

$$z_i := z_i - l_{ij} * z_j \ .$$

That is, usually 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 Matlab convention, which is to count 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:

$$\begin{aligned} \text{LU Factorization Work} &= \frac{2N^3}{3} + O(N^2) \text{ flops} \\ &= O(N^3) \text{ flops.} \end{aligned} \tag{4.18}$$

The flop count for the Forward Solve plus Back Solve is:

$$\begin{aligned} \text{Forward + Back Solve Work} &= 2N^2 + O(N) \\ &= O(N^2). \end{aligned} \tag{4.19}$$

Obviously, for large N, the factorization process is more expensive than the forward/back solve.

4.2.2 Pivoting and the Stability of Factorization

Each of the closely related views of Gaussian elimination given in the opening section can fail if a division by zero is encountered at some stage. This problem can be sidestepped 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 0s and 1s. Each row of a permutation matrix contains exactly one 1 (all the other elements are 0s). The same is true for each column of a permutation matrix. For example, the permutation matrix that interchanges the second and third rows of a 3×3 matrix is

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} .$$

Try multiplying a 3×3 matrix by P to see how it works.

In the LU factorization algorithm, if $a_{kk}^{(k)} = 0$ at some stage, then we examine all entries in the kth column below $a_{kk}^{(k)}$ to find the element with the largest absolute value. Suppose $|a_{\tau k}^{(k)}|$ is the largest, i.e.

$$\max_{j=k,..,N} |a_{jk}^{(k)}| = |a_{\tau k}^{(k)}| . \tag{4.20}$$

Then we swap row τ with row k, and use $a_{\tau k}^{(k)}$ to form the multiplier. This process is called *pivoting.* Note that at least one of $a_{kk}^{(k)}, a_{k+1,k}^{(k)}, \ldots, a_{Nk}^{(k)}$ must be nonzero, otherwise the matrix is singular. Row-swapping produces a re-ordering of the rows of A that could be described by a permutation matrix, P. This same re-ordering is applied to the right-hand side vector b. As a result, we get the new system of equations, $PAx = Pb$, having the same solution, x, as the original system, $Ax = b$.

Algorithmically, the process of pivoting is intertwined with the factorization to produce P, L, and U from A.

Standard computer oriented algorithms for solving $Ax = b$ have two stages:

(a) from A, compute P, L and U such that $PA = LU$;

(b) from P, L, U and b, compute x.

The amount of work required for stage (a) is $\frac{2N^3}{3} + O(N^2)$ flops, while stage (b) requires $2N^2 + O(N)$ flops.

In Matlab, stage (a) can be done using the `lu(A)` command:

```
[L,U,P] = lu(A) .
```

4.2.3 Stability

So far, we have been discussing the factorization algorithm as it operates using mathematically exact arithmetic. What happens when we implement the algorithm in the inexact arithmetic of a floating point number system? Could errors build up during the $O(N^3)$ arithmetic operations of the factorization?

The following is a general rule of thumb concerning the behaviour of algorithms implemented in inexact (floating point) arithmetic.

> *If an algorithm fails under some condition when exact arithmetic is used, then, if the algorithm is implemented using inexact arithmetic, it can generate large errors when the failure condition is approximately true.*

The algorithm of interest here is the LU factorization of matrix A. Using exact arithmetic, it can fail if $a_{kk}^{(k)} = 0$ at some stage. The rule of thumb says that a program to implement it in floating point arithmetic can generate large errors if $|a_{kk}^{(k)}|$ is very small. (So what constitutes being small?)

However, if we add row pivoting to the LU factorization algorithm, then, in exact arithmetic, the resulting algorithm only fails if the row pivoting fails to find a non-zero entry anywhere **on or below** the diagonal in column k of $A^{(k)}$. So we can expect the factorization computed in floating point to be reasonably accurate unless **all** the entries of $A^{(k)}$ on or below the diagonal in column k are small, not just $A_{kk}^{(k)}$ itself.

A measure of the stability of factorization is

$$\rho = \max_{i,j,k} |a_{ij}^{(k)}|, \tag{4.21}$$

i.e. if the size of the maximum entry in $A^{(k)}$ becomes very large during the course of elimination, then this indicates that small pivots are encountered.

4.3 Matrix and Vector Norms

Before we continue, we give a short overview of matrix and vector norms. The norm of a vector is a measure of its size. Let x be the vector

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} . \tag{4.22}$$

Then we define the **1-norm**, **2-norm**, and **∞-norm** as

$$\begin{aligned} \|x\|_1 &= \sum_{i=1}^{n} |x_i| \\ \|x\|_2 &= \left(\sum_{i=1}^{n} x_i^2 \right)^{1/2} \\ \|x\|_\infty &= \max_i |x_i| \; . \end{aligned}$$

Usually, these types of norms are referred to as *p-norms*

$$\|x\|_p \quad \text{where} \quad p = 1, 2, \infty \; . \tag{4.23}$$

Matlab computes these vector norms using the `norm` command.

Properties of Norms

$$\begin{aligned} \|x\| &= 0 \ \text{ iff } \ x_i = 0 \ \ \forall i \\ \|\alpha x\| &= |\alpha| \, \|x\| \; , \quad \alpha = \text{ scalar} \\ \|x + y\| &\leq \|x\| + \|y\| \end{aligned}$$

Matrix Norms

We define a matrix norm for an $n \times n$ matrix A corresponding to a given vector norm, as follows:

$$\|A\|_p = \max_{\|x\| \neq 0} \frac{\|Ax\|_p}{\|x\|_p}$$

for any $\| \cdot \|_p$ 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 $\| \cdot \|_2$ 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

$$\|A\|_2 = \max_i |\lambda_i|^{1/2}.$$

Matlab also computes these matrix norms via the `norm` command.

Note that for any $n \times n$ matrices A and B, and any n-vector x, we have

$$\begin{aligned}
\|A\| &= 0 \text{ iff } a_{ij} = 0 \ \forall i, j \\
\|\alpha A\| &= |\alpha| \, \|A\| , \quad \alpha = \text{scalar} \\
\|A + B\| &\leq \|A\| + \|B\| \\
\|Ax\| &\leq \|A\| \, \|x\| \\
\|AB\| &\leq \|A\| \, \|B\| \\
\|I\| &= 1 , \ I = \text{ identity matrix.}
\end{aligned}$$

4.4 Condition Numbers

4.4.1 Conditioning

Suppose we perturb the right-hand-side 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

$$A(x + \Delta x) = b + \Delta b. \tag{4.24}$$

Since if x is the exact solution, then $Ax = b$, so that equation (4.24) becomes

$$\Delta x = A^{-1} \Delta b. \tag{4.25}$$

Noting that $Ax = b$, so that

$$\|b\| \leq \|A\| \|x\|, \tag{4.26}$$

and from equation (4.25) we obtain

$$\|\Delta x\| \leq \|A^{-1}\| \|\Delta b\|. \tag{4.27}$$

Now, equation (4.26) gives

$$\|x\| \geq \frac{\|b\|}{\|A\|} \quad \text{or} \quad \frac{\|A\|}{\|b\|} \geq \frac{1}{\|x\|}. \tag{4.28}$$

Equations (4.27) and (4.28) give

$$\frac{\|\Delta x\|}{\|x\|} \leq \|A\|\,\|A^{-1}\|\frac{\|\Delta b\|}{\|b\|}. \tag{4.29}$$

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 (4.29) says that

$$\text{relative change in } x \;\leq\; \text{condition number } \times \text{ relative change in } b.$$

If $\kappa(A)$ is large, then the problem *may* be ill-conditioned; since the bound is not very sharp, $\kappa(A)$ 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.

$$(A + \Delta A)(x + \Delta x) = b, \tag{4.30}$$

then, assuming $Ax = b$, i.e. x is the exact solution, equation (4.30) implies

$$\begin{aligned} A\Delta x &= -\Delta A(x + \Delta x) \\ \Delta x &= -A^{-1}\left[\Delta A(x + \Delta x)\right], \end{aligned} \tag{4.31}$$

which then gives

$$\|\Delta x\| \leq \|A^{-1}\|\|\Delta A\|\|x + \Delta x\|, \tag{4.32}$$

or

$$\begin{aligned} \frac{\|\Delta x\|}{\|x + \Delta x\|} &\leq \|A^{-1}\|\|A\|\frac{\|\Delta A\|}{\|A\|} \\ &= \kappa(A)\frac{\|\Delta A\|}{\|A\|}. \end{aligned} \tag{4.33}$$

Once again the condition number appears. So, a measure of the sensitivity to changes in A or b is the condition number. Note that

- $\kappa(A) \geq 1$;
- $\kappa(\alpha A) = \kappa(A), \alpha = \text{scalar}$.

4.4.2 The residual

The accuracy of the computed solution is sometimes measured by the size of the *residual* which is defined as follows. If the computed solution for the system $Ax = b$ is $x + \Delta x$ (it is not exact) then the residual is

$$r = b - A(x + \Delta x)\,. \tag{4.34}$$

If the residual $r = 0$ then $\Delta x = 0$ and we have the exact solution. From equation (4.34) we have

$$A(x + \Delta x) \;=\; b - r\,.$$

Similar analysis as in equations (4.24-4.29) gives

$$\frac{\|\Delta x\|}{\|x\|} \;\le\; \kappa(A)\frac{\|r\|}{\|b\|}\,. \tag{4.35}$$

Thus a small residual does not necessarily imply a small error, if $\kappa(A)$ is large. For example, it is unavoidable that

$$\frac{\|r\|}{\|b\|} \;\simeq\; \varepsilon_{\text{machine}} \tag{4.36}$$

in which case equation (4.35) implies

$$\frac{\|\Delta x\|}{\|x\|} \;\le\; \kappa(A)\,\varepsilon_{\text{machine}}$$

which may not be small if the problem is poorly conditioned.

For implementation in floating point arithmetic, Gaussian elimination with pivoting is known to be a stable method. Moreover, it yields a computed result whose residual is small, namely, the residual is approximately the size indicated by equation (4.36) in most cases. For this reason, it is a commonly used method.

Another type of analysis known as "backward error analysis" yields the following result. Gaussian elimination with pivoting produces a computed solution $\hat{x}$ which satisfies

$$(A + E)\hat{x} \;=\; b$$

where $\|E\| = \varepsilon_{\text{machine}}\,\|A\|$. In other words, Gaussian elimination with pivoting solves a *nearby* problem exactly. From equation (4.33) we have

$$\frac{\|x - \hat{x}\|}{\|\hat{x}\|} \;\le\; \kappa(A)\,\varepsilon_{\text{machine}}\,.$$

As stated before, the condition number is a property of the problem and not a property of the algorithm used to solve the problem. We conclude that the result computed in a floating point number system via Gaussian elimination with pivoting is about as accurate as could ever be achieved. However, the computed solution may not be close to the true solution if the problem has a large condition number.

5 Google Page Rank

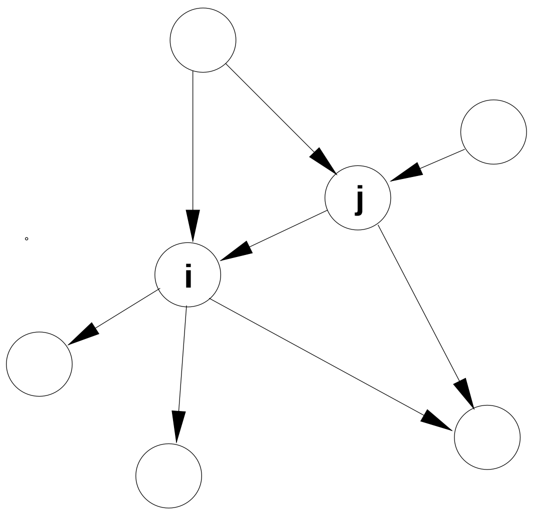
It is well known that various search engines use different techniques in order to give their "best" ranking for a given search query. This implies that a given search query will often give different results for different search engines. One can see this simply by running a few search examples on msn.com, 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.

5.1 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 (5.1). We assume that there are R pages in the web. There are at least two major problems with algorithm (5.1). Can you spot the problems?

Random Surfer Algorithm

```
$Rank(m) = 0 \ , \ m = 1, ..., R$
For $m = 1, ..., R$
    $j = m$
    For $k = 1, ..., K$
        $Rank(j) = Rank(j) + 1$
        Randomly select outlink $l$ of page $j$
        $j = l$
    EndFor
EndFor
$Rank(m) = Rank(m)/(K * R) \ , \ m = 1, ..., R$
```
(5.1)

If K in algorithm (5.1) 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.

5.2 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

$$\begin{aligned} P_{ij} &= \frac{1}{deg(j)} \ , \ \text{ if there is a link } j \rightarrow i \\ &= 0 \ , \ \text{ otherwise.} \end{aligned} \tag{5.2}$$

There are, however, two problems associated with letting the random surfer's movements be governed with the probability matrix P as it stands.

5.2.1 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

$$\begin{aligned} [\mathbf{d}]_i &= 1 \ , \ \text{if } deg(i) = 0 \\ &= 0 \ , \ \text{otherwise} \end{aligned} \tag{5.3}$$

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:

$$P' = P + \frac{1}{R}\mathbf{e} \cdot \mathbf{d}^t \ . \tag{5.4}$$

5.2.2 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

$$M = \alpha P' + (1 - \alpha) \cdot \frac{1}{R}\mathbf{e} \cdot \mathbf{e}^t \ . \tag{5.5}$$

What does equation (5.5) 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

$$\sum_i M_{ij} = 1 \ ; \tag{5.6}$$

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 (5.5) is referred to as the *Google Matrix*.

5.3 Markov Transition Matrices

Let us be a bit more formal here.

Definition 5.1 *A matrix Q is a Markov matrix if*

$$0 \le Q_{ij} \le 1 \quad \textit{and} \sum_i Q_{ij} = 1 \quad . \tag{5.7}$$

Obviously, matrix M in equation (5.5) 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

$$[\mathbf{p}]_i = \frac{1}{R} \ , \ \forall i \tag{5.8}$$

that is, the random surfer visits each page initially with equal probability.

Definition 5.2 *A vector $\mathbf{q}$ is a probability vector if*

$$0 \le [\mathbf{q}]_i \le 1 \quad \textit{and} \sum_i [\mathbf{q}]_i = 1 \quad . \tag{5.9}$$

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

$$\mathbf{p}^{n+1} = M\mathbf{p}^n \ . \tag{5.10}$$

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

$$[\mathbf{p}^{n+1}]_i \ge 0 \ , \ \forall i \tag{5.11}$$

and as well

$$\begin{aligned} \sum_i [\mathbf{p}^{n+1}]_i &= \sum_i \sum_j M_{ij} [\mathbf{p}^n]_j \\ &= \sum_j [\mathbf{p}^n]_j \sum_i M_{ij} \\ &= \sum_j [\mathbf{p}^n]_j \\ &= 1 \ . \end{aligned} \tag{5.12}$$

Thus $\mathbf{p}^n$ a probability vector implies then $\mathbf{p}^{n+1}$ is also a probability vector for each n.

5.4 Page Rank

Given the assumption above, we can now precisely state the algorithm for determining the ranking of each page in the Web. Let

$$\mathbf{p}^0 = \frac{1}{R}\mathbf{e} \tag{5.13}$$

(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

$$\mathbf{p}^\infty = \lim_{k\to\infty}(M)^k\mathbf{p}^0 \ . \tag{5.14}$$

So, essentially, we start with $\mathbf{p}^0$ defined by equation (5.13) and then repeatedly multiply by M, to give us $\mathbf{p}^\infty$. There are two obvious questions

- Does iteration (5.14) converge?
- If yes, how fast does this iteration converge?

5.5 Convergence Analysis

In order to analyze the convergence of iteration (5.14), we have to review some basic linear algebra. (Maybe you haven't seen this before, but its easy). Suppose we have a special vector $\mathbf{x}$ such that

$$Q\mathbf{x} = \lambda\mathbf{x} \tag{5.15}$$

where λ is a (possibly complex) nonzero scalar. The special vector $\mathbf{x}$ is an *eigenvector* of the matrix Q, with *eigenvalue* λ.

Note that equation (5.15) 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.

5.5.1 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 5.5.2.

Theorem 5.3 *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 5.4 *Every (possibly complex) eigenvalue λ of a Markov matrix Q satisfies*

$$|\lambda| \leq 1 \ . \tag{5.16}$$

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 optimization texts) applied to the eigenvalues of Q^t. Since the eigenvalues of Q and Q^t are the same, the theorem follows.

Definition 5.5 *A Markov matrix Q is a positive Markov matrix if*

$$Q_{ij} > 0 \quad , \quad \forall i, j \quad . \tag{5.17}$$

Theorem 5.6 *If Q is a positive Markov matrix, then there is only one eigenvector of Q with $|\lambda| = 1$.*

Proof: See, for example, (Grimmett and Stirzaker, *Probability and Random Processes*, Oxford University Press, 1989.)

5.5.2 Convergence Proof

Theorem 5.7 *If M is a positive Markov matrix, the iteration (5.14) 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

$$\mathbf{p}^0 = \sum_l c_l \mathbf{x}_l \tag{5.18}$$

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 with $\lambda_1 = 1$. Then

$$(M)^k \mathbf{p}^0 = c_1 \mathbf{x}_1 + \sum_{l=2}^{R} c_l (\lambda_l)^k \mathbf{x}_l \ . \tag{5.19}$$

From Theorem (5.6), we have that $|\lambda_l| < 1$ for all $l > 1$, so that

$$\lim_{k \to \infty} (M)^k \mathbf{p}^0 = c_1 \mathbf{x}_1 \tag{5.20}$$

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

$$\mathbf{q}^0 = \sum_l b_l \mathbf{x}_l \tag{5.21}$$

for some coefficients b_l, then

$$\mathbf{q}^\infty = \lim_{k \to \infty} (M)^k \mathbf{q}^0 = b_1 \mathbf{x}_1 \ . \tag{5.22}$$

But, for given $\mathbf{x}_1$, since $\mathbf{q}^\infty, \mathbf{p}^\infty$ are probability vectors, we have $b_1 = c_1$.

Remark 5.8 *The speed of convergence of algorithm (5.14) 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 (5.14) for any starting vector should give reasonably accurate estimates for $\mathbf{p}^\infty$.*

The Web is estimated to contain billions of pages. Using algorithm (5.14) 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 its 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 (5.14). However small values of α result in less significant ranking information.

5.6 Practicalities

If we assume that $\mathbf{p}^n$ is a probability vector then

$$\begin{aligned} \frac{\mathbf{e}\mathbf{e}^t}{R}\mathbf{p}^n &= \frac{1}{R}\mathbf{e}(\mathbf{e}^t\mathbf{p}^n) \\ &= \frac{\mathbf{e}}{R} \end{aligned} \tag{5.23}$$

so that

$$M\mathbf{p}^n = \alpha P'\mathbf{p}^n + (1-\alpha)\frac{\mathbf{e}}{R} \tag{5.24}$$

and that

$$\begin{aligned} P'\mathbf{p}^n &= (P + \frac{\mathbf{e}\cdot\mathbf{d}^t}{R})\mathbf{p}^n \\ &= P\mathbf{p}^n + \mathbf{e}(\frac{\mathbf{d}^t\mathbf{p}^n}{R}) \ . \end{aligned} \tag{5.25}$$

Putting these two steps together gives

$$M\mathbf{p}^n = \alpha(P\mathbf{p}^n + \mathbf{e}(\frac{\mathbf{d}^t\mathbf{p}^n}{R})) + (1-\alpha)\frac{\mathbf{e}}{R} \ . \tag{5.26}$$

Typically, P is quite sparse, so that even though M is dense, we can perform the matrix-vector multiply $M\mathbf{p}^n$ in $O(R)$ operations.

5.7 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 (5.27) below.

Page Rank Algorithm

```
$\mathbf{p}^0 = \mathbf{e}/R$
For $k = 1, ...,$ until converged
    $\mathbf{p}^k = M\mathbf{p}^{k-1}$
    If $\max_i |[\mathbf{p}^k]_i - [\mathbf{p}^{k-1}]_i| < tol$ then quit
EndFor
```
(5.27)

5.7.1 Some Other Interesting Points

Instead of defining M as

$$M = \alpha P' + (1-\alpha)\mathbf{e}\mathbf{e}^t/R \ , \tag{5.28}$$

Google actually uses

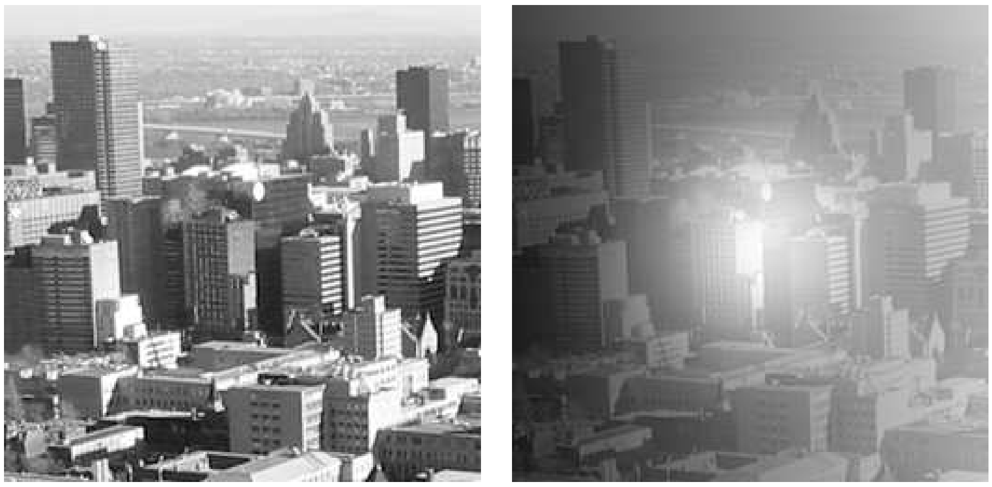
$$M = \alpha P' + (1-\alpha)\mathbf{e}\mathbf{v}^t \tag{5.29}$$

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 (5.27). In mathematical terms, algorithm (5.27) is known as the power method for finding the eigenvector of M corresponding to the largest eigenvalue.

6 Least Squares Problems

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 9. Not all is



(a) Original (b) Over-exposed

Figure 9: Original and Over-exposed pictures of Montreal

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 10 shows a model in which the observed photo can be decomposed into three parts: the true photo, a centered exposure artifact, and a left-right exposure artifact.

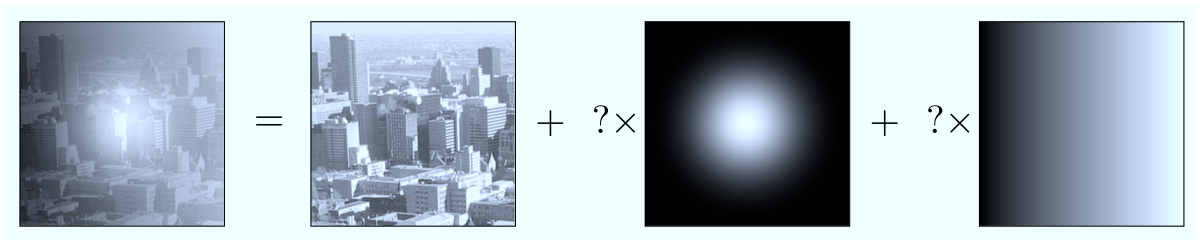


Figure 10: Decomposition of over-exposed photo. Although we know the form of the exposure artifacts, the question-marks indicate that we do not know how strong each artifact is.

[1]In image processing, an "artifact" is an unwanted feature in an image, usually the result of some sort of problem or inaccuracy.

6.1 Least Squares Fitting

Figure 10 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 four components, illustrated in Figure 11.

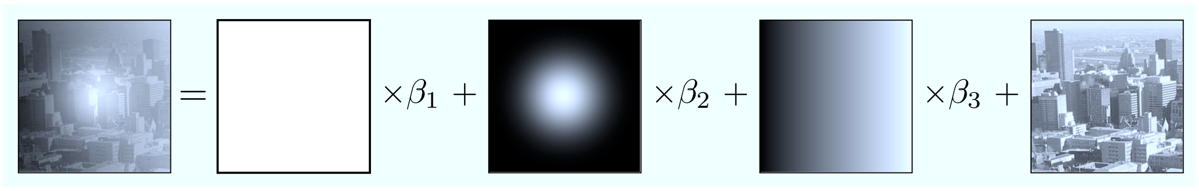


Figure 11: Decomposition of over-exposed photo into four components.

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[2], as in Figure 12. Using the same method, we can convert each of the

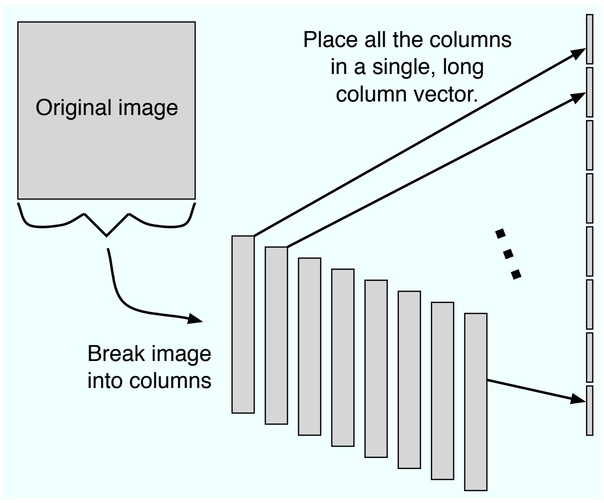


Figure 12: Representing an image as a column vector

components in Figure 11 to vectors, producing four column vectors. Then, Figure 11 can be written as a vector equation,

$$\begin{aligned} y &= a_1\beta_1 + a_2\beta_2 + a_3\beta_3 + \epsilon && (6.1) \\ y &= \mathbf{A}\beta + \epsilon \ , && (6.2) \end{aligned}$$

[2]This is Matlab's ordering of matrix data. However, the specific ordering does not matter, as long as we are consistent.

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 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,

$$r = \begin{bmatrix} r_1 \\ \vdots \\ r_m \end{bmatrix} = y - \mathbf{A}\beta \tag{6.3}$$

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 (6.3) to (6.2)). 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 (6.3), it might look like Figure 13. 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.

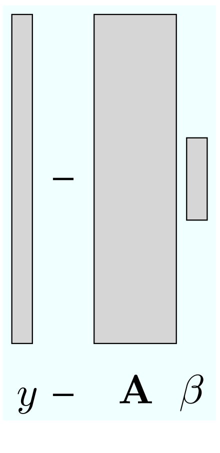


Figure 13: Illustration of the matrix expression for calculating the residual vector.

6.2 Total squared error

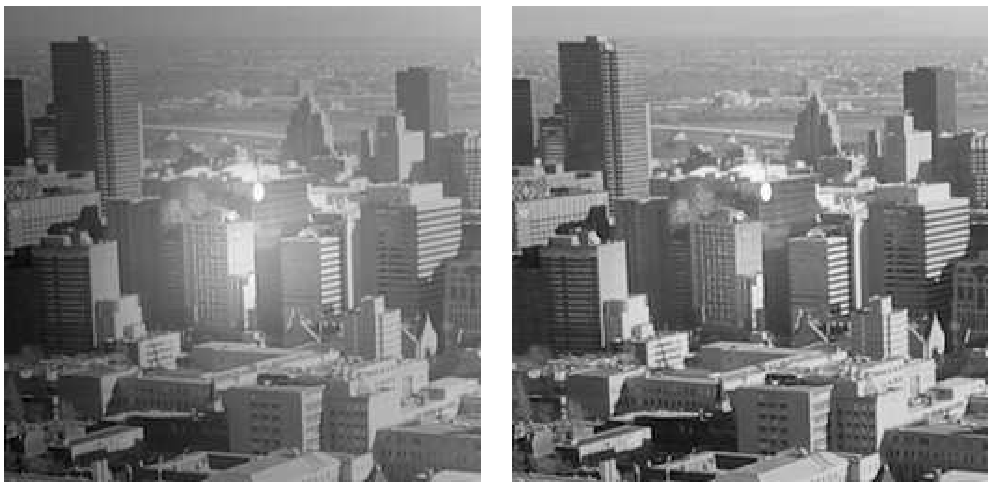
A useful summary of the errors associated with the given choice of β is the square of the (Euclidean) length of r. We will call this measure the **total squared error**, and denote it

$$E(\beta) = \sum_{i=1}^{m} r_i^2 \ . \tag{6.4}$$

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

$$E(\beta) = (y - \mathbf{A}\beta)^t \, (y - \mathbf{A}\beta) \ . \tag{6.5}$$

Figure 14 shows the results when choosing particular parameter values. Figure 14(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 14(b) corresponds to $\beta = [163.4 \ 1.9 \ 0.7]^t$, resulting in a total squared error of 14,261. Figure 14(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).



(a) $E(163.4, 1.5, 0.5) = 29,352$ (b) $E(163.4, 1.9, 0.7) = 14,261$

Figure 14: 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 $E(\beta) = 14,261$.

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 14? 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. I.e. $E(\bar{\beta}) \leq E(\beta)$ 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 10, the least squares fit parameters are $\bar{\beta} = [163.4 \ 2.047 \ 0.7127]^t$ and give a total squared error of $E(\bar{\beta}) = 13,005$.

6.3 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 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.

In calculus, if you want to find the minimum value of a function $f(x)$, you take the derivative and set it equal to zero. The same methodology applies to higher-dimensional functions, such as our total squared error function, $E(\beta)$. The definition of the (directional) derivative of $E(\beta)$ is

$$\frac{\partial E}{\partial \beta} = \lim_{e \to 0} \frac{E(\beta + e) - E(\beta)}{\|e\|} \ . \tag{6.6}$$

The numerator of (6.6) can be written

$$\begin{aligned}
& E(\beta + e) - E(\beta) \\
= \ & (y - \mathbf{A}(\beta + e))^t \, (y - \mathbf{A}(\beta + e)) - (y - \mathbf{A}\beta)^t \, (y - \mathbf{A}\beta) \\
= \ & \left(y^t - (\beta + e)^t \mathbf{A}^t\right) (y - \mathbf{A}(\beta + e)) - \left(y^t - \beta^t \mathbf{A}^t\right) (y - \mathbf{A}\beta) \\
= \ & y^t y - y^t \mathbf{A}(\beta + e) - (\beta + e)^t \mathbf{A}^t y + (\beta + e)^t \mathbf{A}^t \mathbf{A}(\beta + e) \\
& \quad - y^t y + y^t \mathbf{A}\beta + \beta^t \mathbf{A}^t y - \beta^t \mathbf{A}^t \mathbf{A}\beta \\
= \ & -y^t \mathbf{A}(\beta + e) - (\beta + e)^t \mathbf{A}^t y + (\beta + e)^t \mathbf{A}^t \mathbf{A}(\beta + e) + y^t \mathbf{A}\beta + \beta^t \mathbf{A}^t y - \beta^t \mathbf{A}^t \mathbf{A}\beta \\
= \ & -y^t \mathbf{A}\beta - y^t \mathbf{A}e - \beta^t \mathbf{A}^t y - e^t \mathbf{A}^t y + (\beta + e)^t \mathbf{A}^t \mathbf{A}(\beta + e) + y^t \mathbf{A}\beta + \beta^t \mathbf{A}^t y - \beta^t \mathbf{A}^t \mathbf{A}\beta \\
= \ & -y^t \mathbf{A}e - e^t \mathbf{A}^t y + (\beta + e)^t \mathbf{A}^t \mathbf{A}(\beta + e) - \beta^t \mathbf{A}^t \mathbf{A}\beta \\
= \ & -2e^t \mathbf{A}^t y + (\beta + e)^t \mathbf{A}^t \mathbf{A}(\beta + e) - \beta^t \mathbf{A}^t \mathbf{A}\beta \quad (\text{since } y^t \mathbf{A}e = e^t \mathbf{A}^t y) \\
= \ & -2e^t \mathbf{A}^t y + \beta^t \mathbf{A}^t \mathbf{A}\beta + \beta^t \mathbf{A}^t \mathbf{A}e + e^t \mathbf{A}^t \mathbf{A}\beta + e^t \mathbf{A}^t \mathbf{A}e - \beta^t \mathbf{A}^t \mathbf{A}\beta \\
= \ & -2e^t \mathbf{A}^t y + 2\beta^t \mathbf{A}^t \mathbf{A}e + e^t \mathbf{A}^t \mathbf{A}e \quad (\text{since } \beta^t \mathbf{A}^t \mathbf{A}e + e^t \mathbf{A}^t \mathbf{A}\beta) \\
= \ & 2e^t \left(\mathbf{A}^t \mathbf{A}\beta - \mathbf{A}^t y\right) + e^t \mathbf{A}^t \mathbf{A}e \ .
\end{aligned}$$

Phew! Now we can write (6.6) as

$$\begin{aligned}
\frac{\partial E}{\partial \beta} &= \lim_{e \to 0} \frac{2e^t \left(\mathbf{A}^t \mathbf{A}\beta - \mathbf{A}^t y\right) + e^t \mathbf{A}^t \mathbf{A}e}{\|e\|} && (6.7) \\
&= \lim_{e \to 0} \frac{2e^t \left(\mathbf{A}^t \mathbf{A}\beta - \mathbf{A}^t y\right)}{\|e\|} + \lim_{e \to 0} \frac{e^t \mathbf{A}^t \mathbf{A}e}{\|e\|} && (6.8) \\
&= 0 && (6.9)
\end{aligned}$$

The second limit in (6.8) is zero, since

$$0 \ \le \ \lim_{e \to 0} \frac{e^t \mathbf{A}^t \mathbf{A}e}{\|e\|} \ \le \ \lim_{e \to 0} \frac{\|e\|^2 \|\mathbf{A}\|^2}{\|e\|} \ = \ \|\mathbf{A}\|^2 \ \lim_{e \to 0} \|e\| \ = \ 0 \ . \tag{6.10}$$

Therefore, the first limit in (6.8) must also be zero. We can think of the expression as the dot-product of two vectors: the unit vector $e/\|e\|$, and the vector $2(\mathbf{A}^t\mathbf{A}\beta - \mathbf{A}^t y)$. The dot-product of a unit vector with any vector is that vector's length (norm) times the cosine of the angle between the two vectors. Since the limit must be zero for all unit vectors $e/\|e\|$, the only alternative is that

$$\mathbf{A}^t\mathbf{A}\beta - \mathbf{A}^t y = 0 \; , \tag{6.11}$$

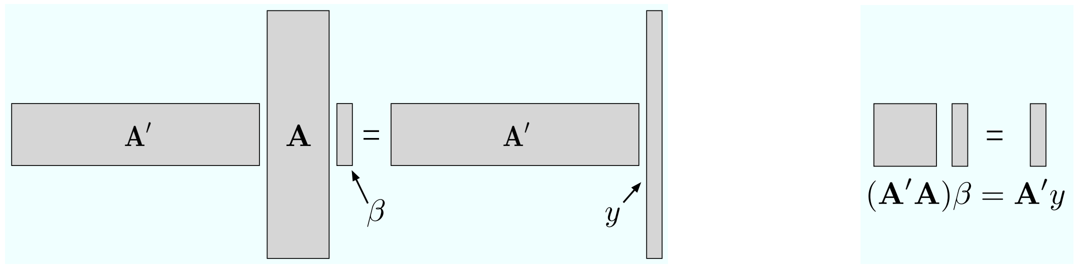
and hence we arrive at the normal equations,

$$\mathbf{A}^t\mathbf{A}\beta = \mathbf{A}^t y \; . \tag{6.12}$$

The important thing to note about the normal equations is that the matrix $\mathbf{A}^t\mathbf{A}$ is square and symmetric. Figure 15 illustrates this fact. Thus, assuming $\mathbf{A}^t\mathbf{A}$ is nonsingular, we can solve for the optimal β by simply inverting $\mathbf{A}^t\mathbf{A}$,

$$\beta = \left(\mathbf{A}^t\mathbf{A}\right)^{-1}\mathbf{A}^t y \; . \tag{6.13}$$

The matrix $\left(\mathbf{A}^t\mathbf{A}\right)^{-1}\mathbf{A}^t$ is called the *pesudoinverse* of $\mathbf{A}$, and is denoted $\mathbf{A}^\dagger$. Using the least squares fitting parameters, the restored image is compared to the original and the over-exposed images in Figure 16.



(a) (b)

Figure 15: Illustration of the normal equations. (a) shows the $\mathbf{A}$ matrices explicitly, while (b) shows the size of the system after multiplying.

There is an important geometrical interpretation to the least squares fit. Consider the following derivation of the residual.

$$\begin{aligned} y - \mathbf{A}\beta &= r & (6.14)\\ y &= \mathbf{A}\beta + r & (6.15)\\ \mathbf{A}^t y &= \mathbf{A}^t\mathbf{A}\beta + \mathbf{A}^t r & (6.16)\\ (\mathbf{A}^t\mathbf{A})^{-1}\mathbf{A}^t y &= \beta + (\mathbf{A}^t\mathbf{A})^{-1}\mathbf{A}^t r & (6.17)\\ \mathbf{A}^\dagger y &= \beta + \mathbf{A}^\dagger r & (6.18) \end{aligned}$$

Then, multiplying (6.18) on both sides by $\mathbf{A}$ gives

$$\mathbf{A}\mathbf{A}^\dagger y = \mathbf{A}\beta + \mathbf{A}\mathbf{A}^\dagger r \; . \tag{6.19}$$

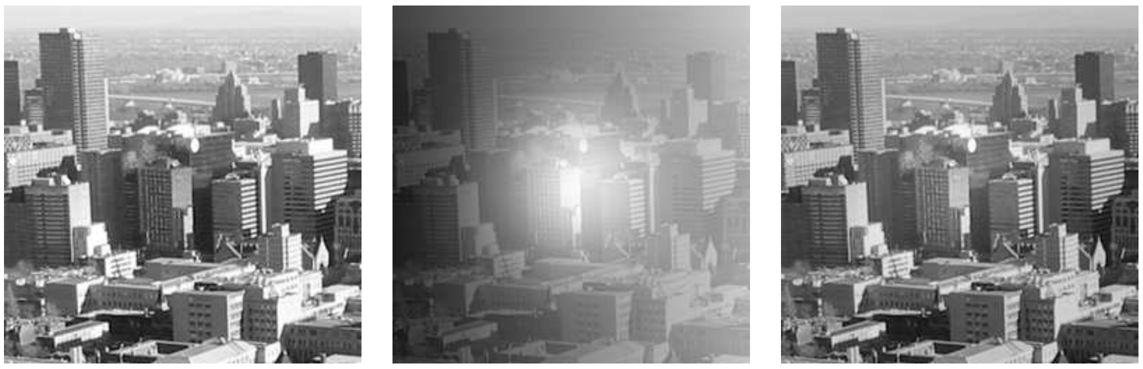

(a) Original (b) Over-exposed (c) Restored

Figure 16: Corruption and restoration of the Montreal image. The total squared error for the over-exposed image is 134,770. The total squared error for the restored image is 13,005 (corresponding to parameter values $[163.4\ 2.047\ 0.713]^t$).

The amazing thing is that $\mathbf{A}\mathbf{A}^\dagger$ is a projection matrix[3]. We will denote it $P_\mathbf{A}$. That is, if y is a vector, then the vector $P_\mathbf{A}y$ is the orthogonal component of y in the range of $\mathbf{A}$ (the subspace spanned by the columns of $\mathbf{A}$), as shown in Figure 17. The nice thing about the least squares solution is that the residual term, $P_\mathbf{A}r$, is zero (compare (6.16) to (6.12)). This gives us a geometrical interpretation for the least squares solution. Here is the way to think about it. Our model consists all the images that we can make by taking linear combinations of the exposure artifacts (stored in the columns of $\mathbf{A}$); essentially, multiplying the matrix $\mathbf{A}$ by the vector of parameters β. Hence, range($\mathbf{A}$) is the space of all possible images that we can model exactly. The observations probably do not fit our model perfectly, so the vector y is not in the subspace range($\mathbf{A}$). The least squares parametric fit corresponds to the closest point in range($\mathbf{A}$) to y, the orthogonal projection (also known as the perpendicular distance).

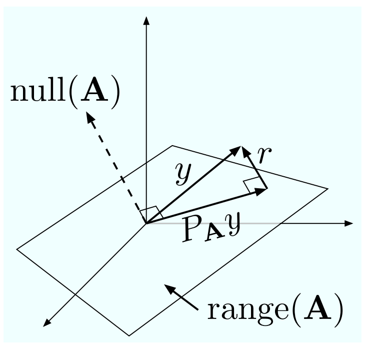


Figure 17: Orthogonal projection decomposition of the vector y.

[3]A projection matrix is any matrix P that satisfies $P^2 = P$. Geometrically, the projection matrix moves all vectors in our space onto a subspace . Which subspace it projects onto depends on P.

6.4 Solving Least Squares Problems in Matlab

Here is a Matlab m-file to compute β.

```
% cubic least squares fitting for the Montreal over-exposure problem
montreal = double( imread('Montreal_corrupt.jpg') );
[h w] = size(montreal);
y = montreal(:);   % this is the observed (corrupted) image

a1 = ones(w*h,1);   % constant component

gauss = fspecial('gaussian', size(montreal), 40);  % Gaussian blob component
a2 = gauss(:) / max(gauss(:)) * 600;

x_grad = repmat(1:w,h,1);
a3 = x_grad(:);     % left-right gradient component

A = [a1 a2 a3];

beta = (A' * A) \ (A' * y)   % solve the normal equations

r = y - A * beta;  % remove unwanted exposure artifacts
montreal_restored = reshape(r,h,w);
imagesc(montreal_restored); colormap(gray);
```

6.5 Another example: Canada's Population

Population growth is often modeled as an exponential process, modeled by the formula

$$P(t) = a\,e^{mt}\ , \tag{6.20}$$

where a and m are parameters that are different for different populations (e.g. the a and m for Canada will be different than for China).

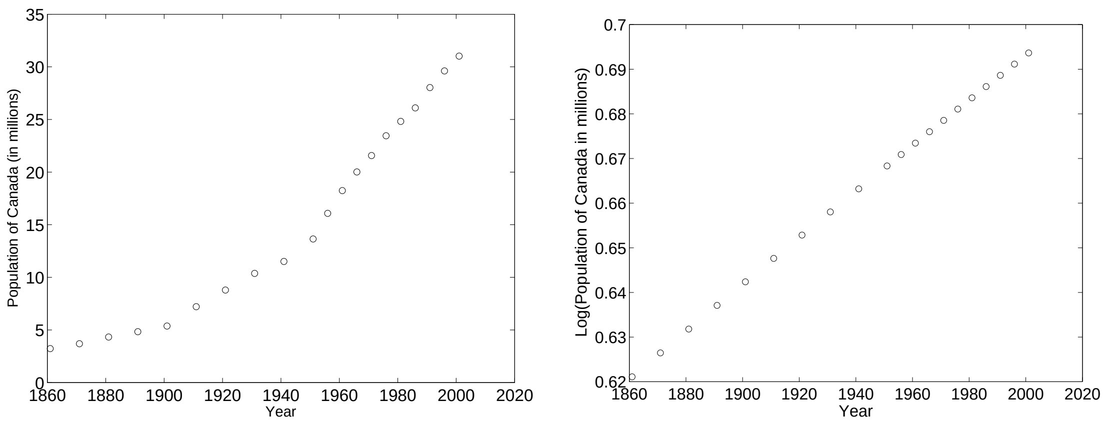
The Statistics Canada web page

`http://www.statcan.ca/english/Pgdb/demo03.htm`

shows census data for Canada's population from 1851 to 2001. The total population data is plotted in Figure 18(a). The data points seem to be on (or close to) an exponential curve, so the model in (6.20) fits.

We would like to perform a least squares parameter fitting of the model in (6.20) to the population data (i.e. find the a and m values that best fit the data). However, since the formula for the population is non-linear, we cannot represent the data using a linear model like that in (6.2). Thus, none of the least squares theory already developed can be used directly on this data. We instead manipulate (6.20), and take the logarithm of both sides.

$$\begin{aligned} P(t) &= a\,e^{mt} && (6.21)\\ \log P(t) &= \log\left(a\,e^{mt}\right) && (6.22)\\ \log P(t) &= \log a + mt \quad (\text{since } \log e^{mt} = mt) && (6.23)\\ & && (6.24) \end{aligned}$$



(a) Population of Canada (b) Logarithm of Population of Canada

Figure 18: Population of Canada over 150 years

If we define y to be $\log P(t)$, and b to be $\log a$, then we have a linear model,

$$y = mt + b \ . \tag{6.25}$$

Figure 18(b) plots the population data with the vertical (population) axis in a log-scale. As you can see, the data falls almost perfectly on a line. Our goal now is to find the m and b values to fit the line in Figure 18(b). This type of fitting is called a **log-linear** fitting.

Recall the linear model in (6.2),

$$y = \mathbf{A}\beta + \epsilon \ . \tag{6.26}$$

We will define β, our vector of model parameters, as $[m\ b]^t$. The corresponding matrix $\mathbf{A}$ then must have all the t-values (years) of the data points in the first column, and a column of ones in the second column.

$$\begin{bmatrix} y_{1851} \\ y_{1861} \\ \vdots \\ y_{2001} \end{bmatrix} = \begin{bmatrix} 1851 & 1 \\ 1861 & 1 \\ \vdots & \\ 2001 & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix} + \epsilon \tag{6.27}$$

Notice that the random noise variable, ϵ, is of no interest in this model, unlike in the image over-exposure example above. It is simply a column-vector of residual values that result when we choose particular values for the model parameters. In this case, we are interested in the model.

Solving the least squares problem is as simple as solving the linear system of equations $(\mathbf{A}^t\mathbf{A})\,\beta = \mathbf{A}^t y$ for β. Figure 19 plots the least squares log-linear model fit over the data points. The following script creates the plot.

```
load Canada_popn_data.txt   % read in data

t = Canada_popn_data(:,1);  % extract 1st column (year)
P = Canada_popn_data(:,2);  % extract 2nd column (pop'n)
N = size(P,1);

y = log(P);            % take the log of the population

A = [t ones(N,1)];     % create the system matrix
beta = (A'*A) \ (A'*y) % Solve the Least Squares problem!

m = beta(1);           % Read out parameter values
a = exp(beta(2));      %  -Recall that log(a) = b = beta(2)

Pmodel = a * exp(m*t); % generate some data to plot the model

% Plot data and model curve
plot(t, P/1000, 'o', t, Pmodel/1000, 'r-');
ylabel('Population of Canada (in millions)');
xlabel('Year');
```

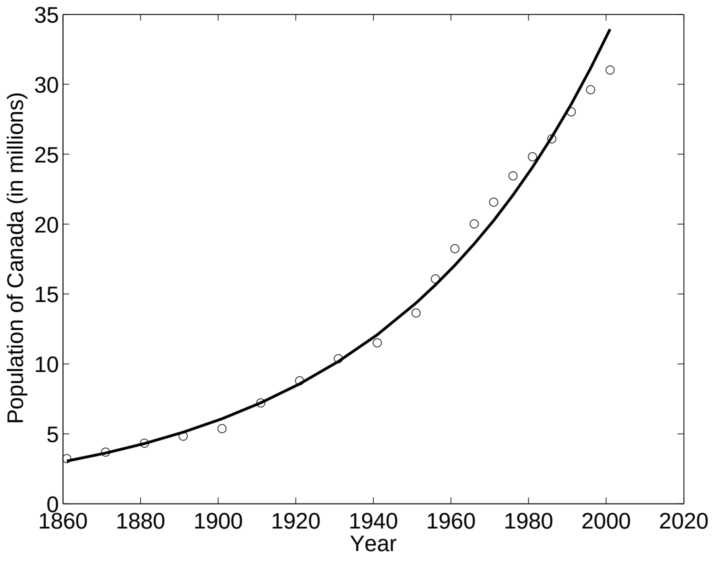


Figure 19: Least squares log-linear model of Canada's population

Exercise

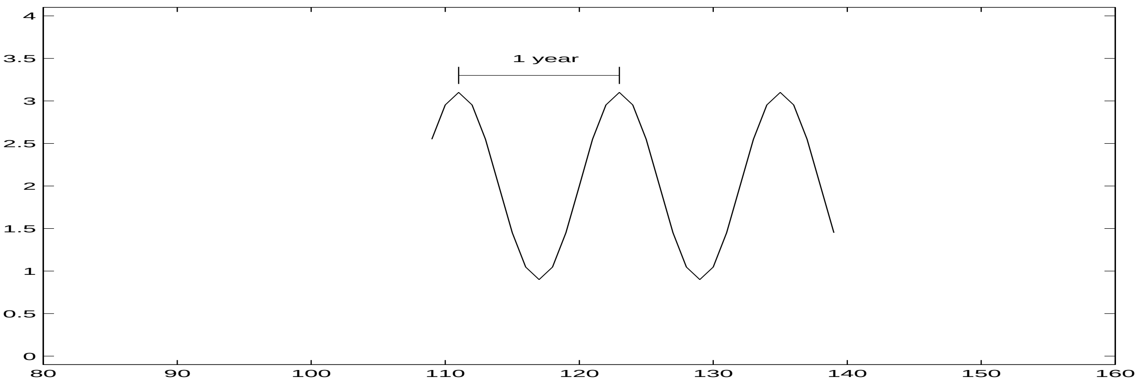
1. Describe a simple but efficient way to compute $\mathbf{A}^t\mathbf{A}$, and indicate how many multiplications it requires?

7 Fourier Transforms

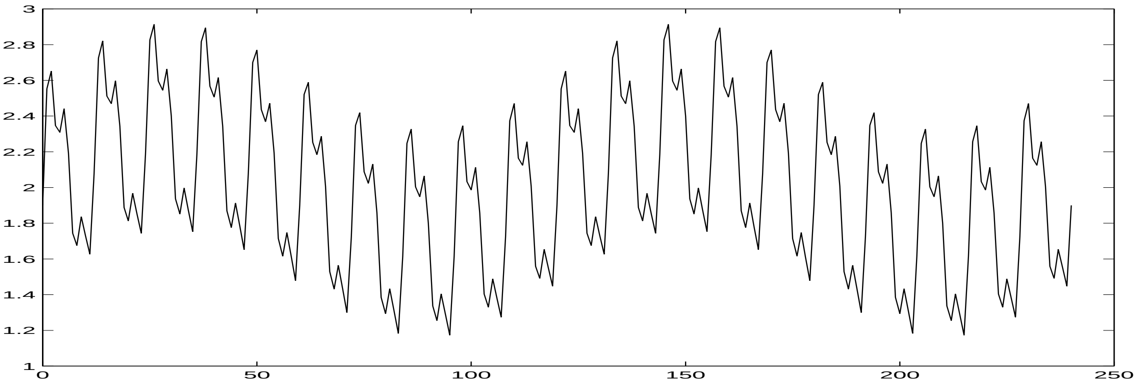
In this chapter we will introduce a tool, the Discrete Fourier Transform, which transforms a finite set of data in one or more dimensions into a second finite set of data. The transformation is reversible and gives an alternate view of information which may be more useful for certain applications. In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

7.1 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



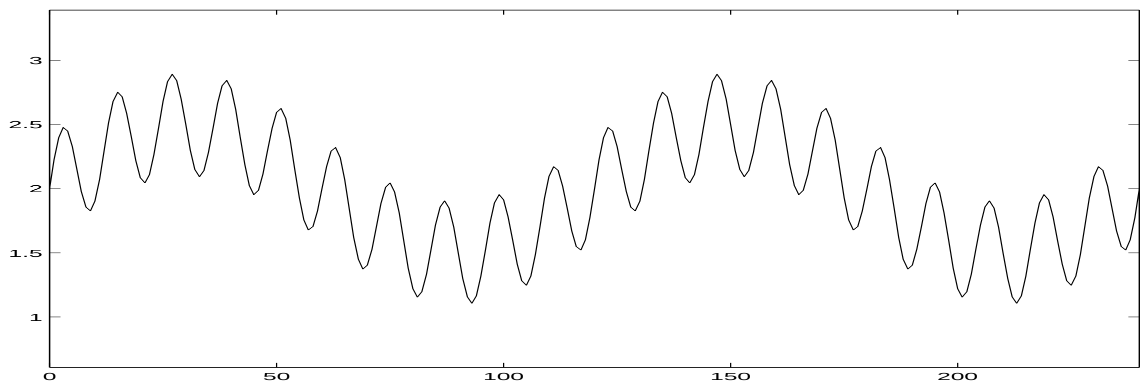
due to such factors as imports from the southern hemisphere, weather fluctuations caused by sunspots or El Nino phenomena, and so on. These factors might also occur on a regular predictable pattern but with alternate cycle periods. As such then we need to see how much the data cycles once every 240 months, 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

$$f(t) = a_0 + a_1 \cos(q \cdot t) + b_1 \sin(q \cdot t) + a_2 \cos(2q \cdot t) + b_2 \sin(2q \cdot t) + \cdots \qquad (7.1)$$

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 (7.1).

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 rather than continuous data $f_0, \ldots, f_{N-1}$.

c) How to do the discrete computation fast.

d) Applications of Fourier analysis.

7.2 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, has the property that

$$f(t \pm T) = f(t) \tag{7.2}$$

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,

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

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

$$f(t) = a_0 + \sum_{k=1}^{\infty} a_k \cos(\frac{2\pi kt}{T}) + \sum_{k=1}^{\infty} b_k \sin(\frac{2\pi kt}{T}) \tag{7.3}$$

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

$$f(t) = a_0 + \sum_{k=1}^{\infty} a_k \cos(kt) + \sum_{k=1}^{\infty} b_k \sin(kt). \tag{7.4}$$

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

$$\begin{aligned}
\int_0^{2\pi} \cos kt \sin kt \; dt &= 0 \\
\int_0^{2\pi} \cos kt \cos k't \; dt &= 0 \; ; \; k \neq k' \\
\int_0^{2\pi} \sin kt \sin k't \; dt &= 0 \; ; \; k \neq k' \\
\int_0^{2\pi} \sin kt \; dt &= 0 \\
\int_0^{2\pi} \cos kt \; dt &= 0.
\end{aligned} \tag{7.5}$$

(this is the same as saying that the functions $(1, \cos kt, \sin kt)$ are *orthogonal* on $[0, 2\pi]$).

From equations (7.4-7.5) we can determine the coefficients a_k, b_k.

$$\begin{aligned}
a_0 &= \frac{\displaystyle\int_0^{2\pi} f(t)\ dt}{2\pi} \\
a_k &= \frac{\displaystyle\int_0^{2\pi} f(t)\cos kt\ dt}{\displaystyle\int_0^{2\pi} \cos^2 kt\ dt} \\
b_k &= \frac{\displaystyle\int_0^{2\pi} f(t)\sin kt\ dt}{\displaystyle\int_0^{2\pi} \sin^2 kt\ dt} \qquad (7.6)
\end{aligned}$$

For example, a formula for a_ℓ is determined by multiplying both sides of equation (7.4) 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* (7.4) more compactly if we use Euler's formula:

$$e^{i\cdot\theta} = \cos(\theta) + i\sin(\theta) \qquad (7.7)$$

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

$$f(t) = \sum_{k=-\infty}^{\infty} c_k e^{ikt} \qquad (7.8)$$

where c_k are in general now complex numbers. The correspondence with the c_k and the a_k, b_k of representation (7.3) 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

$$\begin{aligned} \int_0^{2\pi} e^{ikt}e^{-i\ell t}\, dt &= 0 \; ; \; k \neq \ell \\ &= 2\pi \; ; \; k = \ell. \end{aligned} \tag{7.9}$$

If we multiply both sides of (7.8) by $e^{-i\ell t}$ and integrate term by term we obtain the formula

$$c_\ell = \frac{1}{2\pi}\int_0^{2\pi} e^{-i\ell t} f(t)\, dt. \tag{7.10}$$

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

$$f(t) \simeq \sum_{k=-M}^{M} c_k e^{i\ kt} \tag{7.11}$$

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 (i.e. JPEG) utilize that fact that little information is carried in high frequency components to reduce that storage required for a digital image.

7.3 Discrete Fourier Transform (DFT)

We now have covered the background mathematics for defining the discrete Fourier transform (DFT). In general and formal terms, the DFT is a reversible transformation of a sequence of N complex numbers. Specifically, it transforms the sequence $\{f_n |\ n = 0, \ldots, N-1\}$, which we will refer to as a data sample, into another sequence of N complex numbers, $\{F_k |\ k = 0, \ldots, N-1\}$, which we refer to as the Fourier coefficients of the sample. 'Reversible' means that if we are given the Fourier coefficients, there is a simple inverse transformation by which we can reproduce the data sample itself. In common applications of the DFT, the f_n are real values and are observations of some process (e.g. electrical signals, prices, temperatures, etc.).

Why bother? We will try to answer this in general terms; the assignments make an extensive study of one particular application. 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.

Recall from (B.5) that $W = e^{2\pi i/N}$ is the Nth root of unity. Then we define the Fourier coefficient, F_k, by

$$F_k = \frac{1}{N} \sum_{n=0}^{N-1} f_n \, \overline{W}^{nk} \ . \tag{7.12}$$

We will sometimes denote this operation as a function called "DFT", so that $F = \mathrm{DFT}(f)$. One might reasonably assume that this definition is meant to apply for $0 \le k \le N-1$; 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 distinct; the rest are duplicates of them. Let us see why.

Assume that $\{F_k| \; k = 0, \ldots, N-1\}$ have been computed by (7.12). 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 '$p \equiv n \bmod N$'. Each term in the summation in (7.12) contains a power of $\overline{W}^k$, and

$$\overline{W}^k = e^{-2k\pi i/N} = e^{-2mN\pi i/N} e^{-2p\pi i/N} = \overline{W}^p \ .$$

So

$$F_k = \frac{1}{N} \sum_{n=0}^{N-1} f_n \, \overline{W}^{nk} = \frac{1}{N} \sum_{n=0}^{N-1} f_n \overline{W}^{np} = F_p \ . \tag{7.13}$$

Hence we can regard the definition (7.12) of $\{F_k\}$ as defining a doubly infinite and periodic sequence ($-\infty < k < \infty$) that is determined by the subsequence[4] with $0 \le k \le N-1$.

We now observe another property of 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,

$$\begin{aligned} W^{N-k} &= e^{2\pi i(N-k)/N} \\ &= e^{2\pi i} e^{-2\pi i k/N} \\ &= \overline{W}^k \ . \end{aligned}$$

Consequently, if the data values $\{f_n\}$ are real, then

$$F_k = \frac{1}{N} \sum_{n=0}^{N-1} f_n \, \overline{W}^{kn} = \frac{1}{N} \sum_{n=0}^{N-1} f_n \, W^{(N-k)n} = \overline{F_{N-k}} \ . \tag{7.14}$$

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

[4]Mathematically, any subsequence of length N works.

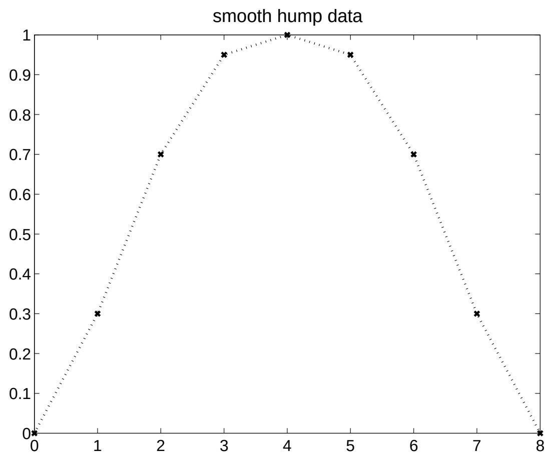
$F_k = \overline{F_{N-k}}$ for $1 \leq k \leq N-1$. For example, if $N = 9$, then we know that $F_1 = \overline{F_8}$, $F_2 = \overline{F_7}$, etc.

Here are two examples for $N = 9$.

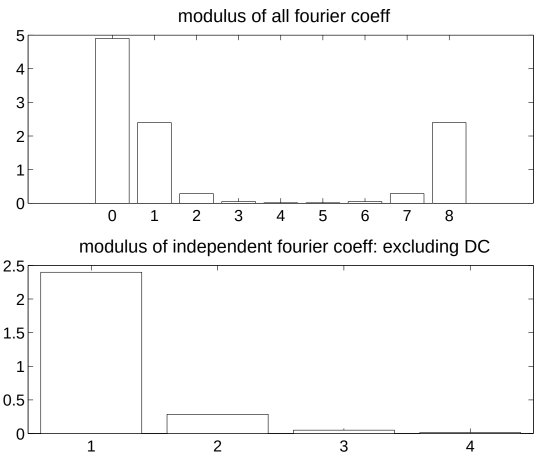
Example 1: a smooth hump

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 (7.14), 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.



data sample for smooth hump example



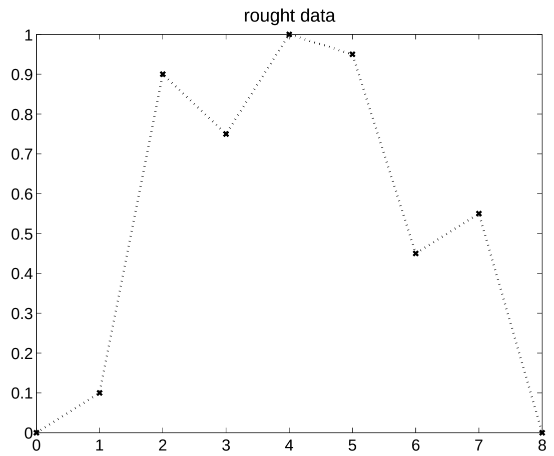
Fourier coefficients for smooth hump data sample

The graphs on the the right, show the modulus (magnitude) of the Fourier coefficients, $|F_k|$. The upper graph shows all 9 coefficients. This is redundant because of the conjugate symmetry. The lower graph shows only the coefficients for $k = 1, 2, 3, 4$. We do not show $|F_0|$ because it contains a particularly simple piece of information about the data sample, i.e. F_0 is the sum of the data sample values.

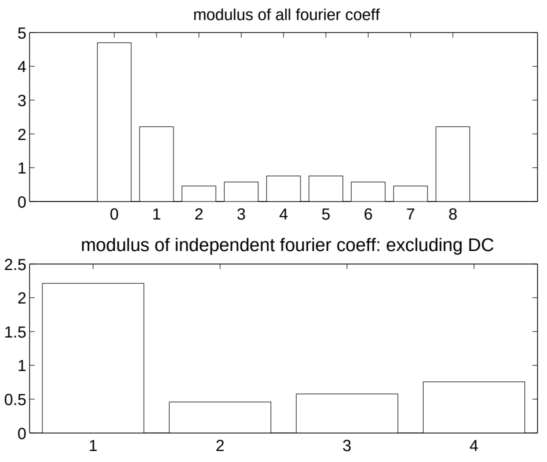
Example 2: a rough hump

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 Example 2 (shown in the figure), the data sample also has a smaller pattern that occurs about four times (i.e. about 4 smaller fluctuations); this is reflected in the fact that F_4 is the largest coefficient other than F_0 and F_1.

The extent to which the data varies *smoothly* with n is related to how quickly 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 $k = 2, 3$ and 4. In the case of the rough hump, these ratios are bigger than 1, and the curve is less smooth.

7.3.1 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_k are defined by (7.12) , then this transformation is

$$f_n = \sum_{k=0}^{N-1} F_k W^{nk} \ . \tag{7.15}$$

This can be established by looking at a matrix representation of (7.12), and using the orthogonality of (B.9). Let f be the column vector of the data sample, and let F be the column vector of Fourier coefficients. Then (7.12) can be written

$$F = \frac{1}{N} M f \tag{7.16}$$

where M is the $N \times N$ matrix in which the jth column is $\overline{W(j)}$. In matrix form, equation (B.9) states that

$$\overline{M}^t\, M = NI \tag{7.17}$$

where I is the $N \times N$ identity matrix. In other words,

$$M^{-1} = \frac{1}{N}\overline{M}^t \ .$$

So, from (7.16), we can conclude that

$$f = \overline{M}^t F \ .$$

If we write this out componentwise, we find that it is (7.15).

7.3.2 Lack of Standardization

Unfortunately, there is no standard definition of the exact formula used to define the Fourier transform and its inverse. The differences do not affect the basic role of the transform (as an alternative representation of a sequence of 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 transformation. We will use variable c for it. The value of c will determine the scale factor needed in the inverse transform, which must be Nc.

Using these parametric symbols, the common definitions of the transform and inverse transform can be written

$$\begin{aligned} F_k &= c\sum_{n=0}^{N-1} f_n(\overline{U}^k)^n && (7.18)\\ f_n &= \frac{1}{Nc}\sum_{k=0}^{N-1} F_k(U^n)^k. \end{aligned}$$

In these notes, we use $U = e^{2\pi i/N}$ and $c = 1/N$. Matlab uses $U = e^{-2\pi i/N}$ and $c = 1$. Moreover, Matlab's array indexing starts with 1 instead of 0. This can cause some minor confusion between the mathematics and the Matlab computation. For example, to store a data sample $\{f_n |\ n = 0 \ldots N-1\}$ in a Matlab array x, it stores f_{n-1} in $x(n)$.

7.4 Dependencies Among the Fourier Coefficients

Let $\{f_n\}$ be a sequence of N real numbers and $\{F_k\}$ be its Fourier transform. Then $\{F_k\}$ is a sequence of N *complex* numbers, and each complex number is a *pair* of real numbers. Hence, the set of complex numbers contains twice as many real numbers, $2N$. Since $\{F_k\}$ is completely determined by $\{f_n\}$, there must be N dependencies among the real and imaginary parts of the $\{F_k\}$. That is, there must be special characteristics in the complex numbers of the Fourier transform of a set of real numbers.

The good news is that a description of these dependencies is independent of the particular details of the definition of $\{F_k\}$ discussed above. The bad news is that it is a bit complicated by slight differences when N is even or odd. Let m be determined by $N = 2m$ or $N = 2m + 1$ ($m =$ `floor`$(N/2)$; see Matlab `floor` function.) The complications start with the range of indices for $\{F_k\}$.

$$N\text{odd} \;\rightarrow\; -m \leq k \leq m \qquad\qquad N\text{even} \;\rightarrow\; -(m-1) \leq k \leq m \tag{7.19}$$

There is one dependency that always holds when $\{f_n\}$ is real, regardless of the parity of N:

$$imag(F_0) = 0 \; . \tag{7.20}$$

i.e. F_0 is real. In fact, F_0 is the sum of the $\{f_n\}$; it is sometimes referred to as the DC component, DC value, or DC coefficient. Its role in representing signals is distinguished from the other coefficients.

If N is odd, then

$$F_{-k} = \overline{F_k} \quad \text{for} \quad k = 1, \ldots, m \; . \tag{7.21}$$

For each k, this represents two real-number dependencies since both $real(F_{-k})$ and $imag(F_{-k})$ are determined by (7.21). Notice that (7.20) and (7.21) provide the expected N dependencies for when N is odd.

If N is even, then $F_{-k} = \overline{F_k}$ for $k = 1, \ldots, m-1$ and $imag(F_m) = 0$. These dependencies plus (7.20) make up the N dependencies for this case.

7.5 1-D Image Compression

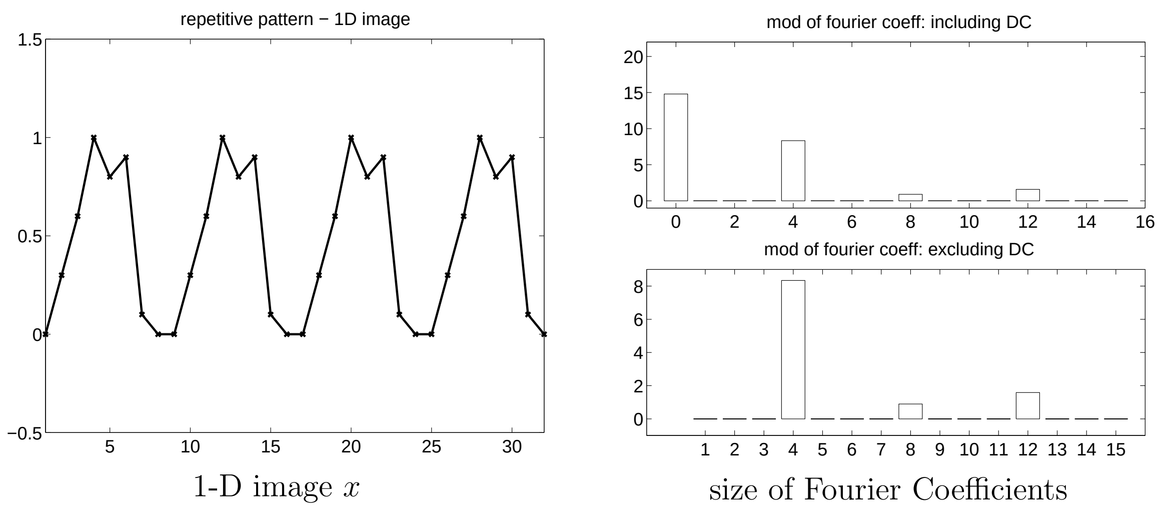


Figure 20: 1-D analogy of scaled image data and its Fourier transform

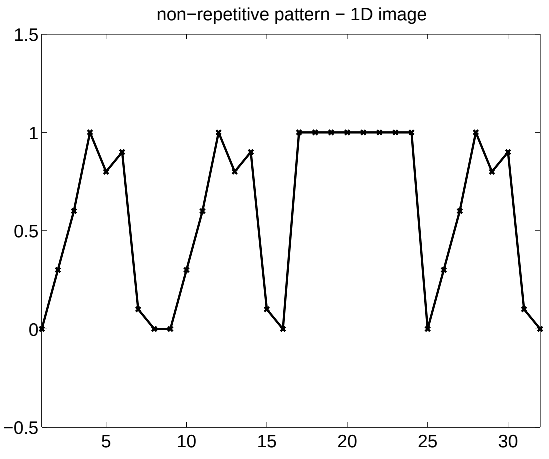
On the left of Figure 20, we show a plot of the intensities of a 32-number 1D image. Actually, it is a small pattern that is repeated 4 times. The pattern can be specified with 7 numbers so there are really only 7 'pieces' of real-number information in the image. On the upper right of Figure 20, we show 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 is F_0 which is real. So F has the information of 7 real numbers. Because the pattern has Fourier frequency 4, $F_1 = F_2 = F_3 = 0$. On the lower right of Figure 20, 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 ($|F_0|$).

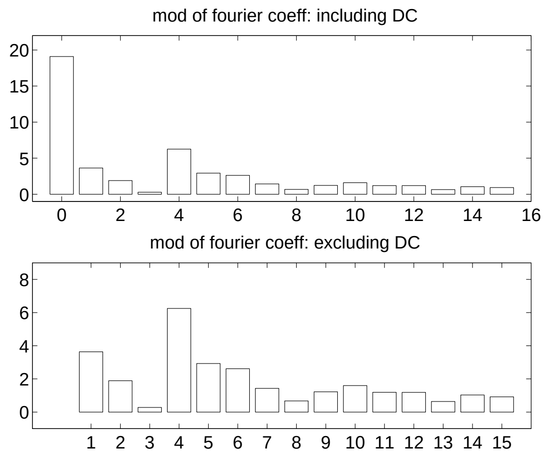
If we create a new pattern, y, which is somewhat like x by setting

$$\begin{aligned} y(1:16) &= x(1:16) \\ y(17:24) &= 1.0 \\ y(25:32) &= x(25:32)\ , \end{aligned} \tag{7.22}$$

then y has some of the pattern of x, but is in fact non-repetitive. The figure below shows y, as well as the size of its Fourier coefficients. Notice that F_1, F_2 and F_3 are not zero for this case. However, Fourier coefficient 4 is still the second largest (after F_0), indicating a strong resemblance to a pattern that repeats four times in the total sequence.



1-D image x



size of Fourier Coefficients

Image y has a number of small Fourier coefficients. How important is the information stored in them? If we drop the four coefficients that have a modulus smaller than 1, and reconstruct a new image vector from the inverse transform of this new set of Fourier coefficients, we get the compressed image shown in Figure 21.

7.6 2-D Image Compression

The data for a Matlab image is an array, X, of pixel data. The image specified by X is displayed as a rectangle of very small squares called pixels. The pixel at position (i, j) is coloured according to the value $X(i, j)$. The pixel data may be of several types. The type that we will discuss is double precision floating point numbers in the range $0 \le x \le 1$. We will call this type of image data *scaled image* data. The image can be displayed in a Matlab figure window by the command `imagesc`(X), in conjunction with a colormap for that figure, set either by default or explicitly by the `colormap` command. When a gray scale is used to display the array (i.e. using the command `colormap`(gray)),

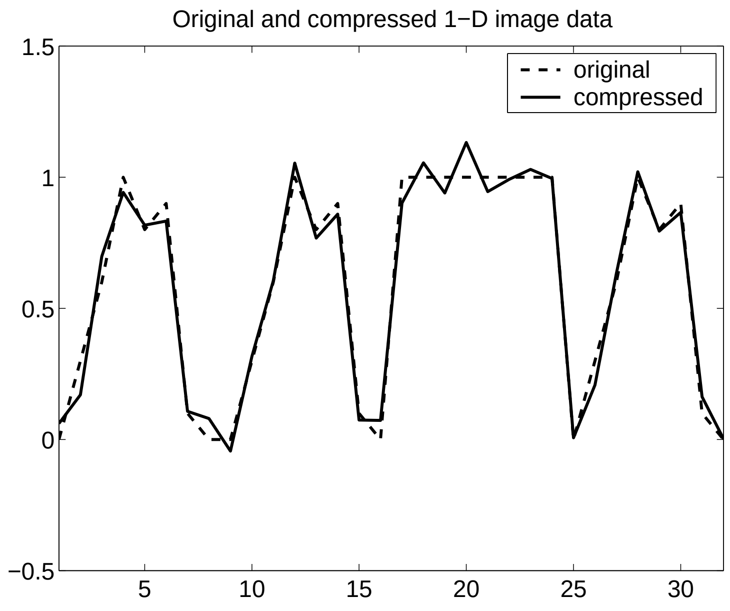


Figure 21: Comparison of original (non-repetitive) image and compressed image

the square at position (i, j) is set to a shade of gray determined by $X(i, j)$, with 0 being black and 1 being white.

Figure 22(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 §7.8).

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 the 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 many images 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 2D Fourier transform of a 16×16 array X of scaled image data is described in §7.8 as a complex array $\{F_{k,l}\}$ indexed by $0 \leq k, l \leq 15$. Matlab uses a 16×16 complex array, `Z`, to represent $\{F_{k,l}\}$, with $F_{0,0}$ stored in `Z(1,1)`, and $F_{k,l}$ stored in `Z(k+1,l+1)` for $0 \leq k, l \leq 15$. The DC component of the 2D Fourier transform is $F_{0,0}$.

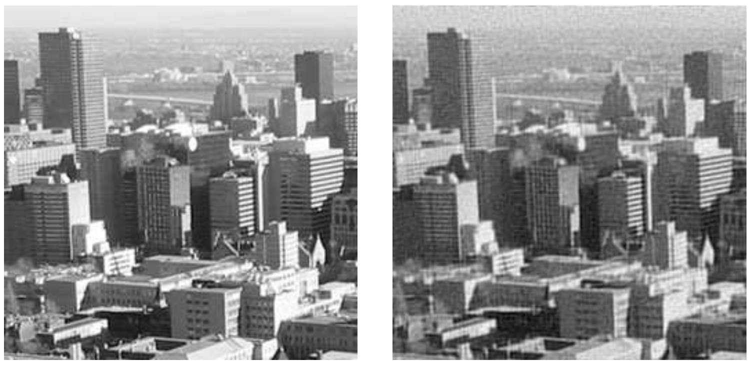

(a) Original (b) Compressed by 85%

Figure 22: Images of Montreal

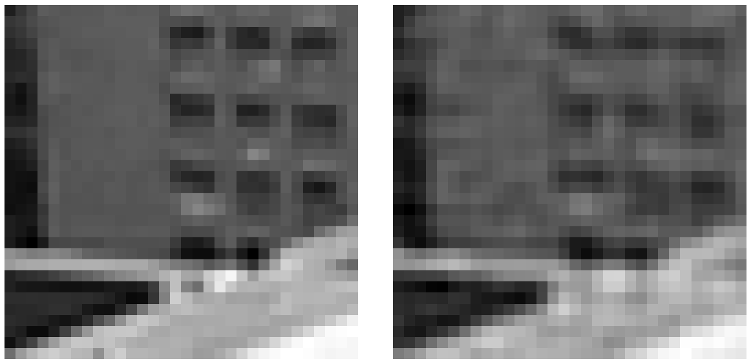

(a) Original (b) Compressed by 85%

Figure 23: A 32×32 sub-block of the images in Figure 22

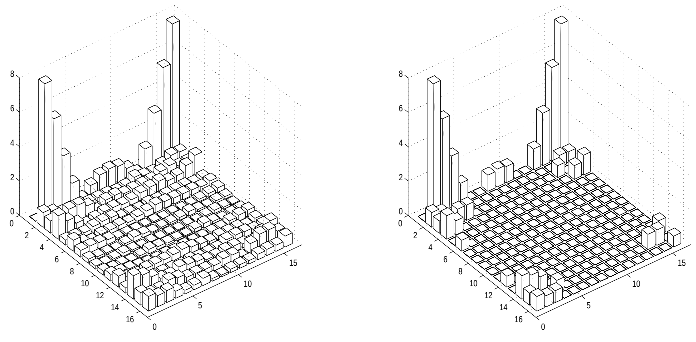
Figure 24 shows the magnitudes of the Fourier coefficients for a 16×16 sub-block of Figure 22 with the DC component set to zero (so the pattern dependent coefficients can be seen more easily). The compressed transform is shown on the right.

7.7 Fast Fourier Transform

The Fast Fourier Transform (FFT) is an algorithm (or group of algorithms) that is designed to evaluate the DFT very efficiently. Suppose we would like to compute all N Fourier coefficients,

$$F_k = \frac{1}{N} \sum_{n=0}^{N-1} f_n \, \overline{W}^{nk} \quad \text{for } k = 0, \ldots, N-1 \tag{7.23}$$

where $W = e^{2\pi i/N}$ is an Nth root of unity. Note that evaluating the summation directly will cost $O(N)$ complex floating point operations for each value of k. Thus, evaluating F_k for all N values of k costs a total of $O(N^2)$ complex floating point operations.



(a) Original (b) Compressed by 85%

Figure 24: Magnitudes of the Fourier coefficients for a 16×16 sub-block of the image in Figure 22. Subfigure (a) shows the original Fourier coefficients, and (b) shows the coefficients after compression.

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. 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,

$$F_k = \frac{1}{N} \sum_{n=0}^{\frac{N}{2}-1} f_n \overline{W}^{nk} + \frac{1}{N} \sum_{n=\frac{N}{2}}^{N-1} f_n \overline{W}^{nk} . \tag{7.24}$$

Letting $p = n - \frac{N}{2}$, these summations can be written

$$\begin{aligned} F_k &= \frac{1}{N} \sum_{n=0}^{\frac{N}{2}-1} f_n \overline{W}^{nk} + \frac{1}{N} \sum_{p=0}^{\frac{N}{2}-1} f_{p+\frac{N}{2}} \overline{W}^{(p+\frac{N}{2})k} \\ &= \frac{1}{N} \sum_{n=0}^{\frac{N}{2}-1} f_n \overline{W}^{nk} + \frac{1}{N} \sum_{n=0}^{\frac{N}{2}-1} f_{n+\frac{N}{2}} \overline{W}^{nk} \overline{W}^{\frac{N}{2}k} \\ &= \frac{1}{N} \sum_{n=0}^{\frac{N}{2}-1} \left(f_n + \overline{W}^{\frac{N}{2}k} f_{n+\frac{N}{2}} \right) \overline{W}^{nk} . \end{aligned} \tag{7.25}$$

We know that $\overline{W}^{\frac{N}{2}k} = e^{-ik\pi} = (-1)^k$. Thus, for even values of k, equation (7.25) becomes

$$F_{2k} = \frac{1}{N} \sum_{n=0}^{\frac{N}{2}-1} \left(f_n + f_{n+\frac{N}{2}} \right) \overline{W}^{2nk} \qquad k = 0, \ldots, \frac{N}{2} - 1 , \tag{7.26}$$

while for odd values of k, equation (7.25) becomes

$$\begin{aligned} F_{2k+1} &= \frac{1}{N}\sum_{n=0}^{\frac{N}{2}-1}\left(f_n - f_{n+\frac{N}{2}}\right)\overline{W}^{n(2k+1)} \\ &= \frac{1}{N}\sum_{n=0}^{\frac{N}{2}-1}\left[\left(f_n - f_{n+\frac{N}{2}}\right)\overline{W}^{n}\right]\overline{W}^{2nk} \qquad k=0,\ldots,\frac{N}{2}-1\ . \end{aligned} \tag{7.27}$$

If we set

$$\begin{aligned} g_n &= \frac{1}{2}\left(f_n + f_{n+\frac{N}{2}}\right) \\ h_n &= \frac{1}{2}\left(f_n - f_{n+\frac{N}{2}}\right)\overline{W}^{n}\ , \end{aligned} \tag{7.28}$$

then equations (7.26-7.27) become

$$F_{2k} = \frac{1}{\frac{N}{2}}\sum_{n=0}^{\frac{N}{2}-1} g_n\overline{W}^{2nk} \tag{7.29}$$

$$F_{2k+1} = \frac{1}{\frac{N}{2}}\sum_{n=0}^{\frac{N}{2}-1} h_n\overline{W}^{2nk}\ , \tag{7.30}$$

for $k = 0,\ldots,\frac{N}{2}-1$. Both of those summations look like new Fourier transforms themselves, but of g_n and h_n instead of f_n. However, the summations have half the number of terms, and use $\overline{W}^2$ instead of $\overline{W}$. This makes sense because a Fourier transform with only $\frac{N}{2}$ elements uses the $\frac{N}{2}$th root of unity, and

$$\overline{W}^2 = e^{-\frac{2\pi i}{N}2} = e^{-\frac{2\pi i}{\frac{N}{2}}}\ .$$

Thus, equations (7.29-7.30) can be regarded as two new Fourier transforms of half the length:

$$\begin{aligned} F_{\text{even}} &= \text{DFT}(g);\ \frac{N}{2}\ \text{points} \\ F_{\text{odd}} &= \text{DFT}(h);\ \frac{N}{2}\ \text{points} \end{aligned}$$

Therefore, we have converted the problem of computing a single DFT of N points into the computation of two DFTs of $\frac{N}{2}$ points. We can then reduce each of the $\frac{N}{2}$ length DFTs into two $\frac{N}{4}$ length DFTs, 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 $O(N\log_2 N)$.

7.8 A Two Dimensional FFT

In image processing applications, a grey scale image is represented by a 2D array of grey scale values $f_{n,m}$, $n = 0, \ldots, (N-1)$ and $m = 0, \ldots, (M-1)$. In this case, the two dimensional DFT uses two roots of unity, $W_N = e^{\frac{2\pi i}{N}}$ and $W_M = e^{\frac{2\pi i}{M}}$, and is defined as

$$F_{k,l} = \frac{1}{MN} \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} f_{n,m} \overline{W}_N^{nk} \overline{W}_M^{ml} \; . \tag{7.31}$$

Given a function which computes a 1D DFT, how can we use this to compute the 2D DFT above?

We can rewrite equation (7.31) as

$$\begin{aligned} F_{k,l} &= \frac{1}{N} \sum_{n=0}^{N-1} \overline{W}_N^{nk} \left[\frac{1}{M} \sum_{m=0}^{M-1} f_{n,m} \overline{W}_M^{ml} \right] \\ &= \frac{1}{N} \sum_{n=0}^{N-1} \overline{W}_N^{nk} H_{n,l} \; , \end{aligned} \tag{7.32}$$

where $H_{n,l} = \frac{1}{M} \sum_{m=0}^{M-1} f_{n,m} \overline{W}_M^{ml}$. Thus $H_{n,l}$ can be computed efficiently by taking 1D FFTs of length M of the rows of the input data array. The final computation of $F_{k,l}$ is carried out by computing 1D FFTs of length N on the columns of $H_{n,l}$. If the complexity of a 1D DFT of length N is $CN \log_2 N$, where C is a constant, then the complexity of a 2D FFT is $CNM \log_2(MN)$.

Exercises

1. Let $\{f_n\}$, $n = 0, 1, \ldots, N-1$, be N samples of a real signal, and N be even.
   (a) Show that $W^{-nk} + W^{-(N-n)k} = 2\cos\left(\frac{2\pi i n k}{N}\right)$.
   (b) Suppose the signal $\{f_n\}$ is an even function ($f_n = f_{N-n}$ for $n = 1, 2, \ldots, \frac{N}{2} - 1$). 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 $F_{2k} = 0$, $k = 1, 2, \ldots, \frac{N}{2} - 1$.

8 Dynamic Simulation: Modeling with ODEs

In this section, we will present some small examples of modeling with differential equations. These examples are based on familiar ideas, but typical simulations can require a very sophisticated understanding of the application area (be it chemistry, finance, electronics, etc.) and usually involves dozens to thousands of equations.

8.1 Single Population Model: How is Canada Growing?

It is common to measure populations in thousands of individuals (Kpersons). Let us use the symbol P_t for a population at time t in years. For example, if the population of Canada in the year 2000 was about 30,750,000, then we write $P_{2000} \approx 30,750$.
See `http://www.statcan.ca/start.html` for the data on the Canadian population used in this section.

A human population in a fixed geographic region changes in size by

- natural regeneration by the net effects of births and deaths
- net immigration (immigration minus emigration)

Let G denote the natural regeneration *rate*; the conventional units for G is "events per Kpersons per year", usually written "events/(Kpersons-year)" (that's a dash, not a minus sign). Average event rates for Canada in 1999-2000 were:

birth rate = 10.9 (i.e. 109 babies per 10,000 persons per year)

death rate = 7.5.

The net regeneration effect is the birth rate minus the death rate, so $G \approx 3.4$. The number of immigrants arriving in Canada in 1999-2000 was about 204K. Stats Canada does not provide a recent figure for emigration; but it does provide data averaging about 50Kpersons/year for 1991-1996. So letting I be the net immigration rate, we can estimate $I \approx 155$ Kpersons/year.

To simplify the move from this data to mathematical models of population evolution, we make the BIG assumptions that G and I are constant over time. For modeling purposes, we need the net regeneration rate in units of Kevents per Kpersons per year; not in the customary census data units of events/(Kpersons-year) quoted above. So let r = G/1000 be this rate in the units of Kevents/(Kpersons-year); i.e. for Canada $r \approx .0034$. Then our conceptual model of population change from time t to time $t + \tau$ can be written

$$P_{t+\tau} \approx P_t + rP_t\tau + I\tau \ . \tag{8.1}$$

These ideas lead us to two closely related mathematical models for predicting population evolution from a known starting population P_{t_0} at a given time t_0

- a difference equation based model
- a differential equation based model.

8.1.1 Difference Equation Model

Let us pick a fixed interval of years, h. We model the population evolution as a sequence $\{p^{(n)}, n = 0, 1, 2, \ldots\}$, where $p^{(n)}$ is the model's estimate of P_{t_0+nh}. Given the initial population of $p^{(0)} = P_{t_0}$, we can compute $p^{(n+1)}$ from $p^{(n)}$ using

$$p^{(n+1)} = (1 + rh)p^{(n)} + Ih \tag{8.2}$$

Solution of (8.2): If we let $\alpha = 1 + rh$, then (8.2) can be written

$$\begin{aligned} p^{(n+1)} &= \alpha p^{(n)} + Ih \\ &= \alpha(\alpha p^{(n-1)} + Ih) + Ih \\ &= \alpha^2(\alpha p^{(n-2)} + Ih) + (\alpha + 1)Ih \\ &= \cdots \end{aligned}$$

If we continue this process, each time substituting $p^{(k)}$ with $\left(\alpha p^{(k-1)} + Ih\right)$, we eventually get down to $p^{(0)}$, and we can write

$$\begin{aligned} p^{(k)} &= \alpha^k p^{(0)} + \frac{\alpha^k - 1}{\alpha - 1} Ih \\ &= \alpha^k \left(p^{(0)} + \frac{Ih}{\alpha - 1}\right) - \frac{Ih}{\alpha - 1} \end{aligned}$$

Notice that $\frac{Ih}{\alpha-1} = \frac{I}{r}$, so

$$p^{(k)} = (1 + rh)^k \left(p^{(0)} + \frac{I}{r}\right) - \frac{I}{r} \tag{8.3}$$

Note the roles of some of the parameters:

- if $r > 0$ then $p^{(k)} \to \infty$ as $k \to \infty$ and the model predicts an unlimited population growth.
- if $r < 0$ and $rh \geq -1$ then $p^{(k)} \to \frac{-I}{r}$ as $k \to \infty$ and the model predicts a stable, finite population.
- if $r < 0$ and $rh < -1$ then the contribution of $(1 + rh)p^{(n)}$ to $p^{(n+1)}$ is negative, a model prediction that doesn't make sense. This is because h is too large for this negative r value.

8.1.2 Differential Equation Model

If we return to (8.1), we can rewrite it as $\frac{1}{\tau}(P_{t+\tau} - P_t) \cong rP_t + I$. If we introduce a continuous function of t, $p(t)$, as a mathematical model for the population at time t then, letting $\tau \to 0$, we see that $p(t)$ must satisfy the differential equation

$$\frac{dp(t)}{dt} = rp(t) + I \ . \tag{8.4}$$

Equation (8.4) is called an "ordinary differential equation", or ODE. The word "ordinary" means that the differential equation does not contain any partial derivatives. The solution of (8.4), assuming that $p(t_0) = p_0$, is

$$p(t) = \left(p_0 + \frac{I}{r}\right) e^{r(t-t_0)} - \frac{I}{r} \ . \tag{8.5}$$

This type of ODE is called an "initial value problem", or IVP, because the initial state is known.

How do models (8.2) and (8.5) compare? Note that (8.2) is a family of models, parameterized by h, and that (8.5) is derived as the limit of this family as $h \to 0$. Table 1 shows what they predict for the population in the year 2020. The entry for $h = 0$ is computed using (8.5) and the other columns are computed using (8.1). The number of time steps, N, is determined by $\frac{20}{h}$. Note that the difference equation model is more accurate when the time step size h is smaller.

h	0	1	2	5
N	-	20	10	4
P_{2020}	37.708	37.703	37.698	37.683

Table 1: Canadian population estimates for the year 2020 (in millions).

8.2 Novelty Golf Driving Range

A golf driving range has installed a one-meter wide moving barrier 30 meters out in front of the driving tee mat from where the ball is hit. The barrier height fluctuates with time according to

$$b(t) = \text{bHeight} + 1.2\cos(2\pi\omega t),$$

where ω is the frequency and t is the time in seconds.

We want to simulate the path of the golf ball. Here we assume that golf balls are hit towards the barrier, and hence we use a two-dimensional (x, y)-coordinate system. The ball starts at the driving tee at $(x, y) = (0, 0)$. At time $t = 0$, the ball is struck by the golf club and is given an initial velocity (V_x, V_y) (both V_x and V_y are greater than 0). The ball follows a simple trajectory $(x(t), y(t))$ determined by the differential equations

$$\begin{cases} \frac{dx(t)}{dt} & = & V_x \\ \frac{d^2y(t)}{dt^2} & = & -g \ , \end{cases} \tag{8.6}$$

where $g = 9.81\frac{\text{m}}{\text{s}^2}$ is the gravitational constant. If the trajectory hits the barrier, the trajectory stops at $x(t) = 30$ meters. Otherwise, it stops when the ball hits the ground at $y(t) = 0$. Figure 25 illustrates a ball trajectory clearing the barrier.

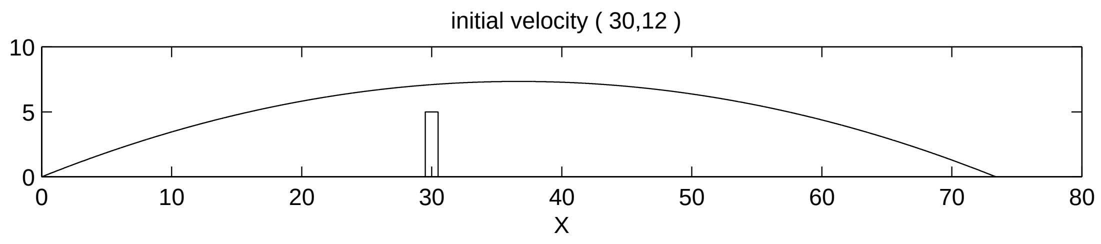


Figure 25: Trajectory of a golf ball. Notice the barrier at 30 m.

8.3 Pursuit Problems

Pursuit problems arise in numerous contexts, the most obvious involving missile guidance systems. Simply put, the problem involves a *target* and a *pursuer*, the latter moving in such a manner that its direction of motion is always towards the target. Figure 26 depicts the situation.

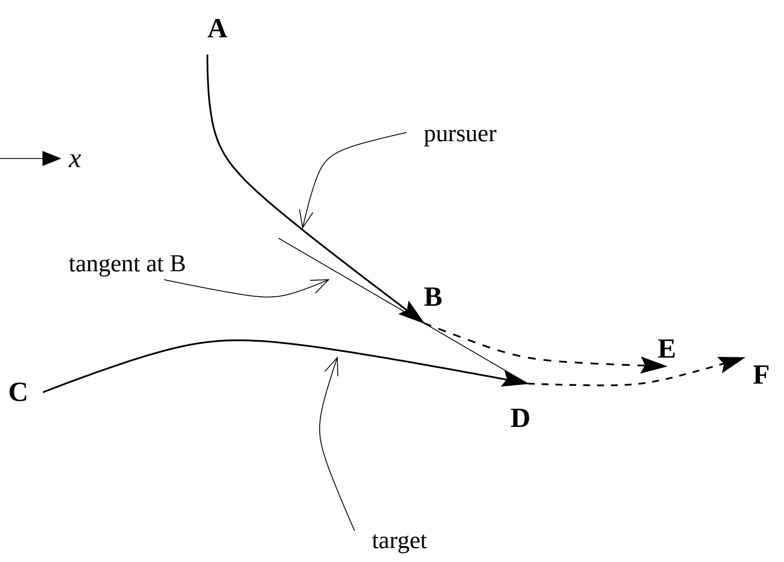


Figure 26: Pursuit problem

The trajectory of the target T is CD, and the “path of pursuit” is AB. The problem is to determine the trajectory of the pursuer, given the initial position of the pursuer and the position vector of the target as a function of time for $t \geq 0$.

8.3.1 Derivation of the Equations

In the pursuit problem, the trajectory of the target is a known curve in space. It is represented by a parametric curve in three dimensions with parameter t being the time.

We denote it by

$$T(t) = \big(x_T(t), y_T(t), z_T(t)\big) \ ,$$

where $x_T(t)$, $y_T(t)$, and $z_T(t)$ are given functions which describe the path of the target.

For any parametric curve, $C(t) = (x(t), y(t), z(t))$, the tangent line at a given $\bar{t}$ has the direction vector

$$v(\bar{t}) \equiv \frac{dC}{dt}(\bar{t}) = \left(\frac{dx}{dt}(\bar{t}), \frac{dy}{dt}(\bar{t}), \frac{dz}{dt}(\bar{t})\right) \ .$$

In the following, we shall use the simplified notation $\dot{x}(t)$ to denote $\frac{dx}{dt}(t)$, etc.

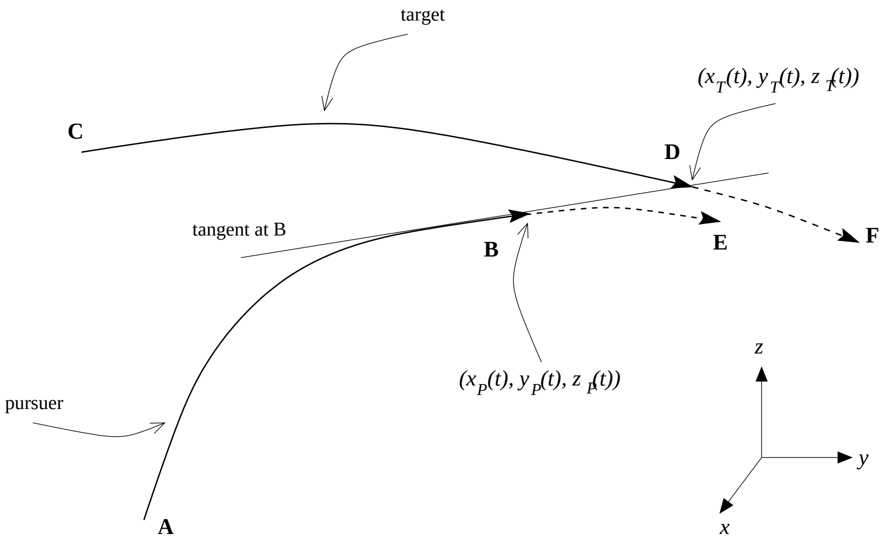


Figure 27: Three-dimensional pursuit problem.

The speed of the pursuer is assumed to be known; we will designate it as s_P and regard it as a constant, although it could be a known function of time. Our task now is to turn what we know about the pursuer's trajectory into a system of differential equations that will permit us to compute the trajectory. Let $P(t) = (x_P(t), y_P(t), z_P(t))$ be the (unknown) parametric curve of the trajectory of the pursuer. At any time t, the pursuer must point directly to the target. In other words, the tangent line at $P(t)$ should be parallel to $T(t) - P(t)$ at every time t.

$$\begin{bmatrix} \dot{x}_P(t) \\ \dot{y}_P(t) \\ \dot{z}_P(t) \end{bmatrix} = \lambda(t) \begin{bmatrix} x_T(t) - x_P(t) \\ y_T(t) - y_P(t) \\ z_T(t) - z_P(t) \end{bmatrix} \ , \tag{8.7}$$

where $\lambda(t)$ is to be determined.

In the pursuit problem, the parameter t represents time. Therefore, the tangent direction at $P(t)$ represents the velocity of the pursuer. Here, we assume that the pursuer chases the target with its full capacity at all times and hence we assume the speed (which is the length of the velocity vector) is constant, denoted by s_P. By (8.7), we conclude that

$$s_P = \lambda(t) d(t) \ ,$$

where

$$d(t) = \sqrt{\big(x_T(t) - x_P(t)\big)^2 + \big(y_T(t) - y_P(t)\big)^2 + \big(z_T(t) - z_P(t)\big)^2}\ . \tag{8.8}$$

Thus, $\lambda(t) = \frac{s_P}{d(t)}$. Finally, the equations describing the pursuer's motion are

$$\begin{bmatrix} \dot{x}_P(t) \\ \dot{y}_P(t) \\ \dot{z}_P(t) \end{bmatrix} = \frac{s_P}{d(t)} \begin{bmatrix} x_T(t) - x_P(t) \\ y_T(t) - y_P(t) \\ z_T(t) - z_P(t) \end{bmatrix} .$$

If s_P is sufficiently large, then the pursuer will intercept the target in finite time. However, in general, the trajectory of the target and the starting positions will determine whether interception occurs.

8.4 Standard First Order Form for Initial Value Problems

We are going to use computers to approximate the solution of IVPs. In order to do this, we need a standard way to communicate to the computer what the IVP is – what the system of ODEs and the initial condition is.

A general system of m ODEs consists of:

- an m-vector valued function, $f(t, z)$ of $1 + m$ variables, sometimes referred to as the system dynamics function. The input variable t is the independent variable, while the input vector z represents the current *state* of the system. We will use the word "state" (or "state variables") to refer to the (vector of) values that we are modeling. For example, the state tells us the population at a given time (for the population model), or the position of the pursuer at a given time (for the pursuer model). Writing out $f(t, z)$ with all its subscripts looks like this:

$$\begin{bmatrix} f_1(t, z_1, z_2, \dots, z_m) \\ f_2(t, z_1, z_2, \dots, z_m) \\ \vdots \\ f_m(t, z_1, z_2, \dots, z_m) \end{bmatrix} .$$

- an $m \times m$ matrix, M. In general, M may be a function of t, or even (t, z) like f. In the simple examples presented in these notes, M is the identity matrix so we don't need to worry about it.

A solution of the general first order system determined by $f(t, z)$ then is an m-vector valued function of time, $y(t)$, that satisfies

$$M \frac{dy(t)}{dt} = f(t, y(t)) \tag{8.9}$$

over some interval of time, $t_0 \leq t \leq t_{\text{final}}$. An initial value problem (IVP) for a system of the form (8.9) specifies a starting time, t_0, and starting state, u^s. The solution of the initial value problem, or trajectory, is an m-vector valued function, $u(t)$, that satisfies

(8.9) and the initial condition $u(t_0) = u^s$. In the context of dynamic simulation, it would be common to say that the system being modeled evolves through the states $u(t)$ with time.

Notice that the dynamics function f takes as input z, the m-vector of state variables. If we assume that M is the identity matrix (and we will), then the function f outputs an m-vector corresponding to the derivatives of these state variables, all listed in the same order as the input z. So, the dynamics function f is simply a way to calculate the right-hand-sides for the system of ODEs. It tells how things are changing at any given time and state.

In the preceding subsection, we saw that the novelty golf driving range model used two event functions to determine the end time and status of the simulated drive. In general, an IVP with m equations in form (8.9) can also have k events that might occur during the evolution of $u(t)$. These events are specified using a k-vector valued function, $E(t, z)$. A common specification that the jth event has happened in time interval $t_a < t < t_b$ is that $E_j(t, u(t))$ changes sign, i.e. $E_j(t_a, u(t_a))$ and $E_j(t_b, u(t_b))$ have opposite signs.

Software for dynamic simulations, including Matlab, require the mathematical model's equations to be identified as data for the standard first order form described above. In this subsection, we identify the mathematical model of each of the dynamic simulations as an IVP for a system of equations in standard form (8.9). The mass matrix for each of these cases is the identity matrix; so we will not comment specifically on it.

8.4.1 The single population model

The identification of the single population model for form (8.9) is simple:

- $m = 1$
- $p \sim y$
- $f(t, y) = r\,y \;+\; I$
- $t_0 = 2000$; $u^{(s)} = 30750.$

8.4.2 The golf driving range

The identification of the novelty golf range model is complicated by the fact that the basic modeling differential equations in (8.6) include a second derivative of the variables, $x(t)$ and $y(t)$. There is a standard technique of introducing new variables to reduce a system of ODEs to first order form. This means

- $m = 3$
- a more complex connection between the variables of the model x and y and the variables of the form (8.9) y_1, y_2, y_3. The identification is

$$y_1 \sim x \;;\quad y_2 \sim y \;;\quad y_3 \sim dy/dt$$

To ensure that y_2 and y_3 remain consistent with $y_2 \sim y$ and $y_3 \sim dy/dt$ we add the equation $dy_2(t)/dt = y_3(t)$ to (8.6). The rest of the identification is

$$f_1(t,y) = V_x \ ; \quad f_2(t,y) = y_3 \ ; \quad f_3(t,y) = -g \tag{8.10}$$

and the initial conditions are

$$t_0 = 0 \ ; \quad u^{(s)} = (0, 0, V_y)^t \ .$$

The two event functions associated with terminating the golf ball trajectories of this model can be written using standard form variables as a 2-vector valued event function, $E(t,y)$

$$\begin{aligned} E_1(t,y) &= y_2 \\ E_2(t,y) &= \begin{cases} 1 & \text{if } |y_1 - 30| > \frac{1}{2} \text{ or } y_2 > b(t) \\ -1 & \text{otherwise} \end{cases} \end{aligned}$$

8.4.3 The pursuit problem

The identification of the pursuit problem as an IVP for the standard form of a first order system of equations is

- $m = 3$ (for 3-D trajectories)
- $y_1 \sim x_P \ ; \ \ y_2 \sim y_P \ ; \ \ y_3 \sim z_P$
- $t_0 = 0 \ ; \ u^{(s)} = P(0)$

and

$$\begin{aligned} f_1(t,y) &= s_P(x_T(t) - y_1)/dist(t,y) \\ f_2(t,y) &= s_P(x_T(t) - y_2)/dist(t,y) \\ f_3(t,y) &= s_P(x_T(t) - y_3)/dist(t,y) \end{aligned}$$

The pursuit model also needs an event function to determine if and when the pursuer captures the target.

8.5 The Matlab ODE Suite

Dynamic simulations based on IVPs in standard form (8.9) are invariably computed using standard mathematical software. Much of this software is in the public domain and some of the best of it has been incorporated into Matlab. Let $u(t)$ be the solution of a standard IVP. A program for numerically solving the standard IVP for u (typically) computes two arrays for output:

- a column-vector of N_{final} time values, $T = [T_1, T_2, \ldots T_{N_{\text{final}}}]^t$, and

- an array, Y, containing values for the state variables at the corresponding times. This array has N_{final} rows, each row holding the m-vector of state variables. Thus, the kth row is the solution at time T_k. The computed Y values are approximations to the solution u in that
$$Y_{k,j} \approx u_j(T_k) \ . \tag{8.11}$$

Matlab has a suite of 7 programs for computing numerical solutions of IVPs. The suite has a user interface that is common to all programs. This interface is quite flexible to accommodate a range of uses from simple to sophisticated. Version 6 of Matlab made major changes to this interface (compared to version 5), simplifying it significantly as well as extending it a small amount.

These notes provide a brief overview of the new interface using some examples. We do not intend to provide full technical details. The Matlab help is a reference for the many details of the complete interface[5].

8.5.1 The basic form of the ODE suite interface

The basic form of the ODE suite interface assumes that a numerical solution is to be computed for an IVP of the form shown in equation (8.9) with no mass matrix (more precisely, with the mass matrix $M = I$). The input for the basic form for the interface is:

a) the data describing the standard initial value problem based on (8.9) with $M = I$. This data is the dynamics function $f(t, z)$ and the initial state data t_0, u^s.

b) an explicit final time for the numerical solution, t_{final}.

How do we make the dynamics function known to the ODE method? From a programming language point of view, a natural approach is to make one of the method parameters a handle (pointer) to a user-defined function. Then the solver could use this handle to evaluate the function for whatever t and z it likes. This mechanism has been introduced into Matlab in version 6 under the name of *function handles*. Try `help function_handle`. Here, we only discuss user-defined functions (stored in `.m` files), such as

```
function res = myFun(t,y)
   res = t*t+ y*y
```

Matlab assumes that the name of the file containing the function is the function name followed by the `.m` extension (`myFun.m` in the above example)[6]. The handle of a function is denoted by adding the `@` symbol to the start of the function name, e.g. `@myFun`.

[5]In the 'full product family help' documentation distributed with Version 6.5, the description is found in the Contents tree by opening the chain of references `MATLAB`, `Mathematics`, `Differential Equations`, `Initial Value Problems for ODEs`.

[6]In fact, the function name is defined by the file name (minus the .m extension) regardless of the function name in the body of the file.

The Matlab command syntax[7] for computing a numerical solution based on this example dynamics function is then

$$\texttt{[T,Y] = ode}X\texttt{(@sdf,tspan,u0)} \tag{8.12}$$

where X is a short string that specifies which particular solver program to use. For example, $X = 23$ refers to the Matlab command `ode23` (discussed in §9.5). The first few lines of the help for `ode23` (type `help ode23`) explain the second and third input arguments.

```
ODE23  Solve non-stiff differential equations, low order method.
    [T,Y] = ODE23(ODEFUN,TSPAN,Y0) with TSPAN = [T0 TFINAL] integrates the
    system of differential equations y' = f(t,y) from time T0 to TFINAL with
    initial conditions Y0. Function ODEFUN(T,Y) must return a column vector
    corresponding to f(t,y)...
```

During the computation of the numerical solution, `ode`X samples the system's dynamics function many times. The simple relationship between the calling program, `ode`X, and the dynamics function is shown in Figure 28.

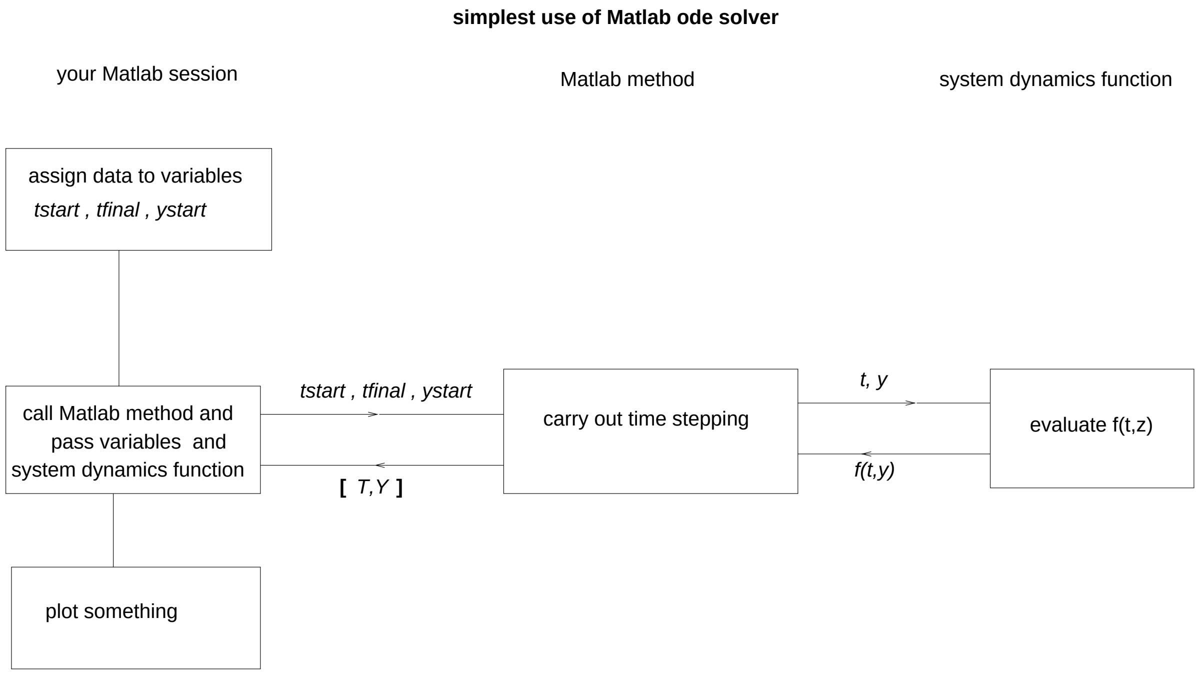


Figure 28: Basic use of Matlab time stepping command

[7]for versions 6 or 7

8.5.2 Extending the basic form of the Matlab ODE suite interface

The Matlab ODE suite has 17 features, called ‘properties’ in the Matlab documentation, that affect the operation of the ODE solvers. Each property has a default value, but the user may want to change some of them in order to

a) compute a numerical solution to an IVP for which $M \neq I$,

b) stop the computation when a certain condition (event) is met, or

c) control the accuracy, or efficiency, of the computation.

How does the ODE suite interface support changing these properties?

The value of each property can be set by a Matlab function called `odeset` (see `help odeset`). This function creates a Matlab internal data structure called an `option structure`, recording the desired options. The input arguments for the `odeset` function are of the form `‘name’, value`, specifying a property-name/property-value pair. A user can change any combination of the 17 properties using a single call to `odeset`; the resulting option structure can then be passed to the appropriate `ode`X command as the 4th argument. The values of individual options in the structure can be obtained by using the Matlab `odeget` command (not commonly necessary).

A common reason for overriding a property of the ODE suite is that the standard form of the model equations includes a mass matrix, M, that is not the identity matrix. In this case, the `Mass` property must be overridden. A second reason is the use of event functions, described in the next section. Scenarios involving controlling the accuracy and efficiency will be discussed in §9.5.2.

Explicitly setting the `Events` property

If an IVP has one, or more, event functions as part of its specification, this is communicated to the ODE suite by appropriately setting the `Events` property. The value of the `Events` property is a function handle for a user-defined Matlab event function; the default is a NULL pointer. The simplest syntax for a Matlab event function is

```
[val, terminal, direction] = eventFunct(t,z)
```

If the IVP in question has, say, 3 events in its specification, then the Matlab `eventFunct` function of this syntax would return `val` as a three-vector giving the values of the three events.

Suppose we wish to use `ode23` to generate a numerical solution to the novelty golf IVP discussed in the preceding section. Communicating the event functions can be done with the following lines of Matlab:

```
myOpts = odeset(‘Events’, @golfEvents);
[T,Y,TEvent,YEvent,EventNum] = ode23(@GolfDyn,tspan,u0,myOpts);
```

The first command sets the value of the `Events` property to the function handle and creates an option structure named `myOpts`. In the second command, the option structure is passed to the `ode` method.

Matlab "help ode23" includes the following description of this command syntax.

```
[T,Y,TE,YE,IE] = ODE23(ODEFUN,TSPAN,Y0,OPTIONS...) with the 'Events'
property in OPTIONS set to a function EVENTS, solves as above while also
finding where functions of (T,Y), called event functions, are zero. For
each function you specify whether the integration is to terminate at a
zero and whether the direction of the zero crossing matters. These are
the three vectors returned by EVENTS: [VALUE,ISTERMINAL,DIRECTION] =
EVENTS(T,Y). For the I-th event function: VALUE(I) is the value of the
function, ISTERMINAL(I)=1 if the integration is to terminate at a zero of
this event function and 0 otherwise. DIRECTION(I)=0 if all zeros are to
be computed (the default), +1 if only zeros where the event function is
increasing, and -1 if only zeros where the event function is
decreasing. Output TE is a column vector of times at which events
occur. Rows of YE are the corresponding solutions, and indices in vector
IE specify which event occurred.
```

If the simulation is intended to end at a time determined by an event, then `TFINAL` of `TSPAN` becomes a safety net to ensure the computation eventually terminates.

An example: the novelty golf driving range

We can use this simple model to demonstrate several of the features of the ODE suite interface. The dynamics function for this model is given in (8.10). There are two model parameters in its definition: V_x, the horizontal velocity of the ball, and g, the vertical acceleration due to gravity. The parameter g is a constant, so its value can be hard-coded into the dynamics function. However, V_x is a key modeling parameter that we want to specify in our call to `ode`X (so that we can easily change it). Figure 28 shows that the user never explicitly calls the dynamics function; the dynamics function is called by the ODE solver. Somehow the value of V_x must be passed through to the dynamics function.

There are two ways (at least) to do this. The simplest is to declare V_x as a `global` variable both in your session and in the dynamics function code. This has the usual problems of global variables as programs get larger and more complex.

The second (and better) way is to expand the parameter list of the dynamics function, adding `Vx` (or some other parameter name) at the end of the default list. So, instead of the default arguments for the dynamics function `(t,y)`, we have `(t,y,Vx)`. But how does `ode`X know that it should supply a value for this extra parameter when it calls the dynamics function? Answer: The parameter list of `ode`X is also extended by one parameter[8]. Making V_x a parameter is the preferred program design route for simulations of any significant size.

Important: Matlab's implementation has the following quirk that shows up in the golf driving range example. The parameter list for Matlab event functions must match that of dynamics functions. You can see the results in the following Matlab function codes for the novelty golf driving range simulation.

[8]making the parameter list of length at least 5

Dynamics Function

```
function dzdt = golf(t, z, Vx)
    % z(1) = x(t)
    % z(2) = y(t)
    % z(3) = y'(t)
    dzdt = [ Vx ; z(3) ; -9.81 ];
```

Events Function

```
function [values, halt, dir] = golfEvents(t, z, Vx)

   % z(1) = x(t); z(2) = y(t); z(3) = y'(t)
   % Vx = horizontal velocity - weird Matlab quirk -
   %          the dynamics function and the events functions MUST have
   %          same number of extra parameters. We don't need Vx here
   %          but we do in the dynamics function, "golf"
   % barrier has average height = bHeight
   %     centered on fieldSize, and fluctuates by up and down by 1.2 meters
   %     every 2 seconds (i.e. with frequency 0.5)

   f = 0.5;         % barrier frequency
   fieldSize = 30;  % meters
   bHeight = 4;     % so barrier fluctuates between 2.8 and 5.2 meters

   values = [0,0]';
   halt = [1,1]';
   dir = [0,0]';

   % Event 1: the ball hits the ground (the ground is located at z(2)=y=0).
   values(1) = z(2);   % change of sign (+ to -) indicates event 1

   % Event 2: the ball hits the barrier
   bDyn   = bHeight + 1.2 * cos(2 * pi * f * t);  % dynamic barrier height
   values(2) = 1;
   if ( abs(z(1) - fieldSize)  < 0.5 ) && ( z(2) < bDyn )
        disp('inside barrier')
        values(2) = -1;
   end;
```

Matlab session driver script

```
   % set initial velocity imparted by the golfer's drive here
   Vx = 22; Vy = 12;        % m/s
   disp(['Vx=' num2str(Vx) ', Vy=' num2str(Vy)]);

   tspan = [0 5]
   initial = [0; 0; Vy]
```

```
% Vx*h is the horizontal distance travelled in time h.
% Set MaxStep so that Vx*MaxStep = 1, so time stepping can't
% pass through a 2 meter wide barrier in one step.

options = odeset('Events',@golfEvents, 'MaxStep', 1/Vx);
[t,z] = ode45(@golf,tspan,initial,options,Vx);
N = length(t)
disp('tfinal, Xfinal, Yfinal');
disp([t(N),z(N,1), z(N,2)]);
```

9 Initial Value Problems

9.1 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 and the physical laws that relate the position, velocity and acceleration of the particular object.

Example 9.1 *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*

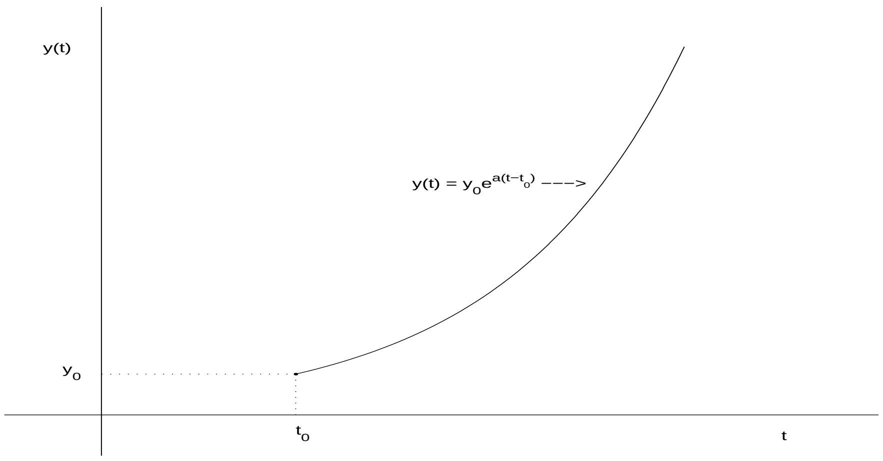
$$y'(t) = a \cdot y(t) \tag{9.1}$$

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

$$y(t) = y_0 \cdot e^{a(t-t_0)} \tag{9.2}$$

is a solution to (9.1). Thus, the mice population 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

$$y'(t) = y(t) \cdot (a - b \cdot y(t)) \tag{9.3}$$

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 IVP, 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 29.

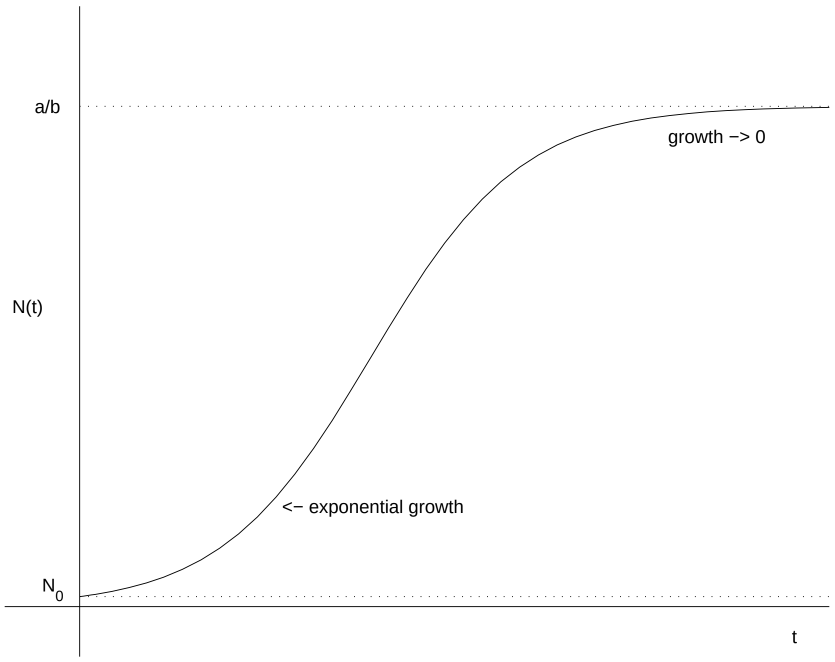


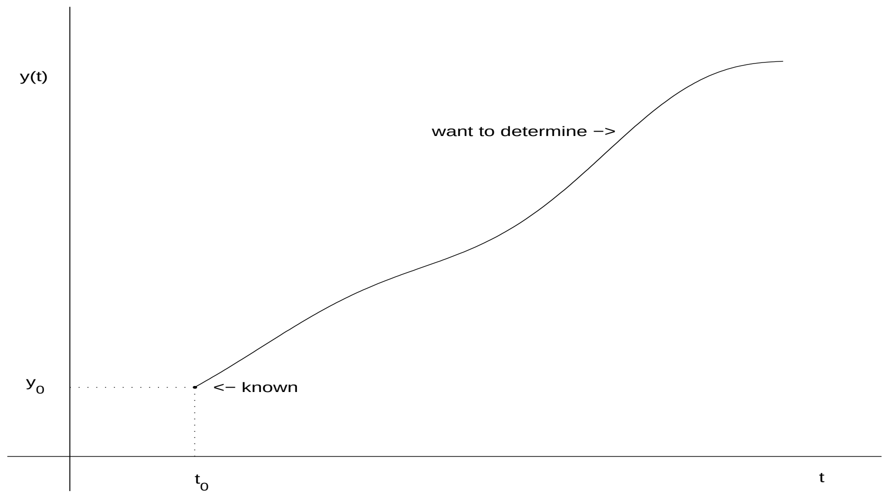
Figure 29: Logistic population growth. The initial population is N_0.

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

$$y'(t) = y(t) \cdot (a(t) - b(t) \cdot y(t)^{\alpha}), \quad y(t_0) = y_0 \tag{9.4}$$

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. □

In the previous example, we wanted 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). But in the third model, no closed-form solution exists and hence other methods need to be used. In this case we can use numerical methods that generate an approximation to the solution.

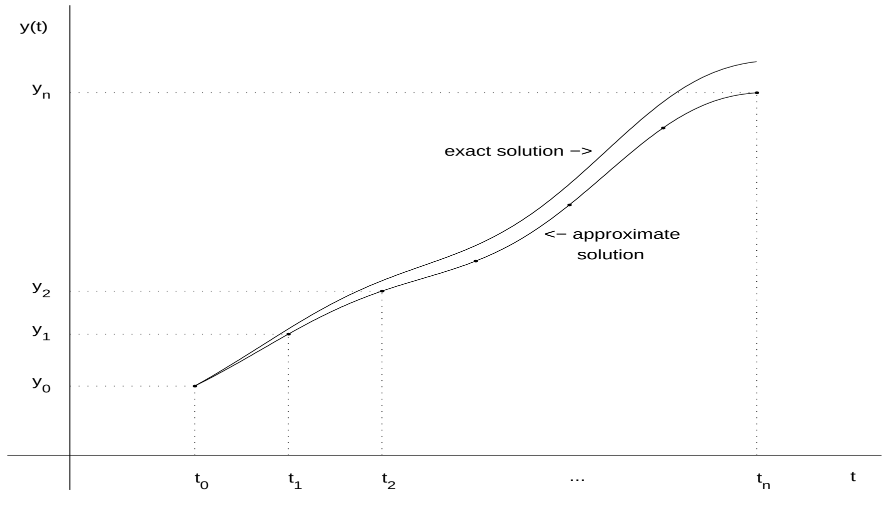


In general we are interested in the problem of numerically solving an equation of the form

$$y'(t) = F(t, y(t)) \quad \text{with} \;\; y(t_0) = y_0 \; . \tag{9.5}$$

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.

Equation (9.5) 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 (9.5) 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).



9.1.1 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 9.2 *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)*

$$\begin{array}{rclrcll} x'(t) &=& v_x(t), & y'(t) &=& v_y(t), & z'(t) = v_z(t), \\ v_x'(t) &=& a_x(t), & v_y'(t) &=& a_y(t), & v_z'(t) = a_z(t) \end{array} \tag{9.6}$$

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

$$(x(0), y(0), z(0)) = (x_0, y_0, z_0) \text{ and } (v_x(0), v_y(0), v_z(0)) = (0, 0, 0). \tag{9.7}$$

Equations (9.6) and (9.7) 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 9.3 *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

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

□

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

$$y^{(n)}(t) = F(t, y(t), y'(t), \ldots, y^{(n-1)}(t)) \tag{9.8}$$

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.

One can also define systems of partial differential equations where there are quantities that depend on more than on 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 gasses on the environment.

9.2 Approximating Methods

There is a variety of methods for determining a numerical solution for a first-order initial value problem. For the most part, there are two components for each method:

- **time step:** find a suitable discrete set of t points $t_0 < t_1 < \cdots < t_N$ for evaluation of our solution;

- **solution step:** compute a set of y values $y_0, y_1, \cdots, y_N$ such that y_n approximates $y(t_n)$, the exact value of the solution at t_n.

For the solution step, there are single-step methods (where y_{n+1} is determined from (t_n, y_n) and the equation for the derivative), and multi-step methods (where y_{n+1} is determined from (t_{n-i}, y_{n-i}) and the equation for the derivative for $i = 0, 1, \ldots, Nsteps$ with $Nsteps > 0$). The solution methods are also classified into *explicit* and *implicit* methods. Explicit methods simply calculate each y_{n+1} from previously calculated points, (t_{n-i}, y_{n-i}). Implicit methods are those where y_{n+1} is computed by solving an algebraic equation involving F.

9.2.1 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 $t_0 < t_1 < \cdots < t_N$. Our goal is to find the y_ns.

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 t-values, $t_0 < t_1 < \cdots < t_N$, 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

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

Isolating the $y(t_{n+1})$ term in the above equation gives the recursive scheme

$$\begin{aligned} y(t_0) &= y_0 \\ y(t_{n+1}) &= y(t_n) + F(t_n, y(t_n)) \cdot (t_{n+1} - t_n) \\ &\qquad \text{for } \ n = 1, \ldots, N-1 \ . \end{aligned}$$

This scheme is called the *Forward Euler* method.

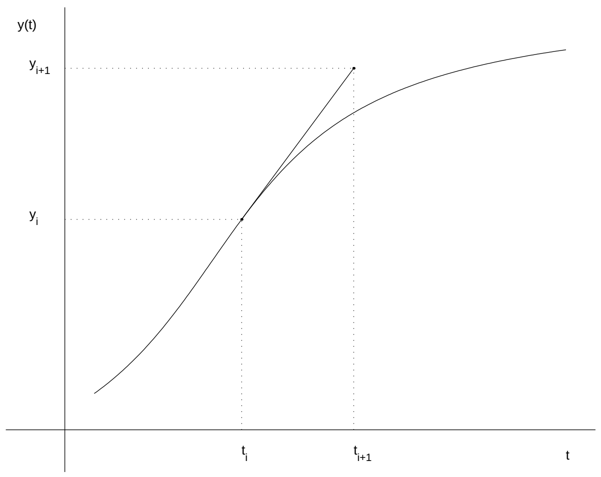


Figure 30: Forward Euler Method

Example 9.4 *Consider the Forward Euler method when applied to the initial value problem*

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

This is the same as equation (9.4) of Example 9.1 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*

$$\begin{aligned} y(0) &= 1 \\ y(t_{n+1}) &= y(t_n) + y(t_n) \cdot (2.5\ t_n - {t_n}^2\ \sqrt{y(t_n)}) \cdot 0.4 \\ &\qquad \text{for}\ n = 0, \ldots, 4\ . \end{aligned}$$

This gives the y*-values*

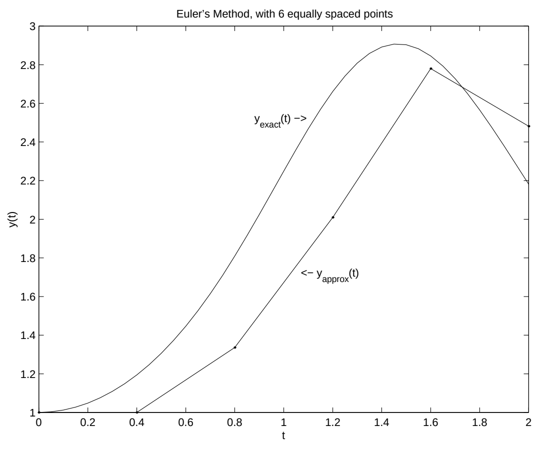
$$y(0) = 1.0,\ y(0.4) = 1.0,\ y(0.8) = 1.34,\ y(1.2) = 2.01,\ y(1.6) = 2.78,\ y(2.0) = 2.48.$$

Figure 31(a) plots this approximate solution along with the exact solution. This approximate solution does not match the exact solution very well because the step size is too large. One would expect that more t*-values would provide a more accurate picture. Figure 31(b) shows the Forward Euler solution with a subdivision of* 30 *equally spaced points.*

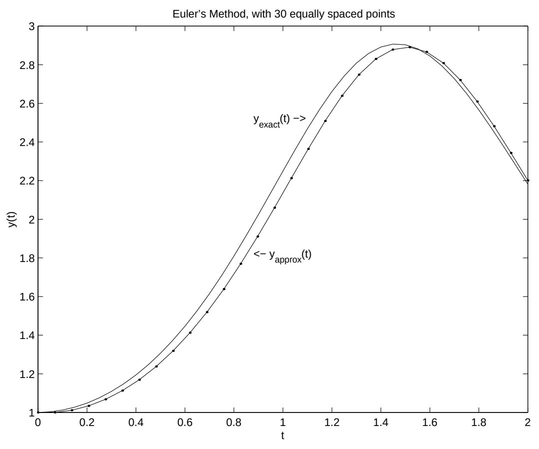
□

Example 9.5 *We can do the same approximation in the case of a system of differential equations. For example, consider the system*

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



(a) Step size = 0.4



(b) Step size = 0.0666

Figure 31: Forward Euler Method

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 population of the prey and y is the population of the predator. The two-dimensional recursion scheme for the Forward Euler method in this case is*

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

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 date where

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

Figure 32(a) plots 600 time steps of the Forward Euler solution with initial conditions $x_0 = 2$ (rabbits per hectare), and $y_0 = 1$ (foxes per 100 hectares).

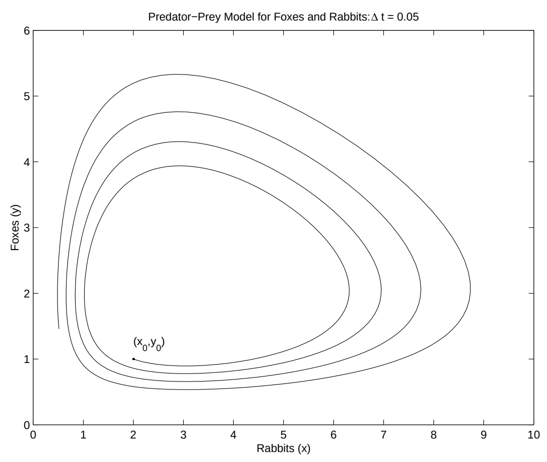
An interesting characteristic of this model (and the chosen parameters) is that the solution settles down to a steady-state cycle (a periodic solution). Figure 32(b) plots an additional 30,000 time steps of the Forward Euler method.

9.2.2 Discrete Approximations

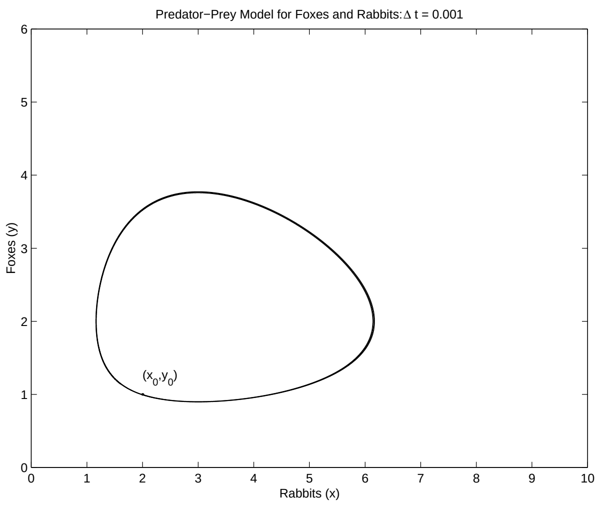
In the last subsection, we presented a recursive scheme based on replacing a derivative by a slope. It is clear that the smaller the interval the better the slope is at approximating a derivative. However it is not clear at this stage how good such an approximation needs to be in order to obtain acceptable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

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

$$y(t) = y(a) + y'(a)h + \frac{y''(a)}{2}h^2 + \frac{y'''(a)}{6}h^3 + \cdots \tag{9.9}$$



(a) 600 time steps



(b) Steady-state

Figure 32: Forward Euler approximation of the Predator/Prey model

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

$$y(t) = y(a) + y'(a)h + \frac{y''(a)}{2}h^2 + \frac{y'''(a)}{6}h^3 + \cdots + \frac{y^{(p)}(a)}{p!}h^p + \frac{y^{(p+1)}(\eta_t)}{(p+1)!}h^{p+1} \qquad (9.10)$$

where η_t is a point between a and t. Often we write equation (9.10) as

$$y(t) = y(a) + y'(a)h + \frac{y''(a)}{2}h^2 + \frac{y'''(a)}{6}h^3 + \cdots + \frac{y^{(p)}(a)}{p!}h^p + O(h^{p+1}) \ . \qquad (9.11)$$

If $a = t_n$ and $t = t_{n+1}$ and $p = 1$, then equation (9.11) becomes

$$y(t_{n+1}) = y(t_n) + y'(t_n)h + O(h^2) \ . \qquad (9.12)$$

which can also be written

$$y'(t_n) = \frac{y(t_{n+1}) - y(t_n)}{h} + O(h) \ . \qquad (9.13)$$

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 proportional to h. It is important to note that this means the error goes to zero as $h \to 0$.

Since $y'(t) = F(t, y(t))$, we can write (9.12) 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. The term $O(h^2)$ is called the *local truncation error* for our formula.

9.2.3 The Modified Euler Method

In order to avoid lots of tedious notation, let us set

$$y'_n = y'(t_n), \quad y''_n = y''(t_n) \text{ and } y'''_n = y'''(t_n) \ .$$

Then the Taylor series expansions give (for $h = t_{n+1} - t_n$)

$$y_{n+1} = y_n + y'_n h + \frac{y''_n}{2} h^2 + \frac{y'''(\eta)}{6} h^3$$

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

$$y''_n = \frac{y'_{n+1} - y'_n}{h} + O(h) \tag{9.14}$$

to obtain

$$\begin{aligned} y_{n+1} &= y_n + hy'_n + \frac{h^2}{2}\left[\frac{y'_{n+1} - y'_n}{h} + O(h)\right] + O(h^3) && (9.15) \\ & && (9.16) \\ &= y_n + h\left(\frac{y'_{n+1} + y'_n}{2}\right) + O(h^3) \ . && (9.17) \end{aligned}$$

We are looking for the solution of

$$y'(t) = F(t, y(t))$$

so we can write

$$y'_n = y'(t_n) = F(t_n, y(t_n)) = F(t_n, y_n) \tag{9.18}$$

and

$$y'_{n+1} = y'(t_{n+1}) = F(t_{n+1}, y(t_{n+1})) = F(t_{n+1}, y_{n+1}). \tag{9.19}$$

Thus we have

$$y_{n+1} = y_n + \frac{h}{2}\big(F(t_n, y_n) + F(t_{n+1}, y_{n+1})\big) + O(h^3). \tag{9.20}$$

We may think of (9.20) as approximating the derivative $y'(t)$ using the average of the slopes at the two end points, as shown in Figure 33.

Since this method uses more information about the dynamics function F, we expect it to be a higher-order method (that is, a higher power in the truncation error term). Thus we have Forward Euler, with

$$y_{n+1} = y_n + hF(t_n, y_n) + O(h^2) \tag{9.21}$$

and

$$y_{n+1} = y_n + \frac{h}{2}\big(F(t_n, y_n) + F(t_{n+1}, y_{n+1})\big) + O(h^3) \tag{9.22}$$

which is called the trapezoidal rule (or Crank-Nicolson) method. Note that y_{n+1} appears only on the left hand side in the Forward Euler method. This is called an *explicit* method.

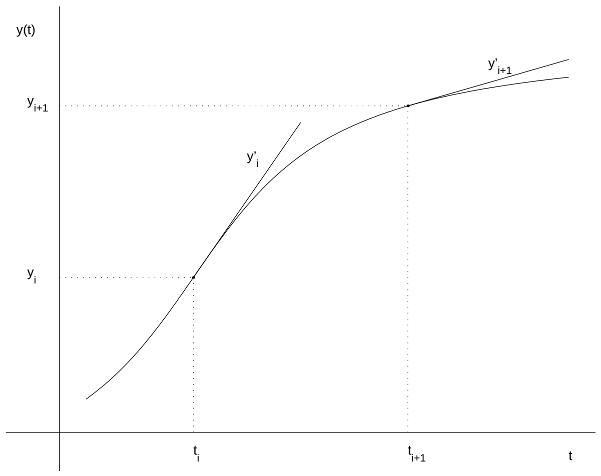


Figure 33: Modified Euler Method

On the other hand, equation (9.22) 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 (9.22) is to combine it with the Forward Euler recursion. That is, we use (9.21) as a first approximation for y_{n+1} and then use this value in the right-hand side of (9.22) for our final value of y_{n+1}. This method is called the *Modified Euler method* and is formally written

$$y^*_{n+1} = y_n + hF(t_n, y_n) \tag{9.23}$$

$$y_{n+1} = y_n + \frac{h}{2}\big(F(t_n, y_n) + F(t_{n+1}, y^*_{n+1})\big) \ . \tag{9.24}$$

We can determine the accuracy of the Modified Euler method as follows. Notice first that

$$y_{n+1} = y_n + hF(t_n, y_n) + O(h^2) = y^*_{n+1} + O(h^2) \tag{9.25}$$

and so the difference $y_{n+1} - y^*_{n+1}$ has error $O(h^2)$. The second-order Taylor approximation of $F(t_{n+1}, y_{n+1})$ is

$$F(t_{n+1}, y_{n+1}) = F(t_{n+1}, y^*_{n+1}) + \frac{\partial}{\partial y}F(t_{n+1}, y_{n+1})(y_{n+1} - y^*_{n+1}) + O\left((y_{n+1} - y^*_{n+1})^2\right) \ . \tag{9.26}$$

Since $y_{n+1} - y^*_{n+1}$ is $O(h^2)$, we have

$$F(t_{n+1}, y_{n+1}) = F(t_{n+1}, y^*_{n+1}) + O(h^2) \ . \tag{9.27}$$

Recall that the Crank-Nicolson method is

$$y_{n+1} = y_n + \frac{h}{2}\big(F(t_n, y_n) + F(t_{n+1}, y_{n+1})\big) + O(h^3) \ . \tag{9.28}$$

Therefore, plugging (9.27) into (9.28) gives

$$\begin{aligned} y_{n+1} &= y_n + \frac{h}{2}\big(F(t_n, y_n) + F(t_{n+1}, y^*_{n+1}) + O(h^2)\big) + O(h^3) && (9.29) \\ &= y_n + \frac{h}{2}\big(F(t_n, y_n) + F(t_{n+1}, y^*_{n+1})\big) + O(h^3) \ . && (9.30) \end{aligned}$$

Thus we have that

$$\begin{aligned} y^*_{n+1} &= y_n + hF(t_n, y_n) && (9.31) \\ y_{n+1} &= y_n + \frac{h}{2}(F(t_n, y_n) + F(t_{n+1}, y^*_{n+1})) + O(h^3) && (9.32) \end{aligned}$$

is an explicit method with a higher-order truncation error than the Forward Euler method. We can also write this as

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

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 $O(h^5)$.

9.3 Global vs. Local Error

In the preceding subsection, we considered just the local truncation error for the 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

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

Therefore we have

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

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.

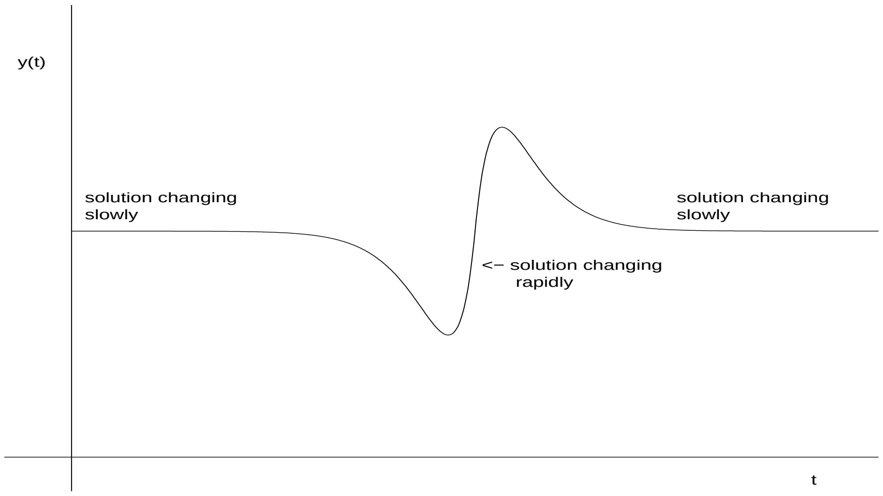
9.4 Practical Issues

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:

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

Software Libraries can be developed for the various methods so that the user is only required to supply a function which evaluates $F(t, y)$ for any (t, y). This is the approach used by Matlab 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

$$y_{n+1} = y_n + hF(t_n, y_n) + O(h^2) \ .$$

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 C 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 Matlab solvers `ode45` and `ode23` work. The `ode23` method combines a second-order and a third-order step with automatic stepsize control, while `ode45` 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 `ode45` 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

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

9.5 Overview of Numerical Methods and the Matlab ODE Suite

In §9.2, we introduced the idea of a numerical solution of an IVP and discussed methods for computing them. We noted that a numerical method has the coupled pair of tasks:

advancing the solution: computing $y^{(n)}$ - which we now designate with a bracketed superscript since it is an m vector, $y^{(n)} = (y_1^{(n)}, y_2^{(n)}, \ldots y_m^{(n)})$

time step size selection: computing t_n

We repeat the conceptual scheme for time stepping methods here.

```
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) $t_{n+1} \leftarrow t_n + h_n$                                        (9.33)
  iii) recompute $h^{cand}$
   iv) $n \leftarrow n + 1$
```

9.5.1 Classification of methods for advancing the solution

The exact solution of a system of ODEs (8.9) satisfies

$$\begin{aligned} y(t_{n+1}) = y(t_n) + \int_{t_n}^{t_{n+1}} y'(s)ds &= y(t_n) + M^{-1}\int_{t_n}^{t_{n+1}} f(s, y(s))ds \\ &= y(t_n) + h_n M^{-1}\text{Avg}(f) \end{aligned} \tag{9.34}$$

where Avg(f) is an average value of the dynamics function over the trajectory in the interval $t_n < s < t_{n+1}$ which is of length h_n, i.e.

$$\text{Avg}(f) = \frac{1}{h_n}\int_{t_n}^{t_{n+1}} f(s, y(s))ds \tag{9.35}$$

If $M \neq I$, then to implement (9.34), a system of linear equations, $Mx = \text{Avg}(f)$, would have to be solved for x and then $y(t_{n+1})$ could be computed as

$$y(t_{n+1}) = y(t_n) + h_n x \ .$$

Of course, even if we know t_n, t_{n+1} and $y(t_n)$, the relation in (9.34) cannot be used to compute $y(t_{n+1})$ exactly because we cannot evaluate the integral term. Actually, there are two difficulties associated with this integral term.

i) We do not know $y(s)$, hence we do not know the integrand function.

ii) Even if we did, we would not usually be able to integrate it by an explicit formula.

Most time stepping methods for advancing the solution can be viewed as approximate evaluations of this integral, or, equivalently, this average. The simplest example is the approximation

$$\text{Avg}(f) \approx f(t_n, y(t_n)) \tag{9.36}$$

which is equivalent to the Forward Euler's method.

There are several standard general classifications of solution computation methods. One is the classification into:

One-step methods in which $(t_{n+1}, y^{(n+1)})$ is computed from $(t_n, y^{(n)})$, (using $f(t, z)$ and M).

Multi-step methods in which $(t_{n+1}, y^{(n+1)})$ is computed from $(t_{n-k}, y^{(n-k)})$ for $k = 0, 1, \ldots, N_{\text{steps}}$, where $N_{\text{steps}} > 0$.

Another standard classification is into:

Explicit methods in which $y^{(n+1)}$ is computed by a formula involving t_{n-j} and $y^{(n-j)}$ and evaluations of $f(t, z)$, $j = 0, 1, ...N_{\text{steps}}$, where $N_{\text{steps}} \geq 0$.

Implicit methods in which $y^{(n+1)}$ is computed by solving an algebraic system of equations that involve $f(t, z)$.

	explicit	implicit
one-step	`ode45`, `ode23`	`ode23s`, `ode23t`, `ode23tb`
multi-step	`ode113`	`ode15s`

Table 2: Classifying the Matlab IVP solvers

The 7 solution members of the Matlab ODE suite fall into the following categories.

In this course, we will be using and studying the explicit one-step methods of the Runge-Kutta family (`ode23`, `ode45`). In these methods, approximations to $\mathrm{Avg}(f)$ are computed using formulae involving h_n, t_n, $y^{(n)}$ and evaluations of $f(t, z)$. They can be viewed as particularly efficient ways to sample $f(t, y(t))$ in order to approximate $\mathrm{Avg}(f)$.

What is the purpose of including all the methods of Table 2 in Matlab? As you might guess, it is because some types of IVPs can be solved efficiently by one of the explicit one-step methods, while others are better computed by one of the alternatives. Note that 'better' here means 'more efficiently' since the time step selection processes of all of them provide error control to meet the user set error tolerances as described in the next subsection. An important property of some IVPs which affects the choice of suitable method is called 'stiffness'. Actually, stiffness is a matter of degree. Explicit methods work very well for IVPs that exhibit little stiffness, but implicit methods are necessary for very stiff IVPs. Then there is an intermediate range of IVPs for which it is less clear. As you might expect, the situation is more complicated for IVPs based on general (nonlinear) systems of differential equations; they can be very stiff for part of their solution history and not stiff for other times.

9.5.2 Time step control II: extending the Matlab ODE interface

Section §9.4 described the way that pairs of Runge-Kutta methods can be used for simultaneously advancing the solution and selecting the time step size, as per Step i) of (9.33). The Matlab command `ode23` uses this technique for a pair of methods of order 2 and 3, which requires a minimum of 3 evaluations of $f(t, z)$ per step. Similarly, `ode45` uses the Runge-Kutta-Fehlberg methods, of orders 4 and 5, requiring a minimum of 6 evaluations of $f(t, z)$ per step. Because it requires twice as many function evaluations per time step, `ode45` can only be more efficient than `ode23` if it can take steps approximately twice as long (or longer) as those of `ode23`.

The primary reason for using pairs of methods like these is that it permits estimation of the local error at each step. If the candidate step size is h_{cand}, and the solution for each method is computed using it:

$$\begin{array}{rl} y^{(n+1)} & \text{computed by the higher order method} \\ y^{*(n+1)} & \text{computed by the lower order method} \end{array} ,$$

then the local error using step size h_{cand} can be estimated by the m-vector

$$LocErr_{est}(h_{cand}) = y^{(n+1)} - y^{*(n+1)} \ . \tag{9.37}$$

The step size is chosen to ensure that the size of $LocErr_{est}$ is less than a specified error tolerance. The basic concept is that

```
if $\max_k |LocErr_{est}(h_{cand})_k| \leq tol$ then
  accept $h_{cand}$ as $h_n$
else
  reduce $h_{cand}$ and try again
end if
```

The Matlab ODE suite has three options for customizing this concept for error control, associated with properties named `AbsTol`, `RelTol`, and `NormControl`. See §8.5.2 for a discussion of 'properties' and options. Here are descriptions of these properties from "help odeset".

```
RelTol - Relative error tolerance  [ positive scalar {1e-3} ]
   This scalar applies to all components of the solution vector, and
   defaults to 1e-3 (0.1% accuracy) in all solvers.  The estimated error in
   each integration step satisfies e(i) <= max(RelTol*abs(y(i)),AbsTol(i)).

AbsTol - Absolute error tolerance  [ positive scalar or vector {1e-6} ]
   A scalar tolerance applies to all components of the solution vector.
   Elements of a vector of tolerances apply to corresponding components of
   the solution vector. AbsTol defaults to 1e-6 in all solvers. See RelTol.

NormControl -  Control error relative to norm of solution  [ on | {off} ]
   Set this property 'on' to request that the solvers control the error in
   each integration step with norm(e) <= max(RelTol*norm(y),AbsTol). By
   default the solvers use a more stringent component-wise error control.
```

Matlab also has several properties that control the form of the output from calling `ode`X. Two of these that are of interest to us are named

`Refine` which takes positive integer values and determines the number of output times per time step

`Stats` which take binary values, i.e.'on' or 'off. This property determines whether `ode`X displays a summary of cost statistics for the computation

In these notes, we have use $h_n = t_{n+1} - t_n$ as the step size, and implied that the output vector, `T(n)`, of Matlab's `ode`X functions is composed of exactly these time values t_n. So the entire stepsize history would be the vector

$$h = \texttt{T(2:length(T)) - T(1:length(T)-1)} \ . \tag{9.38}$$

However, this is only true if `Refine` = 1. The default value for `Refine` is 1 for all methods **except** `ode45`, where its default value is 4. To compute the stepsize history for `ode45` using (9.38), you must use the function `odeset` to set `Refine` = 1 (see §8.5.2).

Exercises

1. Suppose a function $y(t)$ is known at three discrete points t_i, t_{i-1}, t_{i-2}, where $t_i - t_{i-1} \neq t_{i-1} - t_{i-2}$. Use Taylor series arguments to determine a backward difference approximation to y_i'', using three values y_i, y_{i-1}, y_{i-2}. What is your order condition?

2. Verify for the following system of equations

$$\begin{aligned} Y_1' &= Y_2 \\ Y_2' &= -x^2 Y_1 - x Y_2 \end{aligned}$$

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 with respect to t.

$$\begin{aligned} u'' + 3v' + 4u + v &= t \\ u'' - v' + u + v &= \cos t \end{aligned}$$

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

$$\begin{cases} y' = 5y \\ y(0) = 5\ . \end{cases}$$

Show the computation schemes for both methods

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.

Appendix A Conveying Information in Graphs

Figures and graphics can be a powerful way to communicate ideas; they play a key role for this purpose in scientific computation. However, the use of graphics to enhance communication requires effort and skill. And while a good graphics package helps, it is really only a tool. Many other factors play into the successful use of graphics to convey a message (just as a hammer is a useful tool, but one needs expertise to build a house). Lets make a comparison with the task of writing a document using text alone to help explain what is meant by this. Obviously, word processors are a very useful tool for supporting this task. But the effectiveness of the communication comes from the combination of the writer's ability to express themselves and how they use the tool. The writer needs clarity about what is to be conveyed and how to organize it into prose. The word processor doesn't help with this. But creating an effective document can also be strongly enhanced by an understanding of the variety of options that a word processor can offer for the task. Success comes from the combination of author's effort, word processor functionality, author's skill with it and the author's patience, time, and energy.

The same is true of conveying ideas graphically; graphics systems (MATLAB graphics included) are indispensable tools for the task. But to use them successfully, one needs

- skill in creating meaningful images,
- understanding of the capabilities of the tool, and
- patience, time, and energy.

In this section, we hope to introduce some of the basics of how MATLAB graphics can be used to convey ideas effectively by an example based on parametric curve computations.

Figure 34 shows four subplots generated by the MATLAB 'subplot(2,2,X)' command. See the script at the end of this appendix. The plot in the X=1 position (the upper left corner) shows the curve created using piecewise linear interpolation of the data, generated using a single MATLAB "plot" command (and, consequently, all the default image settings for this command). It is not clear what message this plot might be intended to convey; the default plot settings are simply designed to allow a user to quickly and easily display a wide spectrum of data on a screen. Some of the weaknesses of that particular plot for displaying the script letter are:

- The axes are scaled differently, so the figure is distorted.
- The tic mark labelling on the axes would be too small to be readable if the image is shrunk for printing.
- There is no text (no axis variable labelling, no title line) to guide the reader.

This crude picture would need to be accompanied by a lot of text to explain what its message is. The other three subplots show figures customized to convey three specific ideas:

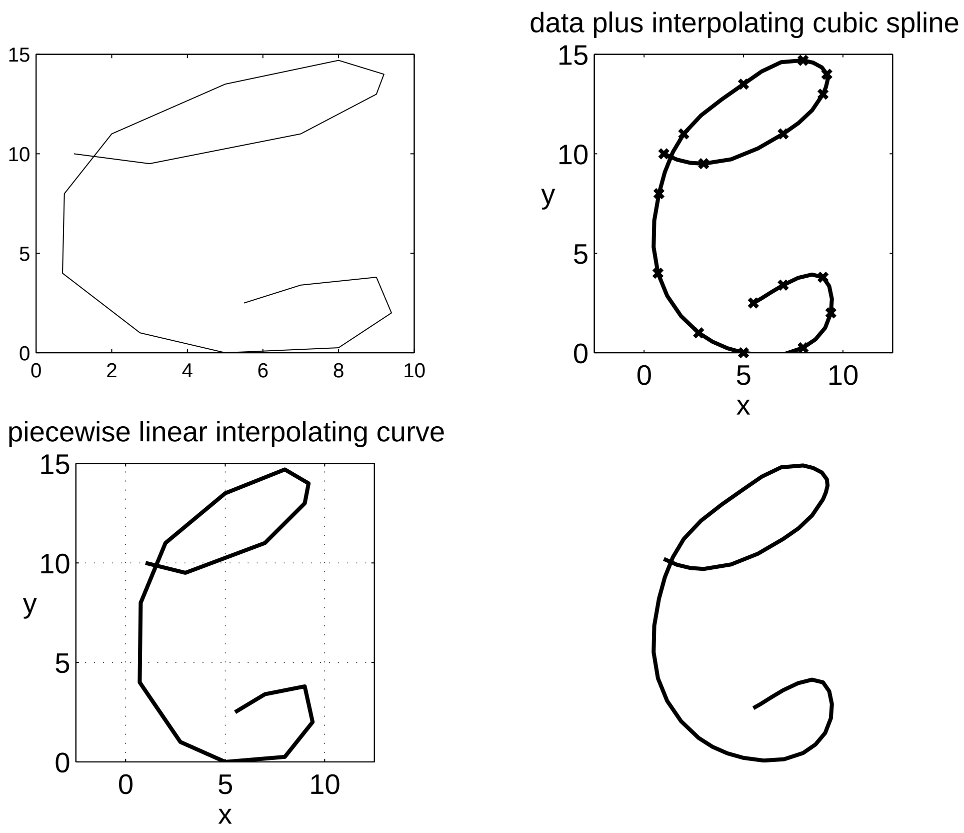


Figure 34: Examples of simple graph customization

subplot(2,2,2) (upper right) The relation of the data to a smooth interpolating parametric curve. The accompanying text might explain that this curve is computed by introducing chord length as a parameter, t, and using two cubic spline interpolations one of (t_i, x_i) and the other of (t_i, y_i) to get the parametric curve.

subplot(2,2,3) (lower left) This plot shows the piecewise linear interpolating parametric curve of the original shape's data points. It includes grid lines and would be appropriate for displaying, or checking, the original data.

subplot(2,2,4) (lower right) This plot shows a refined version of the cubic spline interpolator for the script letter. The encapsulated postscript file for this plot could be useful for an application.

The need to use MATLAB scripts to support graphics

To effectively convey a message using a figure, first you need to be clear about what the message is. Then you need to create a graphic that is focussed on conveying it. It is virtually certain that this second step will require customizing the image to the purpose. So, in MATLAB graphics at least, this means preparing a script to produce the image that includes the customizations that you want. This script allows you to experiment, edit your choices, and change some of the details of your computation.

Here is the script used to make Figure 34:

```
     % load array named C with basic shape data for script C
     %  C(:,1) = parameter values; C(:,2) = x values
initC ;
     % load array  named scriptC with interpolated data for
     % plotting script C
     % scriptC(:,1) = parameter values; scriptC(:,2) = x values
load scriptC ;

figure
     % basic plot using all default properties
subplot(2,2,1),plot(C(:,2)',C(:,3)')

     % read Hanselman and Littlefield -  Handle Graphics, Chapter 30,
     %
     % simple customization to better inform the viewer
     % set ALL future axes labeling to fontsize 14
     % first save current defualt size to be restored later
oldSize = get(0,'DefaultAxesFontSize');
set(0,'DefaultAxesFontSize',14);

subplot(2,2,2),
plotHndl = plot(scriptC(1,:),scriptC(2,:),C(:,2),C(:,3),'x');
     %  the Matlab "help plot" documentation says
     %  ' PLOT returns a column vector of handles to LINE objects
set(plotHndl,'LineWidth',2)
         % customize the axes
axis square
axis([-2.5 12.5 0 15])
         % label stuff
title({'data plus interpolating cubic spline'})
xlabel('x') ;
yHndl =ylabel('y');
set(yHndl,'Rotation',0);

         % plot piecewise linear curve, plus grid
subplot(2,2,3),
plotHndl = plot(C(:,2)',C(:,3)');
set(plotHndl,'LineWidth',2);
grid on;
         % customize the axes
axis([-2.5 12.5 0 15]);
axis square
title('piecewise linear interpolating curve')
```

```
xlabel('x') ;
yHndl =ylabel('y');
set(yHndl,'Rotation',0);

        % plot stand alone curve
subplot(2,2,4),
 plotHndl = plot(scriptC(1,:),scriptC(2,:));
set(plotHndl,'LineWidth',2)
        % customize the axes
axis square;
axis([-2.5 12.5 -0.2 14.8]);
axis off;
   % return to default axes text labeling size
set(0,'DefaultAxesFontSize',oldSize)
```

Appendix B Review of Complex Numbers

We will denote the set of complex numbers by $\mathbb{C}$. A complex number $z \in \mathbb{C}$ can be identified as a pair of real numbers; i.e. $z = (a, b)$. A standard (historical) notation for z involves the symbol i which may be called "the square root of -1" and identified with $\sqrt{-1}$. In this notation z is written $z = a + ib$ and

- i can be identified with $(0, 1)$.
- a is called the real part of z; $a = Real(z)$.
- b is called the imaginary part of z: $b = Imag(z)$.
- In Matlab, the command `display( complex(-3,2) )` will display `-3.000 + 2.000 i`.
- $a - ib$ is called the conjugate of z, sometimes denoted by $\bar{z}$.
- The size of z is $|z| = \sqrt{a^2 + b^2}$, also called the modulus of z.

Arithmetic operations are defined for $\mathbb{C}$: let $z = (a, b)$ and $r = (c, d) \ \in \mathbb{C}$

- $z + r = (a + c, b + d)$,
- $z \times r = (ac - bd, ad + cb)$. Usually we will just write zr for $z \times r$.

A real number, x, is the same as the complex number $(x, 0)$. Notice that if $z = (x, 0)$ and $r = (y, 0)$, then the addition and multiplication of z and r are the same as adding and multiplying in the real number system. In this sense, the complex number system is "backwards compatible", so to speak. Thus, we do not need to make any further distinction between x and $(x, 0)$. You will find that Matlab does not distinguish between these either; try `y = complex(2,0)` in Matlab. Some other interesting relations from the above definitions are $z + \bar{z} = 2\ Real(z)$, and $z\bar{z} = |z|^2 = a^2 + b^2$.

We can form a vector of N complex numbers, $u = (u_1, u_2, \ldots, u_N)^t$, and

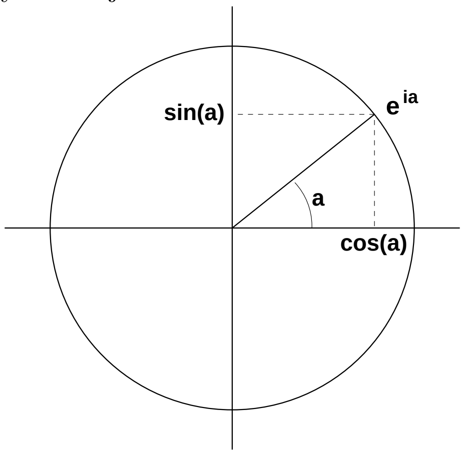
- multiply u by a number $p \in \mathbb{C}$.
- add two complex vectors together. In fact, we can make linear combinations of them. i.e. for two complex vectors u and v (of the same size and shape) and two complex numbers e and f, $eu + fv$ is also a complex vector.
- take the inner product of two complex vectors u and v using

$$(u, v) = \sum_{j=1}^{N} u_j \bar{v}_j \ . \tag{B.1}$$

We denote the set of complex N vectors by $\mathbb{C}^N$.

In the study of the discrete Fourier transform, the following formula for the exponential function with a pure imaginary argument plays a major role.

$$e^{ia} = (\cos(a), \sin(a)) = \cos(a) + i\sin(a) \quad \text{(B.2)}$$



e^{ia} = cos(a) + i sin(a)

The figure adjacent to (B.2) shows the unit circle[9] in the complex plane and the complex number e^{ia}. It shows how a can be identified with the angle at the origin (measured in radians). An a-value of 2π represents one full revolution, returning back to the complex number 1. From this, it can be seen that e^{ia} is a periodic function of a, i.e. $e^{i(a+2\pi k)} = e^{ia}$ for *any* integer k.

This exponential function retains the usual properties of exponentials, connecting addition and multiplication

$$e^{i(a_1+a_2)} = e^{ia_1}\, e^{ia_2} \ . \tag{B.3}$$

Also, the complex conjugate corresponds to simply reversing the sign of the exponent:

$$\text{If } u = e^{ia}, \ \text{ then } \bar{u} = e^{-ia} \ . \tag{B.4}$$

Appendix B.1 Roots of Unity

Let W be the complex number

$$W = e^{2\pi i/N} \tag{B.5}$$

for some integer N. Notice that

$$W^N = 1 \ . \tag{B.6}$$

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. So is $e^{-\pi i/4}$. In general, if v is a root of unity, so is $\bar{v}$ (the conjugate of v). According to (B.6), W is an Nth root of unity. Then so is $e^{-2\pi i/N} = \bar{W}$. The complex number $i = e^{\pi i/2}$ is a 4th root of unity.

It is helpful to see that these 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

[9]circle of radius 1

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.2)), $e^{i3\pi/4}$, …, $e^{i7\pi/4}$, all of which are 8th roots of unity. Figure 35 plots these values on the unit circle in the complex plane.

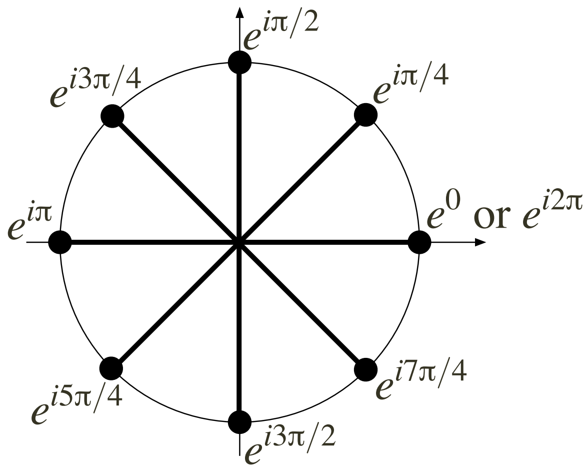


Figure 35: The 8th roots of unity plotted on the unit circle in the complex plane.

Exercises

1. Show that $e^{2\pi k\, i/N}$ is an Nth root of unity for every integer, k.
2. Despite the result in part B.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?

Appendix B.2 Orthogonality Property

For any $z \in \mathbb{C}$ and $z \neq 1$, we have the following formula for the sum of a geometric series:

$$1 + z + z^2 + \cdots + z^{N-1} \;=\; \frac{z^N - 1}{z - 1}\,. \tag{B.7}$$

If $z = 1$, then the above sum is simply equal to N. This leads to an important mathematical basis for the usefulness of Fourier transforms. Let W be the Nth root of unity defined by (B.5). Define $W(k) \in \mathbb{C}^N$ to be the complex N-vector of the powers of W^k, for $0 \leq k \leq N - 1$, as

$$W(k) = \left(1, W^k, W^{2k}, \ldots, W^{(N-1)k}\right)^t\,. \tag{B.8}$$

Then, the jth element of $W(k)$ (where $j \in \{0, \ldots, N-1\}$) is $W(k)_j$, which equals W^{kj}.

The inner product of two complex vectors $W(k)$ and $W(l)$ is

$$\begin{aligned}(W(k), W(l)) &= \sum_{j=0}^{N-1} W(k)_j \overline{W(l)}_j \\ &= \sum_{j=0}^{N-1} W^{(k-l)j}.\end{aligned}$$

Note the bar above $W(l)_j$, indicating the complex conjugate of $W(l)_j$. If $k = l$, then $(W(k), W(l)) = N$. Otherwise, by formula (B.7),

$$(W(k), W(l)) = \frac{W^{(k-l)N} - 1}{W^{k-l} - 1} = 0$$

because the numerator is zero and the denominator is nonzero (be sure to prove this to yourself). To summarize, the inner product evaluates to

$$(W(k), W(l)) = \begin{cases} 0 & \text{if } k \neq l \\ N & \text{if } k = l. \end{cases} \tag{B.9}$$

In other words, $W(k)$ and $W(l)$ are orthogonal complex vectors when $k \neq l$.