James Madison University
Department of Mathematics & Statistics

Math 248:
Computers and Numerical Algorithms
Part II: Algorithms and Numerics

AUTHORS:
C. David Pruett (pruettcd@jmu.edu)
Anthony Tongen (tongenal@jmu.edu)

January 9, 2014

Math 248, as the course name suggests, has a dual focus on *computers* and *algorithms*. In the course, sophomore-level students will learn 1) structured programming in a high-level programming language, and 2) useful algorithms for performing numerical tasks such as rootfinding, solving systems of linear equations, integrating and differentiating, and interpolating. To our knowledge, Math 248 is unique to JMU. We know of no other university that offers programming and numerical methods seamlessly in one course.

Packaging programming and numerical methods together affords many opportunities. It also carries many pitfalls. The purpose of this coursepak is to structure the course so that students and instructors can take advantage of the opportunities while avoiding the pitfalls.

Contents

1 Part II: Numbers and Their Representations

A *positional number system* is one in which the value of a *digit* is determined by its position relative to a "decimal" point. For example, in decimal notation, the number 1047.321 means

$$1\times 10^3+0\times 10^2+4\times 10^1+7\times 10^0+3\times 10^{-1}+2\times 10^{-2}+1\times 10^{-3} \tag{1}$$

In the example above, the *base* is 10 and the same for all terms, the *exponent* is determined by place relative to the decimal point, and the coefficients of each term are given by the respective *digits* of the number.

REMARKS:

1. The origin of our common base 10 number system derives from humans having ten fingers, the medical term of which is *digits*.
2. The idea of assigning the value of a number to its position originates from the Arabic system.
3. Our numerals (symbols) originate from the Sanskrit.
4. Not all number systems are positional in the same way. For example, in Roman numerals IV means 4 and VI means 6, I=1 being added or subtracted from V=5 depending upon whether it follows or precedes the V, respectively. Such a system is awkward at best and ill-suited for electronic computing.

In general, a *positional number system* can be devised for any integer base except 0. The common ingredients of positional number systems are

1. An integer base b. Some common bases are
   - ($b=2$) binary
   - ($b=8$) octal
   - ($b=10$) decimal
   - ($b=16$) hexadecimal
2. A set of b symbols (numerals) with values 0 to $b-1$. For the common number systems above, the respective numerals are
   - $\{0,1\}$
   - $\{0,1,2,3,4,5,6,7\}$
   - $\{0,1,2,3,4,5,6,7,8,9\}$
   - $\{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$

3. A positional representation such that, in general,

$$(\, a_n \, a_{n-1} ... a_1 \, a_0 \, . \, a_{-1} \, a_{-2} \, a_{-3} \, ... \,)_b$$

means

$$a_n \cdot b^n + a_{n-1} \cdot b^{n-1} + ... + a_1 \cdot b^1 + a_0 \cdot b^0 + a_{-1} \cdot b^{-1} + a_{-2} \cdot b^{-2} + a_{-3} \cdot b^{-3} + ...$$

where the a_i are appropriate numerals.

What are the numerals of a tertiary (base 3) number system? What is the decimal value of the digit D in a hexidecimal system?

1.1 Converting Integers Between Bases

If we know the value of an integer in one base, say b_1, how do we determine its value in another base, say b_2? Quite often, we want to know the decimal value of an integer in base b, or vice versa, so let's start with these conversions. From now on, assume that b is a *positive* integer (that is, $b \in Z$ and $b > 0$).

1.1.1 Base b to Base 10 (Decimal) Integers

Let's start with a couple of specific examples and see if we can then generalize to derive an algorithm.

EX1: Convert $(257)_8$ to a decimal integer.

$$\begin{aligned} (257)_8 &= 2(8)^2 + 5(8)^1 + 7(8)^0 \\ &= 2(64) + 5(8) + 7(1) \\ &= 128 + 40 + 7 \\ &= (175)_{10} \end{aligned}$$

EX2: Convert the binary integer (1011011) to a decimal integer.

$$\begin{aligned} (1011011)_2 &= 1(2)^6 + 0(2)^5 + 1(2)^4 + 1(2)^3 + 0(2)^2 + 1(2)^1 + 1(2)^0 \\ &= 1(64) + 0(32) + 1(16) + 1(8) + 0(4) + 1(2) + 1(1) \\ &= 64 + 0 + 16 + 8 + 0 + 2 + 1 \\ &= 91_{10} \end{aligned}$$

Now let's think generically.

EX 3: Convert the integer $(a_n a_{n-1} ... a_1 a_0)_b$ into a decimal integer.

$$(a_n a_{n-1} ... a_1 a_0)_b = a_n (b)^n + a_{n-1} (b)^{n-1} + ... + a_1 (b) + a_0$$

What is the mathematical name for the expression on the right-hand side of the equation above? Yes, a *polynomial* in b.

REMARKS: Converting integers from any base b other than 10 into base 10 is equivalent to evaluating a polynomial in b at the value of the base.

ASIDE: Algorithms for evaluating polynomials

Consider the nth order polynomial in x,

$$P_n(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$$

How many operations (multiplies or adds) does it take to evaluate $P_n(x)$ given the coefficients a_i and a value of x? It depends on how you organize the work. If we approach the problem naively, the operation count can be quite high. For example, here's how *not* to undertake the problem, by the brute force approach.

term	multiplies
$a_n x^n = a_n \cdot x \cdot x \cdot x \cdot x \ldots \cdot x$	n
$a_{n-1}x^{n-1} = a_{n-1} \cdot x \cdot x \cdot \ldots \cdot x$	$n-1$
...	...
$a_1 x = a_1 \cdot x$	1
$a_0 = a_0$	0
total	$\frac{n}{2}(n+1)$

Table 1: Number of multiplies to evaluate $P_n(x)$ by brute force.

The total number of multiply operations in the table above was obtained by the trick of Gauss mentioned in Part I: Chapter 4. In addition to the $\frac{n}{2}(n+1)$ multiplies, there are an additional n add operations necessary to sum the $n+1$ terms. Therefore, the total count for brute force evaluation of a polynomial is $\frac{n}{2}(n+3)$ operations. If n is a moderately large number, the operation count is dominated by the term $\frac{n^2}{2}$, in which case, we say that the brute force algorithm has $O(n^2)$ operations ("order n squared"). In practical terms, this means that if n doubles in size, the operation count *quadruples* in size. Can we do better than this? Yes. A much better algorithm is *Horner's method*. Let's see how it works for a specific example and then generalize.

EX: Nested form of a polynomial.

Consider the fourth-order polynomial $P_4(x) = 3x^4 + 5x^3 + 7x^2 + 8x + 1$. This can be written in *nested form* as follows:

$$P_4(x) = 1 + x\{8 + x[7 + x(5 + x \cdot 3)]\} \tag{2}$$

Now suppose, for the sake of argument, that $x = 2$. Let's evaluate $P_4(2)$. This is most easily accomplished by starting in the innermost nest, replacing the x by 2, multiplying 3 by that 2, and then adding the 7 to obtain 11. The process then repeats itself as we work our way from inside to outside, as follows:

$$2(3) = 6 \quad ; \quad 6 + 5 = 11$$

$$2(11) = 22 \quad ; \quad 22 + 7 = 29$$
$$2(29) = 58 \quad ; \quad 58 + 8 = 66$$
$$2(66) = 132 \quad ; \quad 132 + 1 = 133$$

To evaluate the value of each nest, we must perform one multiplication by 2 and one add, or two operations in total. For a fourth-order polynomial, there are four nests, or eight operations. It appears that the operation count is exactly $2n$ for an nth order polynomial. We will verify this in a moment.

Generalizing the procedure for arbitrary n, we store the highest order coefficient in a register, multiply that coefficient by the value of x and then add to the product the next-highest order coefficient. This result replaces the value in the register, and the process repeats in descending order until the last coefficient, a_0, has been added. There the process terminates. Formalizing this procedure, we obtain the algorithm for Horner's method.

ALGORITHM 1.1: Polynomial evalution by Horner's method

input order n
input $n+1$ *coefficients* a_i, $i = 0, 1, ..., n$
input value of x
$t \leftarrow a_n$ *(initialize temporary storage register)*

repeat for $i = n-1, n-2, ..., 0$ *(descending order)*
| $t \leftarrow t \cdot x + a_i$
| ___

output t *(which contains value of* $P_n(x)$*)*

The operation count is easy to obtain from ALG 1.1. There is a single *decrementing* loop (one that runs backwards). The loop is executed n times. Why? The only operations within the loop are one multiply and one add, or two operations per iteration. Thus the total operation count is exactly $2n$. Compare this to the brute force method when n is a large number. Which algorithm is preferable?

REMARKS: Comparing the brute force algorithm with Horner's method provides yet another example of algorithms that are mathematically equivalent (they give the identical mathematical results) but are not computationally equivalent, in this case because the latter is far more efficient than the former in terms of operations.

Horner's algorithm is ideal for evaluating polynomials. It follows then that Horner's algorithm is also ideal for converting integers in base b to decimal integers. Let's revisit a previous example.

EX: Convert the octal integer $(257)_8$ to a decimal integer using Horner's method.

$n \leftarrow 2$
$a_0 \leftarrow 7$, $a_1 \leftarrow 5$, $a_2 \leftarrow 2$
$b \leftarrow 8$
$t \leftarrow a_2$ *(initialize temporary storage register)*

repeat for $i = 1,0$ *(descending order)*
$| \; t \leftarrow t \cdot b + a_i$
$|$ ___
output t *(which contains value of* $P_n(x)$*)*

Confirm that t is initialized to 2. Further confirm that at the end of iteration $i = 1$, $t = 21$, and at the end of iteration $i = 0$, $t = 175$.

HW1: Convert $(BC123)_{16}$ to a decimal integer using nested polynomial evaluation.

1.1.2 Converting Base 10 Integers to Base b

We now wish to go in the opposite direction. That is, given an integer in base 10, can we find and algorithm for conversion to base b? Of particular interest is conversion to base 2 (binary), for this is the favored base of machine compuation. Before we proceed, however, there is an important principle that should be stated explicitly. Suppose b_1 and b_2 are two different integer bases; that is, b_1 and b_2 are both integers, but $b_1 \neq b_2$. If

$$(\text{integer\_part.fractional\_part})_{b_1} = (\text{integer\_part.fractional\_part})_{b_2}$$

then

$$(\text{integer\_part})_{b_1} = (\text{integer\_part})_{b_2}$$

and

$$(\text{fractional\_part})_{b_1} = (\text{fractional\_part})_{b_2}$$

The practical implication of the principle above is that we can divide and conquer when converting decimal numbers to any other base. We will first convert the integer part of the number and then follow up by converting the fractional part.

Let's start with a specific example:

EX: Convert $(375)_{10}$ to octal (base 8). There is no reason to presuppose that the number of digits will be the same in two different bases (although in this case, they are the same.) Thus, to start

$$\begin{aligned} (375)_{10} &= (...a_4a_3a_2a_1a_0.)_8 \\ &= a_0 + a_1 \cdot 8 + a_2 \cdot 8^2 + a_3 \cdot 8^3 + ... \end{aligned} \tag{3}$$

What happens if we now divide both sides of Eq. 3 by the value of the base? In this case we obtain

$$\frac{375}{8} = 46\frac{7}{8} = \underline{\frac{a_0}{8}} + a_1 + a_2 \cdot 8 + a_3 \cdot 8^2 + a_4 \cdot 8^3 + ... \tag{4}$$

where it is understood that the expression on the left-hand side of Eq. 4 is a base 10 (decimal) number. Note that the exponents denoting the powers of the base on the right-hand side each dropped by one. in each term. Moreover, the underlined term must be a proper fraction, and the remaining terms give a whole number. Why is that? We can now invoke our guiding principle. The left- and right-hand sides of Eq. 4 denote the same number expressed in two different bases. We may therefore independently equate the fractional parts and the whole parts. By equating the fractional parts, we obtain

$$\frac{7}{8} = \frac{a_0}{8} \Rightarrow a_0 = 7 \tag{5}$$

which gives us the lowest order digit of the base 8 number we desire. Similarly, by equating the integer parts we obtain

$$46 = a_1 + a_2 \cdot 8 + a_3 \cdot 8^2 + a_4 \cdot 8^3 + ... \tag{6}$$

At this point, the process is repeated. Dividing both sides of Eq. 6 by the base, we obtain

$$\frac{46}{8} = 5\frac{6}{8} = \underline{\frac{a_1}{8}} + a_2 + a_3 \cdot 8 + a_4 \cdot 8^2 + ... \tag{7}$$

from which it is inferred that $a_1 = 6$ and

$$5 = a_2 + a_3 \cdot 8 + a_4 \cdot 8^2 + ... \tag{8}$$

Finally, dividing both sides by the base one more time gives $a_2 = 5$ and a whole part of zero, the latter of which implies $a_3 = a_4 = a_5 = ... = 0$. Thus, when the whole part vanishes, our job is done.

To summarize, the base 8 digits, obtained from lowest order to highest, are $a_0 = 7$, $a_1 = 6$, and $a_2 = 5$, from which we conclude that $(375)_{10} = (567)_8$.

The process above, with all the details, is useful but cumbersome. Can we strip the process to its bare essence to derive an algorithm? At each step, we divide a base 10 integer by the new base and compute the remainder. The remainder yields the desired digit in the new base. The process is repeated anew with the new whole part. When the whole part vanishes, we are done. The steps are shown below, working *from the bottom upwards*.

8 | 0 R5 (a_2)
8 | 5 R6 (a_1)
8 | 46 R7 (a_0)
8 | 375

From the "bare bones" above, we derive the following algorithm:

ALGORITHM 1.2: Base 10 to base b integer conversions

input base-10 integer $n > 0$
input desired new integer base $b > 0$
$i \leftarrow 0$ *(keeps track of index of coefficient)*

repeat until $n = 0$

| *compute remainder of* (n/b)
| $a_i \leftarrow$ *remainder*
| $n \leftarrow [|n/b|]$ *(greatest integer function)*
| $i \leftarrow i+1$

—

output $a_k,\ k = i,...,0$ *(in descending order)*

HW2: Convert $(375)_{10}$ to binary.

1.2 Converting Fractions Between Bases

Thus far, we have been concerned about converting integer values from one base to another. We now turn attention to fractional values expressed in different bases.

1.2.1 Converting Fractions from Base 10 to Base b

Once again, let's start with a specific example.

EX: Convert $(0.35)_{10}$ to octal (base 8).

$$\begin{aligned}(0.35)_{10} &= (0.a_{-1}a_{-2}a_{-3}...)_8 \\ &= a_{-1}\cdot 8^{-1} + a_{-2}\cdot 8^{-2} + a_{-3}\cdot 8^{-3} + ... \end{aligned} \tag{9}$$

What happens if we now *multiply* both sides of Eq. 9 by the value of the base? In this case

$$8(0.35) = 2.80 = \underline{a_{-1}} + a_{-2}\cdot 8^{-1} + a_{-3}\cdot 8^{-2} + ... \tag{10}$$

where the underlined term is a whole number, and the remaining terms sum to a proper fraction. Why is that true? Once again, by independently equating the integer and fractional parts on the left and right sides of Eq. 10, we see that $a_{-1} = 2$ and

$$0.80 = a_{-2}\cdot 8^{-1} + a_{-3}\cdot 8^{-2} + ... \tag{11}$$

Multiplying both sides by the base again yields

$$8(0.8) = 6.4 = \underline{a_{-2}} + a_{-3}\cdot 8^{-1} + a_{-4}\cdot 8^{-2} + ... \tag{12}$$

from which we learn that $a_{-2} = 6$ and

$$0.4 = a_{-3}\cdot 8^{-1} + a_{-4}\cdot 8^{-2} + ... \tag{13}$$

Continuing in the same fashion,

$$8(0.4) = 3.2 = \underline{a_{-3}} + a_{-4}\cdot 8^{-1} + a_{-5}\cdot 8^{-2} + ... \Rightarrow a_{-3} = 3 \tag{14}$$

$$8(0.2) = 1.6 = \underline{a_{-4}} + a_{-5} \cdot 8^{-1} + a_{-6} \cdot 8^{-2} + ... \Rightarrow a_{-4} = 1 \tag{15}$$

and

$$8(0.6) = 4.8 = \underline{a_{-5}} + a_{-6} \cdot 8^{-1} + a_{-7} \cdot 8^{-2} + ... \Rightarrow a_{-5} = 4 \tag{16}$$

At this point, the pattern begun in Eq. 12 repeats itself. We conclude $(0.35)_{10} = (0.26314\overline{6314})_8$. This leads to an interesting observation: a "decimal" fraction that terminates in one base may repeat when expressed in another base.

Once again, the details look somewhat complicated, but the essence of the procedure is simple. At each step multiply by the base. The integer part of the result gives the next successive coefficient. Repeat the process with the remaining fractional part.

index i	product	integer part = a_i	fractional part
-1	8(0.35)	2	0.80
-2	8(0.80)	6	0.40
-3	8(0.40)	3	0.20
-4	8(0.20)	1	0.60
-5	8(0.60)	4	0.80
-6	8(0.80)	6 (pattern replicates)	0.40

Table 2: Essence of conversion of decimal fraction 0.35 to base 8.

From Table 2 we can infer an algorithm for converting as base-10 fraction to any other integer base. Here is the algorithm written in pseudocode:

ALGORITHM 1.3: Base 10 to base b fraction conversions

input base-10 fractional value as real $x > 0$
input value of base $b > 0$ *as an integer*
$i \leftarrow -1$ *(keeps track of index of coefficient)*
input i_{min} *(lower limit of index)*

repeat until $x = 0$ *or* $i < i_{min}$
| $t \leftarrow b \cdot x$
| $a_i \leftarrow$ *integer_part(t)*
| $x \leftarrow$ *real_part(t)*
| $i \leftarrow i - 1$
| ___

output a_k, $k = -1, -2..., i$ *(in descending order)*

1.2.2 Converting from Base b Fractions to Base 10

Fortunately, with a bit of cleverness, we can use what we have already learned. A specific example will illustrate how.

EX: Convert $(0.257)_8$ to decimal notation.

Consider that

$$\begin{aligned}(0.257)_8 &= 2\times 8^{-1}+5\times 8^{-2}+7\times 8^{-3}\\ &= \frac{1}{8^3}\underline{[2\cdot 8^2+5\times 8^1+7\times 8^0]} \qquad (17)\\ &= \frac{175}{8^3}\\ &= (.341796875)_{10}\end{aligned}$$

The underlined term is a whole number $(257)_8$, whose decimal equivalent is 175, obtained by ALG 1.1. (See EX 1 of Section 1.1.1).

Here then are the essential steps of the algorithm:

1. Shift the "decimal" point n places to the right until the fraction is whole.
2. Convert the base-b whole number using ALG 1.1 (Horner's method).
3. Divide the result by b^n to adjust for the shift in Step 1.

1.3 Summary

The following guidelines summarize the conversions of numbers from base b to base 10 and vice versa:

1. To convert $(a_n a_{n-1}\ldots a_1 a_0.a_{-1}a_{-2}\ldots a_{-m})_b$ to base 10:
   (a) Move the "decimal" point m places to the right until the number is whole.
   (b) Use ALG 1.1 (Nested Polynomial Evaluation or Horner's method) to convert the whole number.
   (c) Adjust for the shift of the "decimal" point by dividing the result by b^m.
2. To convert $(a_n a_{n-1}\ldots a_1 a_0.a_{-1}a_{-2}\ldots a_{-m})_{10}$ to base b:
   (a) Treat the integer and fractional parts separately.
   (b) Convert the integer part by ALG 1.2 using successive divisions by b.
   (c) Convert the fractional part by ALG 1.3 using successive multiplications by b.

1.4 Exercises

1. Convert $(375)_{10}$ to base-7.
2. Convert $(BC123)_{16}$ to base-10.
3. Convert $(104)_5$ to base-10.
4. Convert $(375.1)_{10}$ to binary.
5. Convert $(8.1)_{10}$ to base-2 (binary).
6. Convert $(BC123.A)_{16}$ to base-10.
7. One way to convert long binary numbers to decimal is to first convert them to octal. Basically, you want to group the binary digits in threes. Here is the process: $(1101111.11001)_2 = (157.62)_8$ where the 7 comes from the first three digits to the left of the decimal (111), where the 5 comes from the next three digits (101) in the number, etc. Using this method, convert $(10110011.1110101101101)_2$ to decimal.
8. Using your knowledge of addition of base-10 numbers, experiment with how binary addition works. Write down rules used to add binary numbers.

2 Part II: Machine Representations of Numbers

Computers can represent neither all integers nor all real numbers for the simple reason that there is an infinity of integers and an infinity of reals, but computer memory is finite. In this section we will find out exactly what numbers can be represented and how those representations are stored. Computers treat integers and reals differently; accordingly the two subsections of this Chapter deal with machine integers and floating-point numbers, respectively.

Before turning to specifics, a wee bit of history. Modern digital computers store numbers internally in binary form. The one most often credited with the suggestion that digital computers exploit binary arithmetic was John von Neumann. Von Neumann (1903-1957), a child prodigy, was born in Budapest, Hungary. He immigrated to the United States in 1930, as one of the six original mathematicians and scientists, all brilliant, of Princeton University's Institute for Advanced Studies (IAS). Following natualization as a US citizen, von Neumann became deeply involved in the Manhattan Project to build the atomic bomb. Von Neumann, along with Alan Turing, is considered a pioneer in computational technology. He favored binary arithmetic for machine computation because it simplified electronic circuitry. In particular, the simplest switches have only two positions: on and off. Thus, any number could be represented in binary by sufficiently long strings of switches, where the bits 0 and 1 are associated with the off and on positions of a switch, respectively.

Binary arithmetic is easy, but it takes a little practice. It might help to start with a table (Table 4) of the first few integers in binary.

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001

Table 3: First few nonnegative integers in binary.

Note: If you have forgotten how to convert decimal integers to binary, review the previous section once again. Let's now consider the basic operations of addition and multiplication in binary. Here is an example of each.

EX1: Addition in binary

$b = 10$	$b = 2$
11	1011
+7	+111
18	10010

EX2: Multiplication in binary

$b = 10$	$b = 2$
11	1011
$\times$ 7	$\times$ 111
77	1011
	1011
	1011
	1001101

To complete how a four-function digital calculator works, we should also give examples of the operations of subtraction and division. These, however, vary from machine to machine. For example, top-of-the-line supercomputers such as those manufactured by Cray perform division by multiplying by the reciprocal. The full process requires three steps: 1) approximation of the reciprocal, 2) correction by Newton iteration (see Chapter 5), and 3) multiplication by the reciprocal. Thus, division requires about three times the computational effort of multiplication!

HW1: Compute 12×9 in base 2 and verify.
HW2: Which of the following MATLAB assignment statements is preferred and why?
X = VALUE / 2.0; or X = 0.5*VALUE;

2.1 Machine Integers

Conceptually, a machine integer typically has the following binary form:

$$\pm b_n b_{n-1} b_{n-2} ... b_1 b_0 \text{ where } b_i \in \{0,1\} \tag{18}$$

Recall that the numbers b_i are called "bits," a contraction of the term "binary digits." The *integer word length* is the predetermined number of bits set aside in memory to store a single integer value. For example, the integer word length typical of PCs is 4 bytes = 32 bits. Exactly how those bits are used depends upon the particular machine. In the most simple-minded approach the leading (left-most) bit is used to indicate the sign (+ or -) of the integer and the remaining bits represent the absolute value of the number in binary notation. Thus on a PC, the 32-bit string might be arranged, for example, as follows:

bit string:	–	1	1	0	...	0	1
bit index:	1	2	3	4	...	31	32

In the 32 representation above, what is the largest integer that can be represented? It is fairly obvious that the largest integer would have a sign bit indicating $+$, followed by 31 "on" bits each with value 1. Therefore, the largest integer would be

$$\begin{aligned} N &= +(111...11)_2 \\ &= 1\times 2^{30}+1\times 2^{29}+1\times 2^{28}+...+1\times 2^1+1\times 2^0 \end{aligned} \tag{19}$$

We could evaluate N in decimal notation by Horner's method. However, there is a much simpler way. If we add 1 to N, the result will be a binary integer of 32 bits, the leading bit being one, and the remaining bits being zero. That is $N+1=+(1000...00)_2$. In decimal notation $N+1=2^{31}=2147483648$. Thus, the largest integer $N=2,147,483,647$, a tad more than two billion. Similarly, the most negative integer is $-N=-2,147,483,647$.

ASIDE: If truth be told, the schema described here has two drawbacks: 1) The procedure for adding two such integers is unnecessarily complicated, and 2) representation of zero is non unique; that is, $\pm(000...00)_2=$. In practice, most machines make use of *two's complements*, a notion slightly beyond the scope of this book. For interested readers, however, look for example at *Discrete Mathematics with Applications* (2nd Ed.) by Susanna Epp, Brooks/Cole (1995), in which the following definition is given:

DEF: Given a positive integer a, the *two's complement* of a relative to a fixed bit length n is the n-bit binary representation of 2^n-a.

Curiously, given a and n, 2^n-a can be found in three easy steps:

1. Write the n-bit representation of a.
2. Switch all the 1's to 0's and vice versa.
3. Add 1 to the result in Step 2 in binary notation.

In two's-complement form, nonnegative integers are treated conventionally, but negative integers are represented as the two's complement of their absolute values. As a result, for a 32-bit integer word length, the integers $-2,147,483,648\leq N\leq 2,147,483,647$, are each uniquely represented, *including zero*. But more importantly, a single algorithm suffices for adding any two integers regardless of their signs. Here is a summary of the algorithm for adding two 32-bit integers:

1. Convert both integers from decimal to binary representations (representing negative integers as the two's complement of their absolute values).
2. Add the resulting binary integers using ordinary binary addition.
3. Truncate any leading 1 (overflow) that occurs in the 2^n position.
4. Convert the result back to decimal notation by interpreting any 32-bit integers with leading 0's as positive and those with leading 1's as negative.

2.1.1 Word-Length Options

Although the largest 32-bit integer, $N = 2,147,483,647$, exceeds two billion, this number may not be sufficiently large for some applications. For example, to print out the value of each base pair in the human genome would require a loop from 1 to approximately 3 billion. Most compilers allow the user to access short or long integers, depending upon the needs of the algorithm. In C, for example, one uses the type-declaration keywords **short** or **long** to change the default length of an integer variable. On the other hand, in MATLAB, integer word length is modified during type declaration , which follows the word INTEGER in the type-declaration syntax as follows:

```
I=ones(1,1,'int8')    % 8-bit (1-byte) integer
J=ones(1,1,'int16')   % 16-bit (2-byte) integer
K=ones(1,1,'int32')   % 32-bit (4-byte) integer
L=ones(1,1,'int64')   % 64-bit (8-byte) integer
```

HW3: What is the largest integer of any KIND that can be represented in MATLAB?

Note: For some reason, int64 is not fully supported in MATLAB.

2.2 Floating-Point Numbers

Computers store real numbers in *floating-point notation*. To begin to get a handle on this concept consider the transcendental number $\pi/4 = 0.7853981634...$ and a 1-foot ruler. Is there a length on the ruler that corresponds to $\pi/4$ feet? Is there a tic mark on the ruler that corresponds to $\pi/4$ feet? If you answered both questions correctly, you are well on your way to understanding how computers store real numbers in *floating-point notation*.

There is certainly a length on the ruler corresponding to $\pi/4$ feet, but it has no tic mark. The closest tic is at 9 7/16 inches = 0.786458333... feet.

REMARKS: Floating-point numbers are like the tics on a ruler; there is not a tic for every real number, and so a given real number is represented by the closest number that has a corresponding tic.

Another way of looking at floating-point notation is that it is the binary equivalent of scientific notation. Here are some familiar examples of scientific notation:

$$\text{velocity of light} \Rightarrow 3 \times 10^5 \text{ km/s}$$
$$\text{250 billion dollar deficit} \Rightarrow -2.5 \times 10^9$$
$$\text{Avogadro's number} \Rightarrow 6.022 \times 10^{23}$$
$$\text{Planck's constant} \Rightarrow 6.626068 \times 10^{-34} \text{ m}^2 \text{ kg / s}$$

In general, scientific notation, which is particularly useful for expressing very large or very small numbers, consists of a decimal part, usually between one and ten and called the *mantissa*, and an exponential

part. Floating-point notation is similar, but with stricter format.

Suppose $x \in \mathcal{R}$ (a real number). A computer will represent x in binary as "float x" as follows:

$$fl(x) = \pm .b_1 b_2 b_3 ... b_n \times 2^m \tag{20}$$

The three components of a floating-point number are the sign bit, an n-bit string representing the mantissa, and an exponent m also expressed in binary.

EX1: Express 0.0_{10} in floating-point notation. $0.0_{10} = +.000...0 \times 2^0$

EX2: Express 0.5_{10} in floating-point notation. $0.5_{10} = +.100...0 \times 2^0$

EX3: Express 1.0_{10} in floating-point notation. $1.0_{10} = +.100...0 \times 2^1$

Without an additional constraint on format, floating-point numbers are non-unique. For example, we could have expressed $0.5_{10} = +.010...0 \times 2^2$. To remove the problem of non-uniqueness, it is customary to represent non-zero floating-point numbers in *normalized* form; that is, with a 1 in the leading bit of the mantissa. Thus, the only floating-point number whose leading bit is zero is zero itself.

In general, the word length of a floating point number, and the partition of those bits into mantissa and exponent, depends upon the particular computer. For the sake of illustration, let's consider a lame computer with a 4-bit mantissa and whose exponents can run from $m = -3$ to $m = 4$. Let's now write down *all* the real numbers this lame-frame can represent exactly. Let's first set $m = 0$ and consider all possible normalized 4-bit mantissas, of which there are only 9. The decimal equivalent of these machine values are given in the center of Table 4.

Mantissa	$m=-3$	$m=-2$	$m=-1$	$m=0$	$m=1$	$m=2$	$m=3$	$m=4$
$(0.0000)_2$	0	0	0	0	0	0	0	0
$(0.1000)_2$	0.0625	0.125	0.25	0.5	1	2	4	8
$(0.1001)_2$	0.0703125	0.140625	0.28125	0.5625	1.125	2.25	4.5	9
$(0.1010)_2$	0.078125	0.15625	0.3125	0.625	1.25	2.5	5	10
$(0.1011)_2$	0.0859375	0.171875	0.34375	0.6875	1.375	2.75	5.5	11
$(0.1100)_2$	0.09375	0.1875	0.375	0.75	1.5	3.0	6	12
$(0.1101)_2$	0.1015625	0.203125	0.40625	0.8125	1.625	3.25	6.5	13
$(0.1110)_2$	0.109375	0.21875	0.4375	0.875	1.75	3.5	7	14
$(0.1111)_2$	0.1171875	0.234375	0.46875	0.9375	1.875	3.75	7.5	15

Table 4: Complete list of machine reals for 4-bit floating-point mantissa with exponent range $-3 \leq m \leq 4$.

We can now fill in the table by considering non-zero exponents. For example the values in the column just to the right of the center column were obtained by multiplying those in the center column by $2^1 = 2$, and so on.

Now consider $x = \pi$. What is $fl(x)$ on our lame-frame computer? It is the closest machine number to the actual value of x. In this case $fl(x) = (3.25)_{10} = +(.1101)_2 \times 2^2$. Clearly this is a terrible representation of π, but it is the best we can do with a 4-bit mantissa. Fortunately, the situation improves markedly as

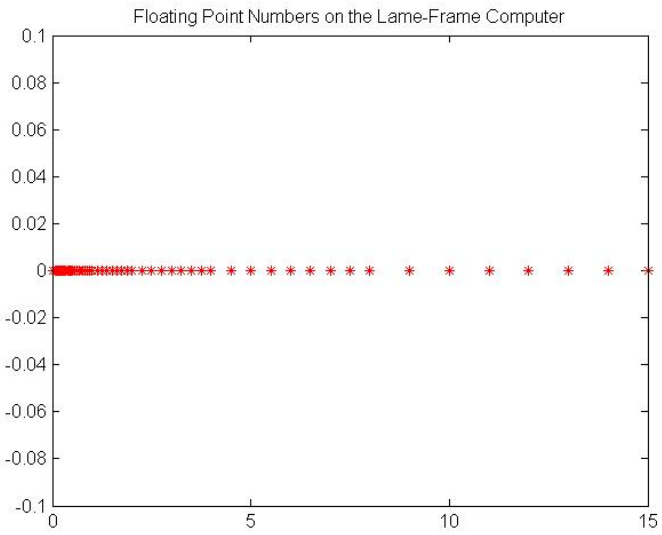


Figure 1: Pictorial representation of the numbers represented by the Lame-Frame computer.

the number of bits in the mantissa increase significantly. Figure 2(a) compares *floating-point word length* on a PC with that of a supercomputer.

To summarize, the mantissa of a floating-point number determines the equivalent number of significant (decimal) digits and the exponent controls the placement of the "decimal" point so that the representation is normalized. Let's make the notion of *significant digits* more precise. The following two tables summarize some of the results of the past two sections.

Integer storage for d-digit word-length

	base-2	base-β
largest number	$2^{d-1}-1$	$\beta^{d-1}-1$
smallest number	-2^{d-1}	$-\beta^{d-1}$
total numbers represented	2^d	$2\cdot\beta^{d-1}$

Floating point storage for n-bit mantissa and m-bit exponent for base-2 machine:

exponent range	$-(2^{m-1})$ to $2^{m-1}-1$
largest positive number	$2^{(2^{m-1}-1)}(2^{-1}+\ldots+2^{-n})$
smallest positive number	$.1\times 2^{-2^{m-1}}$
total numbers represented	$2^{n+m}+1$

2.3 Precision

When we say *significant bits* we are referring to the binary representation of a real number, and when we say *significant digits* we are referring to its decimal representation. How are significant bits and significant digits related?

DEF: *Precision* is the approximate number of decimal digits represented by the mantissa of a floating-point number. Although most modern computers are 64-bit with 53-bit mantissas, for the purpose of the following discussion we are going to examine single precision PCs. For example, single-precision PC's have 23-bit mantissas. In single precision, $2^{23} - 1 = 8388607 \approx 8 \times 10^6$ is the largest number that can be represented by the mantissa. (Here, the position of the decimal point is irrelevant, because the number of significant digits does not change if the decimal point is shifted.) This is a 7-digit number, so it appears that a 23-bit mantissa gives 7-digit precision. Not quite. As not all 7-digit numbers can be represented by the mantissa (for example, 9,000,000), we say the precision is 6-7 digits. To summarize, if n is the number of bits in the mantissa, we say a computer has p-digit precision iff

$$2^n \approx 10^p \tag{21}$$

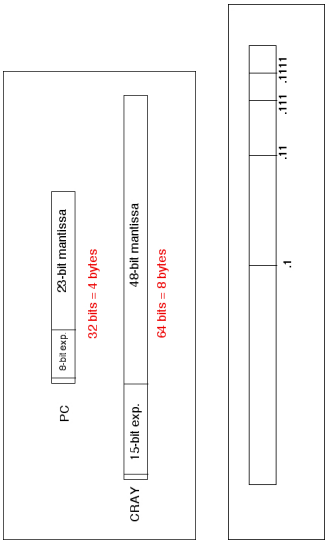
HW1: Find the precision of a Cray supercomputer.

It may come as an unwelcome shock that, whereas most scientific calculators have 10-12 digits of precision, the average single-precision PC has only 6-digit precision. What can be done to remedy this undesirable situation? The default declaration for MATLAB is double precision, i.e. 64-bits. On many supercomputers 64-bits are assigned for *single-precision* in which case *double-precision* reals have a whopping 128 bits. It should *not* be assumed, and in fact is wrong, that invoking "double precision" effectively doubles the number of bits in the mantissa. Exactly how the bits of double-precision words are partitioned between the mantissa and the exponent varies by machine.

There is another way to look at the notion of precision. Let's place the binary numbers $(.1)_2$, $(.11)_2$, $(.111)_2$, etc., which can be represented exactly, as tic marks on a ruler, as shown in Fig. 2(b). Note that each additional bit yields a new tic halfway between the previous one and the end of the ruler. Suppose then that there are n bits in the mantissa. How far apart are the tic marks? The distance between two adjacent tic marks is $2^{-n} \approx 10^{-p}$. For a 23-bit mantissa, $2^{-23} = 0.119209 \times 10^{-6} \approx 10^{-7}$, so, once again, the precision p is almost 7 digits, but not quite.

2.4 Underflow and Overflow

A run-time *underflow (overflow) error* occurs whenever the exponent of a floating-point number is too large (small) to be represented. To assess whether or not a calculation is in danger of an overflow error, we need to know what the largest exponent p can be in floating-point notation. On a PC, there are 8 bits assigned to the exponent. Suppose all 8 bits are "on," in which case $p_{\max} = (1111111)_2 = 255_{10}$. Zero is also a possibility, so the exponents could range from 0 to 255. But wait. This won't work, because we need negative exponents also. Therefore, normally, the exponents are split more-or-less evenly between negative and positive. With 256 possibilities, and equal opportunity for negative and positive, $-128 \leq p \leq 127$.



(a) Single precision floating-point numbers on PC vs. supercomputer.

(b) Selected machine mantissas as tic marks on a ruler.

Figure 2:

Finally, $2^{128} \approx 10^{38}$, so the largest real number that can be represented in floating-point notation is approximately 1.0×10^{38} in scientific notation. Similarly, the smallest positive real number that can be represented is roughly 1.0×10^{-38}.

EX: Consider the following lines of MATLAB and explain what happens when the variable Z is computed.

X = ones(1,1,'single')*3.7E20, Y = ones(1,1,'single')*4.2E28, Z=ones(1,1,'single')
Z = X*Y;

In case you are wondering, 3.7E20 is MATLAB's syntax for 3.7×10^{20} in scientific notation. The "E" means exponent. The product of X and Y has an exponent of at least 20+28=48. This number exceeds the maximum exponent size of 38. The computation will result in Z not being correct. How could you change the above commands so that the computation would work correctly?

2.5 Exercises

1. The following parts involve operations in base-8 (octal).

   (a) Compute $(36)_8 + (42)_8$ in base-8 arithmetic.

   (b) Complete the following multiplication table for base-8:

$\times$	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2						
3	0	3						
4	0	4						
5	0	5						
6	0	6						
7	0	7						

(c) Use part(b) to compute $(217)_8 \times (16)_8$ in base-8 arithmetic.

2. Suppose you are employed to design a digital watch. How many bits will you need to store the time? Justify your answer.

3. Suppose $x = (13)_{10}$. What is $\text{fl}(x)$ on a binary computer with a 3-bit mantissa and 3-bit exponent?

4. IEEE specifications for double precision variables use one bit for the sign, 11 bits for the exponent, and 52 bits for the mantissa. What precision does it have?

5. Find the precision of real numbers on a Cray supercomputer with a 64-bit word, of which 15 bits are reserved for the exponent.

6. Suppose that on a base-2 machine, the distance between 7 and the next largest floating point number is 2^{-12}. What is the distance between 70 and the next largest floating point number?

7. You recently invented a way to store 9 bits of information in an 8 bit unit (the patent is pending).

   (a) What is the term for 8 bits of information?

   (b) What is the largest and smallest integer that can be stored in 9 bits with 1 bit for the sign?

   (c) Consider using the following to store floating point numbers

   $$(-1)^{b_1} \times (0.b_6b_7b_8b_9) \times 2^{b_2b_3b_4b_5-7},$$

   where b_i represents the ith bit. What is the precision? What is the largest value that can be stored?

8. How does a single precision type represent the number -285.75?

9. Is $(0.111)_{10}$ a machine number in most machines? Why or why not?

10. Explain how a computer subtracts two binary numbers. After your explanation, give an example.

11. Consider a machine whose storage consists of numbers in base-3 stored as

    $$(0.b_1b_2b_3) \times 3^k$$

    and also includes a sign bit. Suppose that $-3 \leq k \leq 4$. What is the largest floating point number this machine can represent exactly?

3 Part II: Error and Its Sources

If the result of a digital computer is almost never exactly correct because of the limitations of storing real numbers on a finite bit machine, then the question "How much in error is my result?" is an important one. Throughout the remainder of this course, error and its estimation will be a recurring theme.

DEF: *Absolute error* is the difference between the exact value and its approximation. Thus, letting x and $\bar{x}$ represent the exact and approximate values of x, respectively, and δ represents absolute error, then

$$\delta = x - \bar{x} \tag{22}$$

Absolute error, therefore, is more-or-less meaningless. Only relative error really matters.

DEF: *Relative error* is absolute error divided by the true value. If ε represents relative error, then

$$\varepsilon = \frac{x - \bar{x}}{x} = \frac{\delta}{x} \tag{23}$$

It should be pointed out that, whereas absolute error is a *dimensional* quantity with units such as miles, grams, miles/hour, hours, etc. relative error, being a ratio of values, is always *dimensionless* (unit-less).

Error sneaks into a calculation from many sources. Among these are initial errors in the input data, which leads to the old adage of computing: "Garbage in, garbage out." A more subtle source of error is model error. For machine solution of physical problems, the equations to be solved are often *models* of the physical phenomenon. Certain assumptions may have been necessary to derive the models, in which case the model may not be a complete or accurate description of the physical problem. Model errors are beyond the scope of this course, but you should at least be aware of this source of error. The two sources of error that most concern us are *round-off error* and *truncation error*. Each represents the limitation of a finite process in describing an infinite one. In this Chapter, we address the first of these sources of error: round-off error.

3.1 Round-Off Error Defined

Round-off error is the price one pays for digital computing.

DEF: *Round-off error* is that error incurred when real numbers are represented approximately by floating-point numbers because of finite word length. If $\bar{x} = \mathrm{fl}(x)$, then

$$\delta = x - \mathrm{fl}(x) \tag{24}$$

is the *absolute round-off error*, and

$$\varepsilon = \frac{x - \mathrm{fl}(x)}{x} \tag{25}$$

is the *relative round-off error* associated with the storage of x in memory.

EX: Find the exact absolute and relative round-off errors incurred when the decimal fraction 1/10 is approximated by a floating-point number with a 10-bit mantissa.

First, we need to represent $x = (0.1)_{10}$ in binary. From a previous HW exercise, recall that $0.1_{10} = (0.0001100110\overline{0011})_2$. Recall that, although $(0.1)_{10}$ is a terminating decimal fraction, it is a repeating binary fraction, which could be represented exactly only by an infinity of bits. Truncating that infinity of bits to a finite (in this example, 10-bit) length will result in significant round-off error. In normalized form,

$$x = .1100110\overline{01100}... \times 2^{-3} \tag{26}$$

Before we find the round-off error, let's work backwards to see if we can verify that the repeating binary fraction above is indeed the same as the decimal fraction 1/10.

ASIDE: Geometric Series (Review from Math 236)

DEF: The series $\sum_{n=0}^{\infty} ar^n = a + ar + ar^2 + ar^3 + ...$ is called a *geometric series*.

THEOREM: The geometric series above with $a \neq 0$ converges to the sum $S = \frac{a}{1-r}$ provided $|r| < 1$, and it diverges for $|r| \geq 1$.

PROOF: Consider the sequence of partial sums, whose first $n+1$ elements are given by

$$\begin{aligned} S_0 &= a \\ S_1 &= a + ar \\ S_2 &= a + ar + ar^2 \\ &... \\ S_n &= a + ar + ar^2 + ... + ar^n \end{aligned}$$

Multiplying the nth partial sum S_n by r gives

$$rS_n = ar + ar^2 + ar^3 + ... + ar^{n+1} \tag{27}$$

Note that the sums S_n and rS_n agree in all terms but for the first and last. From this, it follows that

$$S_n - rS_n = a - ar^{n+1} \tag{28}$$

Finally, solving Eq. 28 for S_n yields

$$S_n = \frac{a - ar^{n+1}}{1 - r} \tag{29}$$

Finally, the sum $S = \lim_{n\to\infty} S_n$. Provided that $|r| < 1$, $\lim_{n\to\infty} |r|^n = 0$, in which case the conclusion of the theorem follows from the limit of Eq. 29.

Why detour from the topic of round-off error to re-visit geometric series? Because repeating "decimal" fractions, whether in base 10 or binary notation, are simply geometric series in disguise. For example the first (left-most) block of 4 bits in Eq. 26, namely $(.1100)_2$, is the equivalent of the decimal fraction 3/4.

The second block of 4 bits, $(.00001100)_2$, has the decimal equivalent of $3/4 \times 2^{-4}$, because the "decimal" point is shifted four spaces to the left. Similarly $(.000000001100)_2 = 3/4 \times 2^{-8}$, etc. Thus the entire repeating binary fraction represented by the *mantissa* is the geometric series

$$3/4 + 3/4 \times 2^{-4} + 3/4 \times (2^{-4})^2 + 3/4 \times (2^{-4})^3 + ... \tag{30}$$

Here $a = 3/4$ and $r = 2^{-4} = 1/16$; hence $S = \frac{3/4}{1-1/16} = \frac{3/4}{15/16} = 4/5 = 0.8_{10}$. However, the exponential term of Eq. 26 is $2^{-3} = 1/8$. Thus, $0.8 \times 1/8 = 1/10$, and hence, we have successfully recovered the decimal fraction 1/10 from its repeating binary equivalent.

We are now poised to determine the absolute and relative round-off errors associated with storing 1/10 on our machine with 10-bit mantissa, for which

$$\text{fl}(x) = .1100110011 \times 2^{-3} \tag{31}$$

From Eqs. 26 and 31, we obtain

$$\begin{aligned} \delta &= x - \text{fl}(x) = (.11001100\overline{1100} - .1100110011) \times 2^{-3} \\ &= (.0000000000001\overline{1100}) \times 2^{-3} \\ &= (.\overline{1100}) \times 2^{-15} \\ &= (.8)_{10} \times 2^{-15} \end{aligned}$$

The relative round-off error is found by dividing δ above by the exact value $x = 0.1$. Thus,

$$\varepsilon = \frac{\delta}{x} = \frac{.8 \times 2^{-15}}{.1} = 2^{-12} \tag{32}$$

Finally, in decimal notation $\varepsilon \approx 0.00024$ or about 0.024 percent error. While this error is not terrible, imagine what happens when floating-point numbers that are in error are used again and again in lengthy calculations. In such cases round-off error tends to grow like a snowball.

A fundamental question to ask is the following: *Given a machine whose real numbers are represented by an n-bit mantissa, what is the absolute value of the worst possible relative round-off error that can be incurred?* This little number is so important that it is given a special name and symbol: *unit round-off error*, u.

3.2 Unit Round-Off Error

To answer this question, recall Fig. 2(b) in Chapter 2, which compares machine numbers to the tic marks on a ruler. Let's ignore the exponent of a floating-point number and just consider its mantissa. Further consider the finite sequence of mantissas given by the binary numbers $\{(.1), (.11), (.111), ..., (.111...1)\}$, where the last number has exactly n non-zero bits. Further let's assign a tic mark on the ruler to each of these binary numbers. Once again, note that each additional bit yields a new tic halfway between the previous one and the end of the ruler. How far apart are the fine-grained tic marks? We have shown previously that the distance between two adjacent tic marks is 2^{-n}. The relevant question now is: What is

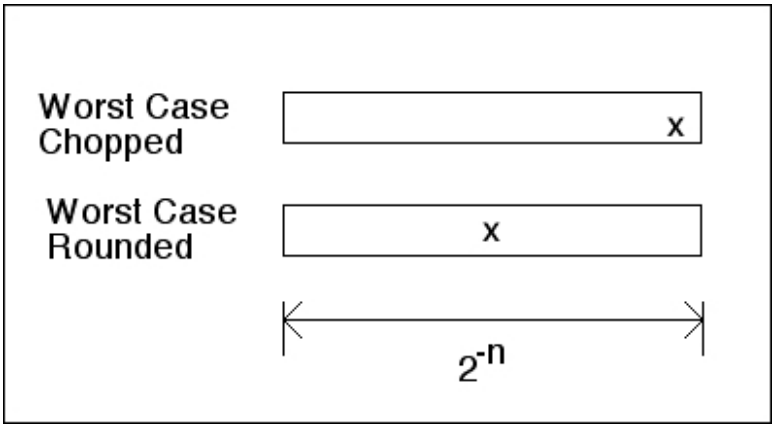


Figure 3: Chopping vs. rounding.

the largest possible relative error that can be incurred when x is represented by $\mathrm{fl}(x)$? First, we must find the largest absolute error. This depends upon whether the machine *chops* or *rounds*. A floating-point number $\mathrm{fl}(x)$ is said to be rounded if $\mathrm{fl}(x)$ is the closest machine number to x. One the other hand, $\mathrm{fl}(x)$ is chopped if all bits of the exact x beyond n are dropped when x is approximated. This is best illustrated graphically. Figure 3 zooms in on the smallest interval on our ruler in which x is contained. The worst case scenario for chopping is when x falls just shy of the next available machine number, in which case $|\delta| \leq 2^{-n}$. The worst case scenario for rounding occurs when x falls midway between two available machine numbers, in which case $|\delta| \leq \frac{1}{2}2^{-n} = 2^{-(n+1)}$. The two situations are summarized by the following inequality:

$$|\delta| \leq \begin{cases} 2^{-(n+1)} & \text{(rounded)} \\ 2^{-n} & \text{(chopped)} \end{cases} \tag{33}$$

Our real interest is relative error though, not absolute error. The largest (absolute) relative error $|\delta|/x$ occurs when $|\delta|$ is at its maximum and x is simultaneously at its minimum. The smallest non-zero mantissa in normalized form is $(0.1)_2 = (1/2)_{10}$. Thus $|\varepsilon| \leq \frac{|\delta|}{1/2}$. Here then is the final result:

$$|\varepsilon| \leq u \equiv \begin{cases} 2^{-n} & \text{(rounded)} \\ 2^{-n+1} & \text{(chopped)} \end{cases} \tag{34}$$

The unit round-off error u is a measure of the innate precision of a computer, and because of that, it is of fundamental importance in scientific computing. There are two ways to discuss this concept of machine epsilon. We have already discussed the largest relative error possible in $fl(x)$, but an equivalent method is to examine the distance from 1 to the next floating point number.

You might be wondering why we did not consider zero as the smallest possible mantissa in the derivation above. The reason is that zero can be represented exactly; hence there is no round-off error incurred in storing zero. Another point of possible confusion is that the generic term *round-off error* applies both to chopping and rounding, not specifically to rounding.

Finally, what if one does not know the length of the mantissa of the computer that one is using. Is there any way to determine u in advance? Yes, a very clever way: by the unit round-off error algorithm. Here is the algorithm in pseudocode:

ALGORITHM 3.1: *Machine Unit Round-Off Error*

$u \leftarrow 1$ *(initialize temporary storage register)*

repeat until $(1.0 + u = 1.0)$
| $u \leftarrow u/2.0$
|—

$u \leftarrow 2.0 \cdot u$
output u

This is admittedly a strange algorithm that violates most of the rules you have learned. For starters, there is NO input! How can that be? In essence the input is the machine itself. The algorithm is asking the machine about itself! Secondly, the exit condition is bizarre. How can $1 + u = 1$ unless $u = 0$? This would be the case in infinite precision, but in finite precision, eventually u, which is divided by 2 after each iteration, becomes so small that it fails to register relative to 1.0 even though it is non-zero. Finally, you were told never to use equality for an exit condition if REAL numbers are involved. Here, we have intentionally violated that convention, for a very good purpose.

How does the algorithm work? Normally floating-point numbers are stored in normalized form, with 1's in the leading bit. However, two floating-point numbers can be added only when they have the same exponent. Hence, the smaller of the two numbers is placed in non-normalized form so that its exponent matches that of the larger number. For example, consider adding (1/2 + 1/4) in binary on with a 3-bit mantissa. In normalized form

$$\begin{aligned} 1/2 &= (.100)_2 \times 2^0 \\ 1/4 &= (.100)_2 \times 2^{-1} \end{aligned}$$

At the time of addition, the binary value of 1/4 is adjusted as follows:

$$\begin{array}{rl} & (.100)_2 \times 2^0 \\ + & \underline{(.010)_2 \times 2^0} \\ & (.110)_2 \times 2^0 \end{array}$$

Let's play the same game again, but this time add 1/16 to 1/2 on our hypothetical 3-bit machine. Note that $1/16 = (.100)_2 \times 10^{-3}$ is a perfectly acceptable number on a 3-bit machine. However, here's what happens during the add:

$$\begin{array}{rl} & (.100)_2 \times 2^0 \\ + & \underline{(.0001)_2 \times 2^0} \\ & (.100)_2 \times 2^0 \end{array}$$

If the machine does not round up, then the 1 bit in the 4th place "falls off" the end of the bit string in the result and fails to register in the sum, which is restricted to 3 bits only. Similarly, the loop in ALG 3.1 continues to halve the second number in the sum until its leading bit falls off the end of the mantissa,

at which point the computer has revealed the number of bits n it allots to the mantissas of floating-point numbers, and by inference, it has revealed u. ALG 3.1 returns the correct u regardless of whether the machine chops or rounds. It does not reveal, however, which method is used.

Finally, most compilers provide an intrinsic function that returns the value of u, which is also sometimes called "machine epsilon." Accordingly, MATLAB's intrinsic function for this purpose is *eps*.

3.3 Propagation of Round-Off Error

The previous section dealt with the error incurred in approximating a real number as a floating-point number. What happens, however, when these machine numbers, each of which contains error, are used time and again in a calculation? Although the initial errors were small, it is not inconceivable that the error accumulates in a "snowball effect." If so, can we quantify and ultimately bound the process of the propagation of round-off error?

Let's start by a simple example: the addition of two machine numbers on a hypothetical machine with an 8-bit mantissa. For simplicity, let's assume chopping rather than rounding. Let $x = 1/3$, $y = 1/5$, and $z = x + y$, the exact result of which is $z = 8/15 = 0.533\overline{3}$. What happens on our hypothetical computer is the following:

$$\begin{aligned} \text{fl}(\frac{1}{3}) &= (.10101010)_2 \times 2^{-1} \\ +\text{fl}(\frac{1}{5}) &= (.11001100)_2 \times 2^{-2} \text{ (normalized)} \\ &= \underline{(.01100110)_2 \times 2^{-1}} \text{ (aligned)} \\ &\quad (1.00010000)_2 \times 2^{-1} \\ &\quad (.10001000)_2 \times 2^{0} \text{ (chopped \& normalized)} \end{aligned} \tag{35}$$

The final result, $\text{fl}(z)$ is $(0.53125)_{10}$. Thus the relative round-off error following the addition operation is $(0.533\overline{3} - 0.53125)/0.53125$, or 0.39 percent. A re-examination of the addition operation reveals that there are *three* possible entry points for error: 1) the approximation of x by $\text{fl}(x)$, the approximation of y by $\text{fl}(y)$, and the approximation of the sum by $\text{fl}(z)$. We now examine a generic addition operation to see if we can bound the error of the result.

Exact:	Machine:
$z \leftarrow x + y$	$\bar{x} \leftarrow \text{fl}(x)$
	$\bar{y} \leftarrow \text{fl}(y)$
	$s \leftarrow \bar{x} + \bar{y}$
	$\bar{z} \leftarrow \text{fl}(s)$

Our goal is to bound the relative error of the final result, namely $(z - \bar{z})/z$. To simplify the logic, let's assume $x > 0$ and $y > 0$.

$$\delta_x = x - \bar{x} \Rightarrow \bar{x} = x - \delta_x \tag{36}$$

$$\delta_y = y - \bar{y} \Rightarrow \bar{y} = y - \delta_y$$

From Eq. 36 the definition of relative error it follows that

$$\begin{aligned} \varepsilon_x &= \frac{\delta_x}{x} \Rightarrow \bar{x} = x(1-\varepsilon_x) \\ \varepsilon_y &= \frac{\delta_y}{y} \Rightarrow \bar{y} = y(1-\varepsilon_y) \end{aligned} \tag{37}$$

Thus,

$$\begin{aligned} s &= \bar{x} + \bar{y} \\ &= x(1-\varepsilon_x) + y(1-\varepsilon_y) \\ &= x + y - x\varepsilon_x - y\varepsilon_y \\ &= z - x\varepsilon_x - y\varepsilon_y \end{aligned} \tag{38}$$

It follows that

$$\begin{aligned} \bar{s} &= s(1-\varepsilon_z) \\ &= (z - x\varepsilon_x - y\varepsilon_y)(1-\varepsilon_z) \\ &= z - x\varepsilon_x - y\varepsilon_y - z\varepsilon_z + x\varepsilon_x\varepsilon_z + y\varepsilon_y\varepsilon_z \end{aligned} \tag{39}$$

where ε_z is the relative error from storing s as a float. The last two terms of the last line of Eq. 39 contain products of presumably small relative errors and can be safely neglected. Thus, an accurate approximation is

$$\bar{s} \approx z - x\varepsilon_x - y\varepsilon_y - z\varepsilon_z \tag{40}$$

The absolute error in z is now available.

$$\delta \equiv z - \bar{s} = x\varepsilon_x + y\varepsilon_y + z\varepsilon_z \tag{41}$$

By the Triangle inequality and the fact that x and y are presumed positive, it follows

$$\begin{aligned} |\delta| &\leq x|\varepsilon_x| + y|\varepsilon_y| + z|\varepsilon_z| \\ &\leq xu + yu + zu = 2zu \end{aligned} \tag{42}$$

Dividing through by the exact value z above gives the sought-after result:

$$|\varepsilon| \equiv \frac{|\delta|}{z} \leq 2u \;\text{ (machine addition)} \tag{43}$$

This took some doing, but the result is simple to interpret: the absolute value of the relative error incurred by adding two machine numbers is at most twice the unit round-off error.

Were we to perform a similar analysis for the multiplication of two machine numbers, namely $z \leftarrow x \cdot y$ (which we will not do), the result would be

$$|\varepsilon| \leq 3u \;\text{ (machine multiplication)} \tag{44}$$

Not too surprisingly, the result for the division of two machine numbers ($z \leftarrow x/y$), multiplication by the inverse, is identical to that for multiplication, namely

$$|\varepsilon| \leq 3u \quad \text{(machine division)} \tag{45}$$

So, the relative error of either a multiplication or a division operation is bounded by three times the unit round-off error. All of this goes to show that adding, multiplying, or dividing machine numbers *may* (because these are upper bounds only) increase the relative round-off error of the result, but the rate of growth is relatively slow, doubling or tripling with each operation. However, over thousands or millions of operations, round-off error can become a significant problem, eroding the accuracy of a machine result. It may be necessary to use double-precision arithmetic in such cases.

It might be tempting to assume that substraction behaves as addition. Unfortunately, the situation is far, far worse than you might imagine. Suppose we want to bound the relative error for $z \leftarrow x - y$. The analysis is similar to that for addition until near the end, from which it is obtained that

$$\delta \equiv z - \bar{s} = x\varepsilon_x - y\varepsilon_y + z\varepsilon_z \tag{46}$$

However, trouble arises when the triangle inequality is implemented because of the negative sign in front of y, in which case $|-y| = y$ and

$$\begin{aligned} |\delta| &\leq x|\varepsilon_x| + y|\varepsilon_y| + |z||\varepsilon_z| && (47)\\ &\leq xu + yu + |z|u && (48) \end{aligned}$$

From this comes the troubling result:

$$|\varepsilon| \equiv \frac{|\delta|}{|z|} \leq u\left[1 + \frac{x+y}{|x-y|}\right] \quad \text{(machine subtraction)} \tag{49}$$

When x and y are nearly the same, the term in the denominator above approaches zero, and the round-off error is multiplied dramatically. For example, if two machine numbers, each with 7-digit significance, agree in 6 of their first seven significant digits, the result of subracting them will retain only *one* significant digit. In a single operation, the relative error has been multiplied one million fold!!!

3.4 Catastrophic Loss of Significance

It is sobering to learn that a *single* operation in a long program of hundreds or thousands of lines can completely destroy the accuracy of the result. That, in fact, can happen, and when it does, it is called *catastrophic loss of significance*. Here is a very real example.

Consider the quadratic formula, which solves $ax^2 + bx + c = 0$, namely

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{50}$$

Suppose $b > 0$ and $4ac$ is very, very small relative to b. In this case the numerator of Eq. 50 looks almost like $-b \pm b$. This is not a problem for the root $-b - b$, but it is a problem for the root $-b + b$, because

this root is computed by the subtraction of two numbers that are almost identical. Let's put some numbers in to see what happens. Let $a = 1.1$, $b = 1.11$, and $c = .000000001$. Equation 50 gives the solutions $x_1 = -1.00909$ and $x_2 = 0.0$. The first root is correct to 6 places and the second root is correct *to no significant digits*!

It is a little-known fact that there is an alternative quadratic formula, given below, which is derived from the first by rationalizing its numerator.

$$x = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}} \tag{51}$$

In the second form, the root that required subtraction in the first formula is found by *addition* to be $x_2 = -9.00901 \times 10^{-10}$. This result is correct to six significant digits. The original result was in error by 100 percent!

The moral to this story is not to never use subtraction. It is to avoid the subtraction of numbers that are nearly the same. A general rule of thumb is that the round-off error grows by a factor of 10^n, where n is the number of leading digits of agreement. And whenever choices are available, choose the algorithm that is most benign in terms of error propagation.

3.5 Exercises

1. The expression $\sum_{i=1}^{\infty} \frac{1}{i}$ is the well known divergent series. What do you think would happen if you tried to sum it on a computer? Don't actually try, it would take far too long!

2. Find the absolute and relative round-off errors incurred when $\left(\frac{2}{3}\right)_{10}$ is approximated by a floating point number with a 3-bit mantissa in base-2.

3. Consider a machine in base-2 with a 4-bit mantissa and 4-bit exponent. What is the machine epsilon for this machine?

4. Derive the alternative quadratic formula of Eq. 51.

5. You know from algebra that the roots of a quadratic polynomial can be found using the quadratic formula. There is an alternate quadratic formula which can be derived from the first. The standard and alternate formulas are below:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{52}$$

$$x = \frac{-2c}{b \mp \sqrt{b^2 - 4ac}} \tag{53}$$

   Complete the following parts.

(a) Find the roots of the equation

$$x^2 + 111.11x + 1.2121 = 0$$

by hand using formula (52) using base-10 and mantissa length 5. Show all work.

(b) Find the roots of the equation in part (a) by hand using formula (53) using base-10 and mantissa length 5. Show all work.

(c) The true roots of this equation, with 5 significant digits are -1.1110×10^2 and -1.0910×10^{-2}. Explain any differences in the roots calculated in parts (a) and (b), and the true roots. What causes these differences? Is either formula (52) or (53) better than the other? Why or why not?

6. One way of calculating the inverse hyperbolic cosine function is to rewrite it as $\cosh^{-1}(x) = -\ln\left(x - \sqrt{x^2-1}\right)$. Explain why this is a bad idea for large values of x, then rewrite it in a form that is worth using for large x.

7. (a) Evaluate the polynomial $y = x^3 - 7x^2 + 8x - .35$ at x=1.37. Use 3-digit arithmetic with chopping. Evaluate the percent relative error.

   (b) Repeat (a) but express y as $y = ((x-7)x+8)x - .35$. Evaluate the error and compare with (a).

   (c) Explain what is going on.

8. Consider a binary machine with a 4-bit mantissa and 8-bit exponent. Suppose $a = (3)_{10}$ and $b = (7.5)_{10}$.

   (a) Suppose $x = a + b$. Find the value of $\text{fl}(x)$ after the machine computes the addition (assume rounding). Represent your answer in base-10.

   (b) Compute the relative error associated with the addition in part (a).

9. Recall from calculus, that the mathematical constant, e, can be defined as

$$e = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n$$

This problem involves approximating the value of e using the formula above. To approximate this limit, I wrote an m-file that computes

$$\left(1 + \frac{1}{n}\right)^n$$

for successively larger values of n. I calculated the absolute and relative error associated with this approximation and $\text{fl}(e)$. Here are my results:

```
    n        e_approx     fl(e)     abs.error     rel.error
-------------------------------------------------------------
1.00e+01      2.5937      2.7183      0.1245      4.5815e-02
1.00e+02      2.7048      2.7183      0.0135      4.9546e-03
```

```
1.00e+03     2.7169     2.7183     0.0014     4.9954e-04
1.00e+04     2.7181     2.7183     0.0001     4.9995e-05
1.00e+05     2.7183     2.7183     0.0000     4.9999e-06
1.00e+06     2.7183     2.7183     0.0000     5.0008e-07
1.00e+07     2.7183     2.7183     0.0000     4.9416e-08
1.00e+08     2.7183     2.7183     0.0000     1.1077e-08
1.00e+09     2.7183     2.7183     0.0000     8.2240e-08
1.00e+10     2.7183     2.7183     0.0000     8.2690e-08
1.00e+11     2.7183     2.7183     0.0000     8.2735e-08
1.00e+12     2.7185     2.7183     0.0002     8.8905e-05
1.00e+13     2.7161     2.7183     0.0022     7.9896e-04
1.00e+14     2.7161     2.7183     0.0022     7.9896e-04
1.00e+15     3.0350     2.7183     0.3168     1.1653e-01
1.00e+16     1.0000     2.7183     1.7183     6.3212e-01
1.00e+17     1.0000     2.7183     1.7183     6.3212e-01
1.00e+18     1.0000     2.7183     1.7183     6.3212e-01
1.00e+19     1.0000     2.7183     1.7183     6.3212e-01
1.00e+20     1.0000     2.7183     1.7183     6.3212e-01
```

Explain what happens to both types of error as n increases. Explain why the error behaves as it does (you do *not* need to convert things to base-2 or do any arithmetic here, just explain in words). Why does `e_approx` approach 1?

4 Part II: Fixed-Point Iteration

Fixed-point iteration (FPI) is a very simple idea with widespread applications and deep implications. It is also fun, something that can be accomplished with a computer program that consists of little more than a single loop.

DEF: Let $g(x)$, the *iteration function*, be continuous on $[a,b]$, and let x_0 be the *initial condition*. An *iteration* of the form

$$x_{i+1} \leftarrow g(x_i) \quad i = 0,1,2,... \tag{54}$$

is called a *fixed-point iteration*. The *iterates* x_i define the infinite *sequence*

$$\begin{aligned} & x_0 \\ x_1 &= g(x_0) \\ x_2 &= g(x_1) = g[g(x_0)] = g^2(x_0) \\ x_3 &= g(x_2) = g\{g[g(x_0)]\} = g^3(x_0) \\ & ... \end{aligned} \tag{55}$$

DEF: The sequence of iterates $\{x_0,x_1,x_2,...\} = \{x_i\}_{i=0}^{\infty}$ that is generated by FPI with $g(x)$ is called the *orbit* of $g(x)$ with initial condition x_0 and is denoted $O[g(x),x_0]$.

EX: List the elements of the orbit $O[x^2,1/2]$. In this case $x_{i+1} \leftarrow x_i^2$. Thus the orbit is given by the sequence $\{1/2,1/4,1/16,1/256,...\}$. This sequence clearly *converges* to zero. If, however, the initial condition were $x_0 = -1$, the iterates would have converged quickly to 1. Finally, if the initial condition were $x_0 = 2$, the iterates would have diverged to ∞.

Of primary interest is the question: What types of orbits are possible? If the iteration function is *nonlinear* it turns out that a rich variety of orbits may be possible: convergent, divergent, periodic, and chaotic. We had better define what we mean by each of these terms.

DEF: A sequence $\{x_i\}_{i=0}^{\infty}$ is said to converge to limit L, written $\lim_{i\to\infty} x_i = L$, provided that for every $\varepsilon > 0$, there exists integer $N(\varepsilon)$ such that $|x_i - L| < \varepsilon$ whenever $i > N(\varepsilon)$.

A sequence is a function whose domain is the set of non-negative integers. It is therefore illustrative to plot a convergent sequence as a function. Keep in mind, that sequences are not continuous, so when plotted, we get a set of points, not a curve. Notice that, in Fig. 4, all elements after x_5 fall within the range $[L-\varepsilon, L+\varepsilon]$. That is, for the given ε, $L-\varepsilon \le x_i \le L+\varepsilon$ for $i > 5$. Another way to say the same thing is that $|x_i - L| < \varepsilon$ for $i > 5$. Thus for the given ε, $N = 5$. If an N can be found for *any* $\varepsilon > 0$, no matter how small, then it can be said that the sequence converges to L. Graphically, the sequence converges to L if, beyond some point, the iterates (elements) fall ever closer to the horizontal line labeled L.

DEF: A sequence that does not converge is said to *diverge*.

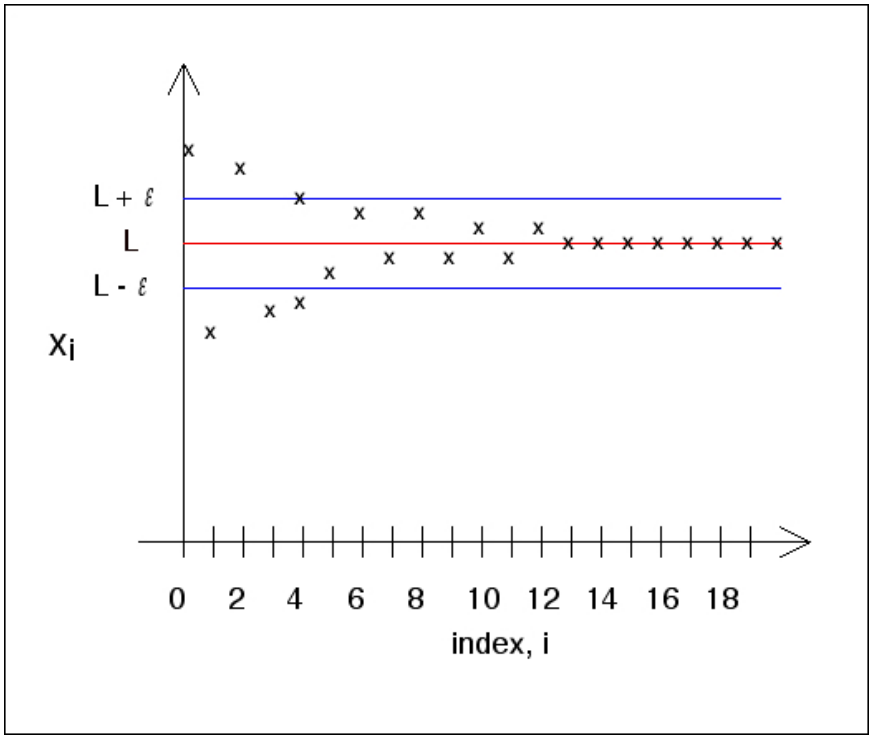


Figure 4: Elements of a convergence sequence.

4.1 Divergent Orbits

In Math 236, the focus is on convergence and theorems proving the convergence of certain types of sequences. Here our interest concerns sequences that are generated by FPI; henceforth we use the term "orbits." Divergent orbits may be in some sense more interesting than convergent ones, because of the variety of modes of divergence. Orbits may fail to converge in three entirely different ways: if their elements are periodic, if the elements grow unbounded, or if the elements are chaotic. Once again, we should define these terms precisely.

DEF: Let B be a positive real number. If for any $B > 0$, no matter how large, there exists $N(B)$ such that the iterates x_i of the FPI satisfy $|x_i| > B$ for all $i > N(B)$, then the orbit is said to be *unbounded*.

DEF: If the FPI $x_{i+1} \leftarrow g(x_i)$ does not converge, but $x_{i+1} \leftarrow g^n(x_i)$ converges, and n is the smallest positive integer for which this is true, then $O[g(x), x_0]$ is said to be *periodic* with period n.

EX: Consider $g(x) = -x$. Note that $O[g(x), 1] = \{1, -1, 1, -1, 1, -1, ...\}$, which is divergent. However $g^2(x) = g[g(x)] = -(-x) = x$, so that $O[g^2(x), 1]$ converges to 1. We say $g(x) = -x$ has a period-two orbit from the initial condition $x_0 = 1$.

Another way to identify periodic orbits of FPI's is by examining regular subsets of their associated sequences. For example, the sequence $\{1, -1, 1, -1, 1, -1, ...\}$ is associated with FPI using $g(x) = -x$ and $x_0 = 1$. From this divergent sequence, let's look at the subsequences formed by selecting even- and odd-indexed elements, respectively, namely the subsequnces $\{1, 1, 1, 1, ...\}$ and $\{-1, -1, -1, -1, ...\}$. Clearly,

each of these subsequences is convergent. In general, if the sequence S_k associated with a fixed-point iteration diverges, but the subsequence S_{nk} converges, and n is the smallest positive integer for which this is true, then the orbit of the FPI is periodic with period n.

We define a chaotic orbit by a process of elimination as follows:

DEF: A divergent orbit that is neither unbounded nor periodic is *chaotic*.

It is an interesting fact that even simple nonlinear iteration functions may exhibit the full range of convergent and divergent behaviors. For, example, one of the most oft-studied FPI's is called the *logistic map*. The iteration function for the logistic map is $g(x) = \alpha x(1-x)$, where $0 \leq \alpha \leq 4$ is a *parameter* that can be varied to produce different behaviors. Depending upon the value of α, the logistic map produces convergent, periodic, or chaotic orbits. However, the most remarkable attribute is that orbits of *any* period can be obtained if you know where to look in the parameter range!

HW1: Consider FPI with the logistic map, with initial condition $0 < x_0 < 1$. Write a simple computer program to input the values of α and x_0 and then compute and output the first 100 or so iterates of the logistic map, namely

$$x_{i+1} \leftarrow \alpha x_i(1-x_i) \tag{56}$$

Note that the "guts" of the program are comprised of a simple loop. a) Do a numerical experiment to find a value of α for which the iterates converge to 0.5 from any initial guess within the specified interval. b) Find a value of α for which the orbit is of period 2. c) Find a value of α for which the orbit is of period 8. d) Find a value of α for which the orbit appears chaotic.

4.2 Convergent Orbits

The $64,000 question is: Why do some FPI's converge and others diverge? More precisely, can we find criteria that guarantee convergence?

DEF: If there exists some value p such that $p = g(p)$, then p is called a *fixed-point* of $g(x)$.

EX: If $g(x) = x^2$, then solving $p = p^2$ defines the fixed points of the iteration $x_{i+1} \leftarrow g(x_i)$. Thus the fixed points are the roots $p \in \{0,1\}$ of $p^2 - p = p(p-1) = 0$.

Fixed points are appropriately named. If the initial condition x_0 corresponds to a fixed point, the iterates will never change. In the example above, the fixed points were found analytically, by solving $p = g(p)$ for p. Fixed points can also be found graphically by graphing line $y = x$ and the iteration function $y = g(x)$ on the same axes. These graphs intersect where $x = g(x)$; that is, at the fixed point(s). Figure 5 shows the fixed points for the iteration function of the logistic map, $g(x) = \alpha x(1-x)$ for several values of the parameter α. Notice that, although $p = 0$ is always a fixed point, the occurrence and location of a second fixed point depends upon the value of the parameter. There is only one fixed point ($p = 0$) if $0 \leq \alpha < 1$. For $1 \leq \alpha$, there are two, and the value of the second increases as the parameter increases.

Of primary interest is what happens to the iterates when the iteration begins not at a fixed point, but

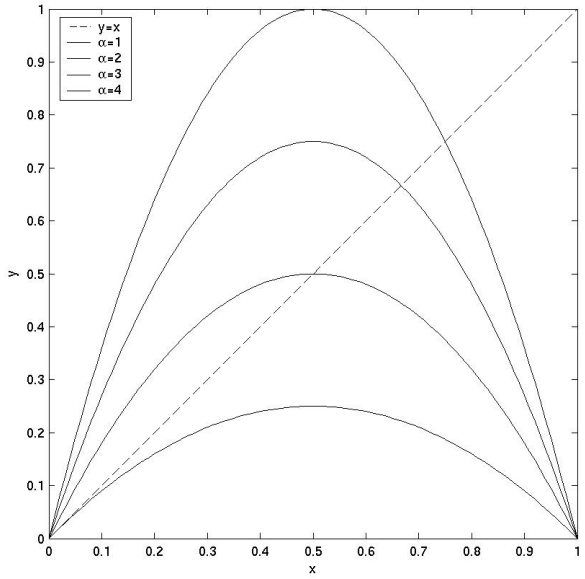


Figure 5: Graphical determination of fixed-points.

near a fixed point.

THEOREM: Suppose $g(x)$ is continuous, and $\{x_i\}_{i=1}^{\infty}$ is the sequence of iterates generated by FPI with $g(x)$. If $\lim_{i\to\infty} x_i = L$, then L must be a fixed point of $g(x)$.

PROOF:

$$\begin{aligned}
& \lim_{i\to\infty} x_i = L \ \text{(by hypothesis)} \\
\Rightarrow \quad & \lim_{i\to\infty} x_{i+1} = L \\
\Rightarrow \quad & \lim_{i\to\infty} g(x_i) = L \ \text{(by definition of } x_{i+1}) \\
\Rightarrow \quad & g(\lim_{i\to\infty} x_i) = L \ \text{(by continuity)} \\
\Rightarrow \quad & g(L) = L \ \text{(by hypothesis)}
\end{aligned}$$

In plain English, the theorem states that if a FPI converges, the value to which it converges must be a fixed point. Thus, fixed points are the only candidates for the limit of a sequence that originates from FPI. The theorem does *not* say that a FPI always converges. It does say, however, that *if* it converges, the limit value is not accidental: it's a fixed point.

Let's examine two closely related fixed-point iterations: one that converges and one that does not. More to the point, can we determine *why* one converges and the other doesn't?

EX: A tale of two fixed-point iterations.

For both FPI's, we will use the logistic map with the same initial condition $x_0 = 0.5$. Thus, for both

$$x_{i+1} \leftarrow \alpha x_i(1 - x_i)$$

The FPI's differ only in the value of the parameter α. Here are the two cases and their first few iterates to nine significant digits:

CASE I: $\alpha = 5/2 = 2.5$

$p \in \{0, 3/5\}$

$$\begin{aligned} x_0 &= 0.5 \\ x_1 &= 0.625 \\ x_2 &= 0.5859375 \\ x_3 &= 0.596624741 \\ x_4 &= 0.601659149 \\ x_5 &= 0.599163544 \\ &\ldots \\ x_{13} &= 0.600006516 \end{aligned}$$

CASE II: $\alpha = 7/2 = 3.5$

$p \in \{0, 5/7\}$

$$\begin{aligned} x_0 &= 0.5 \\ x_1 &= 0.875 \\ x_2 &= 0.3828125 \\ x_3 &= 0.826934814 \\ x_4 &= 0.500897695 \\ x_5 &= 0.874997180 \\ x_6 &= 0.382819904 \\ &\ldots \\ x_n &= 0.382819... \\ x_{n+1} &= 0.826940... \\ x_{n+2} &= 0.500884... \\ x_{n+3} &= 0.874997... \end{aligned}$$

The behaviors of these two FPI's are dramatically different. For Case I, the iteration appears to be converging in oscillatory fashion toward the larger fixed point $p = 3/5 = 0.6$. For Case II, the iteration appears to be generating a period-4 orbit. It is tempting to suggest that if the initial condition were closer to the upper fixed point of $p = 5/7$ in Case II all would be well, and the iteration would converge. But this is not the case. No matter how close the initial condition is to the fixed point, unless it is exactly (to machine precision) the value $x_0 = p = 5/7$, the iterates in Case II wander away from the fixed point. It is as if the fixed point and the iterates are magnets. In Case I, the magnets have opposite polarities and the iterates are attracted by the fixed point. In Case II, it is as if the fixed point and the iterates have the same polarities, in which case the fixed point repels the iterates. This is all very strange. Why the difference?

4.3 The Fixed-Point Iteration Theorem

The fundamental difference between Cases I and II above is the slope of the iteration function $g(x)$ at the fixed point p. That is, Cases I and II differ primarily in $g'(p)$. Specifically, for Case I, $|g'(p)| < 1$, and for Case II $|g'(p)| > 1$. This observation, in fact, lies at the heart of the Fixed-Point Iteration Theorem (FPI Theorem).

Fixed-Point Iteration THEOREM: Suppose that $g(x)$ and $g'(x)$ are continuous on $[a,b] = [p-\delta, p+\delta]$ ($\delta > 0$) containing the unique fixed point p of $g(x)$, and let $x_0 \neq p$ be any starting approximation to p within this interval.

1. If $|g'(x)| \leq K < 1$ for all $a \leq x \leq b$ then the FPI $x_{i+1} \leftarrow g(x_i)$ converges to p, and p is called an *attracting fixed point* or simply an *attractor*.
2. If $|g'(x)| > 1$ for all $a \leq x \leq b$ then the FPI diverges and p is called a *repelling fixed point* or simply a *repulsor*.

However, before we can prove this important theorem, we need to revisit an idea from Math 235, namely the Mean Value Theorem for Derivatives.

ASIDE: Mean Value Theorem for Derivatives (MVTD): If $f(x)$ is continuous on $[a,b]$ and differentiable on (a,b), then there exists $a < c < b$ such that

$$f'(c) = \frac{f(b) - f(a)}{b-a} \tag{57}$$

or equivalently

$$f(b) - f(a) = (b-a)f'(c) \tag{58}$$

Figure 6 illustrates the MVTD, which in plain English says that, for a differentiable function, there must be a point c in the interval of differentiability for which the instantaneous rate of change of the function is equal to its average rate of change over the entire interval. A related theorem, Rolle's Theorem, is the special case of the MVTD for which $f(b) = f(a)$, in which case the average rate of change is zero.

We are now in the position to sketch the proof of the FPI Theorem. Let p be a fixed point of the iteration function $g(x)$. Also let x_0 be an initial guess at p and x_i denote the ith iterate of the FPI. Finally, let $e_i \equiv p - x_i$ denote the error of the ith iterate relative to fixed point. Thus, if the sequence of errors e_i tends to zero, the FPI converges to p. On the other hand, if the sequence of errors diverges, the FPI fails to converge to p. We now examine the errors of the first few iterates with a little help from the second variant of the MVTD (Eq. 58). The initial error is given by

$$e_0 = p - x_0 \tag{59}$$

After the first iteration, the error is

$$e_1 \quad = \quad p - x_1$$

$$
\begin{aligned}
&= p - g(x_0) \quad \text{(definition of } x_1\text{)} \\
&= g(p) - g(x_0) \quad \text{(definition of fixed point)} \\
&= g'(c_0)(p - x_0) \quad \text{(MVTD)} \\
&= g'(c_0)e_0 \quad \text{(definition of } e_0\text{)}
\end{aligned} \tag{60}
$$

where c_0 lies between p and x_0. Notice that the error has diminished in magnitude if $|g'(c_0)| < 1$. Similarly,

$$
\begin{aligned}
e_2 &= p - x_2 \\
&= g'(c_1)e_1 \quad \text{(as above)} \\
&= g'(c_1)g'(c_0)e_0 \quad \text{(Eq. 60)}
\end{aligned} \tag{61}
$$

where c_1 lies between p and x_1. To generalize, after n iterations, the error is given by the following:

$$
\begin{aligned}
e_n &= p - x_n \\
&= g'(c_{n-1})g'(c_{n-2})...g'(c_1)g'(c_0)e_0 \\
&= \Pi_{k=0}^{n-1} g'(c_k)e_0
\end{aligned} \tag{62}
$$

where c_k lies between p and x_k, and Π is a mathematical symbol similar to Σ (for summation) that means *product*. From Eq. 62, it is clear that if the derivative of the iteration function is less than one in absolute value, the magnitude of the error of each iterate is less than that of the previous iterate. Thus, as $n \to \infty$, the error diminshes toward zero, and the FPI converges. We now formally present the FPI theorem, which is based upon the error analysis above.

Let's apply the FPI THEOREM to our previous examples with the logistic map, for which $g'(x) = \alpha(1-2x)$.

CASE I: $\alpha = 5/2 = 2.5$

$p \in \{0, 3/5\}$

$$
\begin{aligned}
g'(p_2) &= 5/2[1-2(3/5)] = -1/2 \\
&\Rightarrow |g'(p_2)| < 1 \\
&\Rightarrow p_2 = 3/5 \text{ is an attactor}
\end{aligned}
$$

CASE II: $\alpha = 7/2 = 3.5$

$p \in \{0, 5/7\}$

$$
\begin{aligned}
g'(p_2) &= 7/2[1-2(5/7)] = -3/2 \\
&\Rightarrow |g'(p_2)| > 1 \\
&\Rightarrow p_2 = 5/7 \text{ is a repulsor}
\end{aligned}
$$

Thus the results of our numerical experiment are consistent with the theory, because the theory predicts that the second fixed point p_2 switches from attractor to repulsor when α increases from 2.5 to 3.5.

Strictly speaking, the hypothesis of the FPI Theorem requires that we test the derivative of the iteration function over an *interval* containing p, not just at p itself. However, because the iteration function and its derivative are continuous, what is true at the fixed point must also be true for at least a small interval containing the fixed point.

Before we leave this section, it is instructive to discuss the meaning of K in Part (1) of the FPI Theorem. First, K can be viewed as the least upper bound of $g'(x)$ on the interval $[p-\delta, p+\delta]$. If $K = 1/2$, for

example, the error diminishes roughly by a factor of two with each iteration. This implies that each additional iteration yields an additional *bit* of accuracy. On the other hand, if $K = 1/10$, the error diminishes by a factor of approximately 10 with each iteration. Thus, each iteration yields another significant *digit*. However, if $K = 9/10$, the error is reduced only by 10 percent with each iteration; hence convergence to the fixed point is slow. We conclude with some additional remarks about convergence based upon the sketch of the proof of the FPI Theorem, followed by an algorithm for FPI in pseudocode.

REMARKS:

1. If $0 < g'(x) < 1$, then convergence is *monotone* because the error does not change sign from iteration to iteration.
2. If $-1 < g'(x) < 0$, then convergence is *oscillatory* because the error alternates sign with successive iterations. Note that convergence to $p_2 = 0.6$ for the logistic map (Case I) is oscillatory as expected.
3. The rate of convergence is determined by the least upper bound K in the sense that the smaller the value of K, the faster the rate of convergence.

ALGORITHM 4.1: Fixed-Point Iteration

given iteration function $g(x)$ *(user defined function)*
input initial guess x_1
initialize x_0 *to outrageous value*
input relative error tolerance ε
input iteration limit i_{max}
$i \leftarrow 0$ *(initialize the iteration counter)*

repeat until $|x_1 - x_0| \leq x_1\varepsilon$ *or* $i > i_{max}$
$| \; x_0 \leftarrow x_1$
$| \; x_1 \leftarrow g(x_0)$
$| \; i \leftarrow i + 1$
$| \; \_\_$

output x_1

HW1: Let $g(x) = \frac{1}{2}\cos(x)$, $0 \leq x \leq \pi/2$. Using a calculator, find the fixed point p of the iteration $x_{i+1} \leftarrow g(x_i)$ to five significant digits. Also perform the following

a) Carefully sketch by hand the functions $y = x$ and $y = g(x)$ on the same axes. Their intersection occurs at $x = g(x)$; that is, at the fixed point p. *Estimate* p from your sketch and use this value as your initial guess x_0.

b) Evaluate $g'(x_0)$. Based on this value, do you expect the FPI to converge? If so, monotonically or in oscillatory fashion?

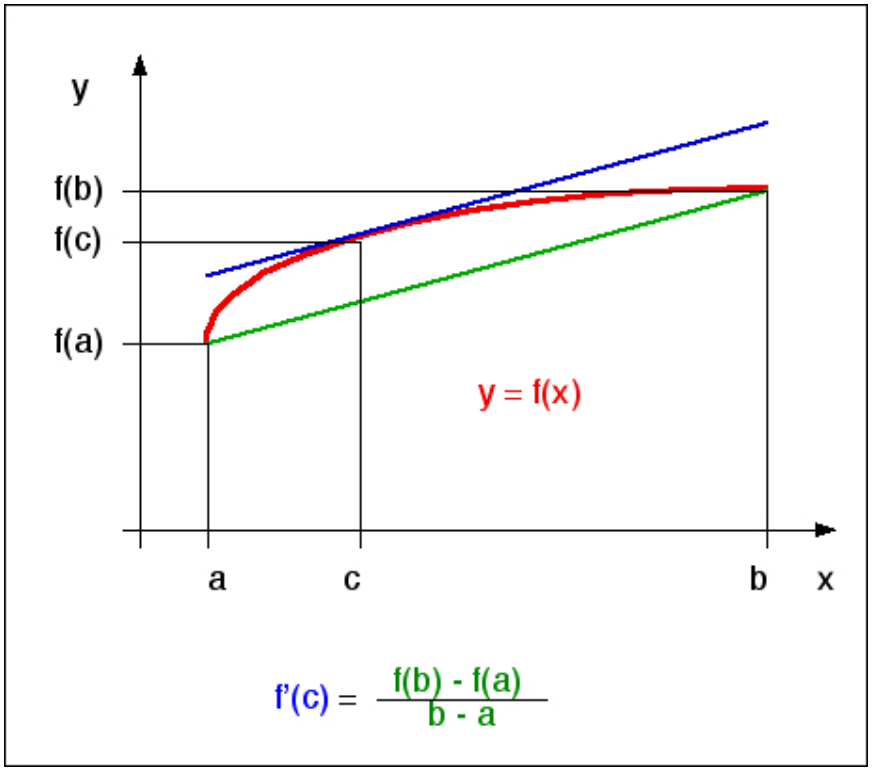


Figure 6: Mean Value Theorem.

c) How fast do you expect the iteration to converge if it converges? Estimate the number of iterations you think it will for each additional digit of accuracy. When you have your final result, comment on whether or not your actual experience was commensurate with what you anticipated?

4.4 Exercises

1. Consider the fixed point iteration $x_{n+1} = \frac{x_n}{2} + \frac{3}{2x_n}$.
   (a) Find the fixed point(s).
   (b) What does the fixed point iteration theorem tell us about this FPI?
   (c) Find the interval(s) of convergence for the above FPI.

2. Let $g(x) = \dfrac{x^2+2}{3}$. For any starting value $1 < x_0 < 2$ the fixed point iteration $x_{i+1} = g(x_i)$ always converges to $x = 1$. Show why.

3. Create a careful sketch of the graph of $y = \frac{x^2+2}{3}$ and $y = x$ on the interval [0,3]. What FPI does this graph correspond to? Illustrate the idea of a cobweb diagram for two different initial conditions. How many fixed points are there? What is the state of the fixed point(s)?

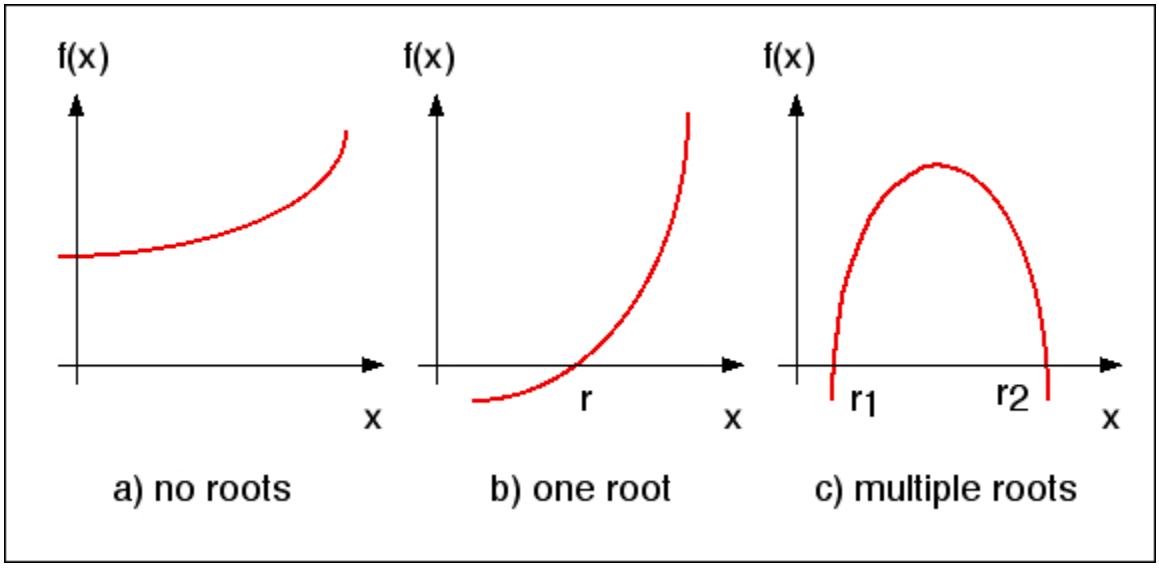


Figure 7: Roots of nonlinear functions.

5 Part II: Rootfinding Methods

In the previous section we discussed fixed-point iteration, a type of *iterative* process. Many numerical procedures are iterative in nature. Among the most important iterative techniques are rootfinding methods that solve nonlinear equations.

DEF: Suppose $f(x)$ is a continuous function on the closed interval $[a,b]$. If there exists $r \in [a,b]$ such that $f(r) = 0$, then r is said to be a *root* or *zero* of $f(x)$. In plain English, roots are the x-intercepts of functions that cross the x axis.

NOTE: Whereas a linear function whose slope is non-zero has a unique root, nonlinear functions may have no roots, a unique (one and only one) root, or multiple roots, as illustrated in Fig. 7. Rootfinding methods are unnecessary for linear problems. Thus, the methods of this section will almost always apply to *nonlinear* functions. Before we proceed to the methods themselves, it would be well to give a couple of examples that underscore the necessity of numerical rootfinding methods.

EX1: The Fundamental THEOREM of Algebra (FTA)

DEF: If $a_n \neq 0$, then a function of the form

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \ldots a_{n-1} x^{n-1} + a_n x^n$$

is called a *polynomial* in x of *degree* (order) n.

REMARKS: In general, the graphs of polynomials can be quite "wiggly." In fact, it can be proven that the graph of a polynomial of degree n can change direction as many as $n-1$ times. From this simple fact follows the Fundamental Theorem of Algebra (FTA): An nth degree polynomial with real coefficients a_i has at most n real roots. By way of analogy, think of the polynomial as the Shenandoah River, which

is undulating, and the x-axis as I-81, which is relatively straight through the Valley. Each intersection of these two, at a bridge, corresponds to a root of the polynomial. (The FTA can be restated in the following way if one allows the coefficients a_i and the independent variable x to be complex numbers: An nth degree polynomial has exactly n (complex) roots counting multiplicities.)

Let's consider the general form of three low-order polynomials: the *quadratic* ($n = 2$), the *cubic* ($n = 3$), and the *quartic* ($n = 4$) , given explicitly below.

$$\begin{aligned} P_2(x) &= a_0 + a_1 x + a_2 x^2 \\ P_3(x) &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 \\ P_4(x) &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 \end{aligned}$$

The roots of a quadratic are readily found by solving $P_2(r) = 0 = a_0 + a_1 r + a_2 r^2$ for r by the quadratic formula. No need for numerical solutions here. It turns out that there are general but obscure closed-form techniques for determining the roots of cubic and quartic polynomials. Thus, no need for numerics for $n = 3$ or $n = 4$ either. However, for $n > 4$ there are no analytical techniques that work in the most general case. We don't mean to imply that the analytical techniques just aren't known. It can be proven, in fact, that such techniques do not exist! Thus, the *only* completely general way to find the roots of polynomials of degree $5 \leq n$ is by numerical rootfinding techniques! Analysis has its limits.

EX2: The Rayleigh Pitot Formula

Presumably most of you have flown on a private or commercial aircraft during you lifetime. All aircraft, from Cessnas to the Space Shuttle have an on board *air data system*. Roughly speaking, the air-data system is the "speedometer" of an aircraft, which provides an indicator of speed as Mach number, the ratio of airspeed to the speed of sound. At sea level, the speed of sound is about 700 mph, but it varies gradually with temperature, diminishing with altitude. Commericial airliners, which are currently limited to about Mach 0.85, fly at approximately 600 mph. In contrast, before it retired, the supersonic Concorde cruised at Mach 1.6 or about 1100 mph, and during the first moments of atmospheric re-entry, the Shuttle is traveling at a whopping Mach 25.

An air data system extracts Mach number based upon the ratio of two pressures. The relationship between pressure and Mach number M_∞ is given by the Rayleigh Pitot formula. Let P_∞ be the ambient atmospheric pressure and P_t be the total (or impact) pressure measured by the pitot-static system on the noseboom of an aircraft (or elsewhere). Let their ratio be denoted R. That is, $R = P_\infty / P_t$. The Rayleigh Pitot Formula, which is derived from the relationships of gas dynamics, states

$$R(M_\infty) = \begin{cases} \left[1 + \frac{\gamma-1}{2} M_\infty^2\right]^{\frac{-\gamma}{\gamma-1}} & \text{if } M_\infty \leq 1 \\ \left[\frac{2}{(\gamma+1)M_\infty^2}\right]^{\frac{\gamma}{\gamma-1}} \left[\frac{2\gamma M_\infty^2 - (\gamma-1)}{\gamma+1}\right]^{\frac{1}{\gamma-1}} & \text{if } M_\infty > 1 \end{cases} \tag{63}$$

For air under normal conditions, $\gamma = 1.4$.

The Rayleigh Pitot Formula gives pressure ratio R as a function of Mach number M_∞; that is $R = f(M_\infty)$. But an air data system needs the inverse function $M_\infty = f^{-1}(R)$. If the aircraft is traveling at subsonic speed ($M_\infty < 1$), then a closed-form inverse can be found to the Rayleigh Pitot Formula. If

the speed is supersonic ($M_\infty > 1$), however, a closed-form inverse *cannot* be found. The way out of this dilemma is to turn the Rayleigh Pitot Formula into a rootfinding problem and to use a numerical technique that converges quickly, because someone's life is riding on the answer.

5.1 Fixed-Point Iteration for Rootfinding

With a little bit of manipulation, many rootfinding problems can be re-cast as Fixed-Point Iteration (FPI) problems, so that the techniques of the previous section can be applied to the solution process. Here's how it works (when it works).

Our initial problem, given $f(x)$, is to find r such that $f(r) = 0$. Suppose that $f(r) = 0$ can be re-written as $r = g(r)$. In this case, a root r of f is also a fixed point of g. The root r may then be found by iterating on $x_{i+1} \leftarrow g(x_i)$ by ALG 4.1, *provided the FPI converges*. Here is an example to illustrate how it works.

EX: Suppose we want to find the roots of $f(x) = x^5 - 2x + 1$, a quintic polynomial for which closed-form methods do not exist. An obvious root is $r = 1$, but there is at least one more real root between 1/2 and 3/4. How do we know this? Consider that $f(1/2) > 0$ but $f(3/4) < 0$. Because f is continuous, it must therefore pass through zero somewhere on the interval $[0.5, 0.75]$.

Let's begin with our quintic expressed at the unknown root r, where f vanishes by definition.

$$f(r) = r^5 - 2r + 1 = 0 \tag{64}$$

By a little manipulation , we arrive at the following:

$$\begin{aligned} r^5 - 2r + 1 &= 0 \\ \Rightarrow 2r &= r^5 + 1 \\ \Rightarrow r &= \frac{1}{2}(r^5 + 1) \end{aligned}$$

Thus the root r satisfies $r = g(r)$ for the iteration function $g(x) = \frac{1}{2}(x^5 + 1)$. So, let's try iterating on

$$x_{i+1} \leftarrow g(x_i) \tag{65}$$

with $x_0 = 0.5$ and see what happens. The iterates, to nine significant digits and obtained by ALG 4.1, are as follows:

$$\begin{aligned} x_0 &= 0.5 \\ x_1 &= 0.515625 \\ x_2 &= 0.518223837 \\ x_3 &= 0.518687746 \\ x_4 &= 0.518771542 \\ x_5 &= 0.518786710 \\ x_6 &= 0.518789546 \end{aligned}$$

$$
\begin{aligned}
x_7 &= 0.518789954 \\
&\ldots \\
x_{11} &= 0.518790064
\end{aligned}
$$

The 11th iterate is correct in all nine significant digits! This almost seems too good to be true: an answer of extraordinary accuracy by a simple method in relatively few iterations. Suppose we naively tried to find the other root ($r = 1$) by the same method, starting, say, at $x_0 = 1.1$. Here are the iterates in the latter case:

$$
\begin{aligned}
x_0 &= 1.1 \\
x_1 &= 1.305255 \\
x_2 &= 2.394291594 \\
x_3 &= 39.84188892 \\
x_4 &= 50196057.74 \\
x_5 &= 1.59 \times 10^{38} \\
x_6 &= \text{Error!}
\end{aligned}
$$

Yikes! This is as bad as it gets. Not only did the iteration diverge, the computation "blew up" in only six iterations. It appears we forgot the lesson learned in the previous section: FPI's converge only if $|g'(r)| < 1$. For $g(x) = \frac{1}{2}(x^5 + 1)$, $g'(x) = \frac{5}{2}x^4$. For $r_1 = 0.518790064$, $g'(r_1) \approx 0.18$, which implies that convergence is monotonic and that the error is reduced by more than 80 percent with each successive iteration when near the root. On the other hand, for $r_2 = 1$, $g'(r_2) = 5/2 >> 1$, so the iteration diverges, and quickly.

The moral of this story should be clear: FPI can be adapted for rootfinding but the method is far from foolproof. Here are some things to consider before resorting to FPI to find roots.

REMARKS:

1. FPI does not always converge.
2. If there are multiple roots, FPI may converge to the *wrong* root.
3. The rate of convergence, which depends on $|g'(r)|$, can vary wildly depending upon the particular problem.
4. In general, the form $r = g(r)$ obtained from $f(r) = 0$ is non-unique; that is, there are usually multiple ways to rearrange the problem. It may take trial and error to find which rearrangement works for a desired root.

HW1: Consider the quadratic $f(x) = x^2 - 3x + 2$ whose roots, by factoring are $r \in \{1, 2\}$. Pretend you don't know the roots and try to find *both* by FPI. Explain what happens and why.

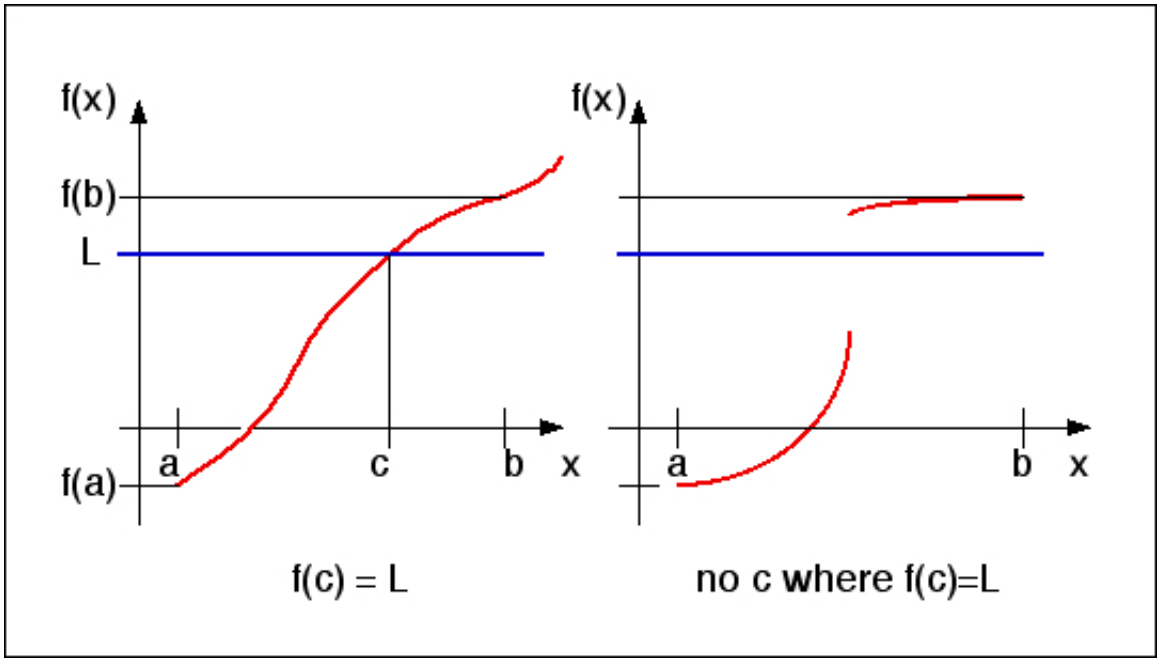


Figure 8: Intermediate Value Theorem: a) for continuous f; and b) why continuity is essential.

5.2 Bisection

If you need a foolproof method for finding a root, bisection is your choice. In truth, no rootfinding method is foolproof (because fools are so ingenious), but bisection is guaranteed to converge provided you have set up the problem correctly. To do that, you must select the search interval $[a,b]$ such that 1) you know at least one root exists in the interval (*existence*), and 2) you know no more than one root exists in the interval (*uniqueness*).

We'll address the existence issue here, and the uniqueness issue in the context of Newton's method. Regarding existence, rootfinding is a bit like game hunting. It doesn't make sense to hunt a lion (with a camera please, lions are endangered) if there are no lions around. Same with roots. Fortunately, the Intermediate Value Theorem (IVT) from Math 235 is all that we need to establish the existence of a root.

Intermediate Value THEOREM (IVT): Suppose that $f(x)$ is continuous on the closed interval $[a,b]$ and L is *any* number between $f(a)$ and $f(b)$; that is either $f(a) < L < f(b)$ or $f(b) < L < f(a)$. Then there exists at least one $a < c < b$ such that $f(c) = L$.

Figure 8a demonstrates the IVT for a continuous function, and Fig. 8b shows why continuity is crucial to the theorem. In plain English, the IVT says that a continuous function must pass through all its intermediate values.

COROLLARY: (Existence of a Root) If $f(x)$ is continuous on $[a,b]$ and $f(a) \cdot f(b) < 0$, then there exists $a < r < b$ such that $f(r) = 0$; that is, r is a root of $f(x)$.

Before the simple proof, let's make sure we understand the hypothesis. If $f(a) \cdot f(b) < 0$ then the

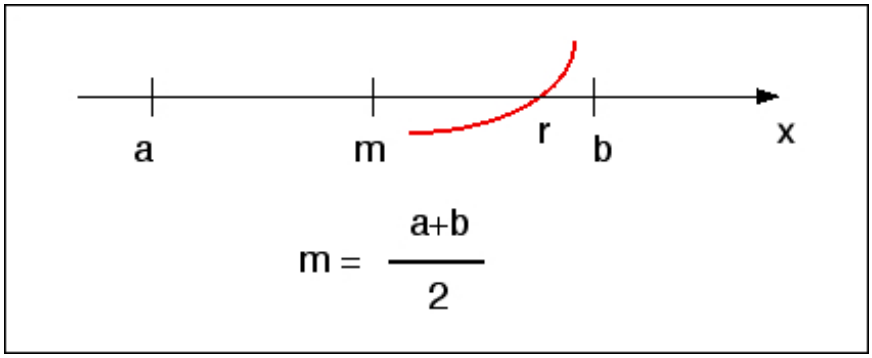


Figure 9: Principle of bisection.

function *changes sign* on the interval. Because f is continuous, it cannot change sign without somewhere passing through $L = 0$.

PROOF: If $f(a) \cdot f(b) < 0$, then $L = 0$ lies between $f(a)$ and $f(b)$. Hence by the IVT, there exits an $a < r < b$ such that $f(r) = L = 0$. By definition then, r is a root of $f(x)$.

The method of bisection begins with the root r trapped in the interval $[a,b]$, as shown in Fig. 9. The interval is then bisected at its midpoint $m = \frac{1}{2}(a+b)$. Three possibilities then exist: 1) the root lies in the subinterval $[a,m]$, 2) the root lies in the subinterval $[m,b]$, or 3) $r = m$. The third option is highly unlikely, but if the midpoint m were indeed the root, we would know it because $f(m)$ would vanish (be zero). One of the other possiblities is far more likely. If the function changes sign on $[a,m]$ then the root lies in that subinterval. On the other hand, if it doesn't change sign on $[a,m]$ the root must lie in the second subinterval. Once the subinterval containing the root is identified, the process is repeated until the subinterval containing the root is as small as the desired accuracy.

Let's illustrate the bisection method with a very simple example.

EX: Consider $f(x) = x^2 - 2$, the roots of which are $r \in \{-\sqrt{2}, +\sqrt{2}\}$. We are going to use bisection to actually *compute* the positive square root of 2. The positive root is known to lie between $a = 1$ and $b = 2$, because $f(x)$ changes sign on this interval. Thus, we will use $[a_0, b_0] = [1,2]$ as our starting interval. Note that $f(a_0) \cdot f(b_0) = f(1) \cdot f(2) = (-1)(2) = -2 < 0$. This establishes that a root lies in the interval. We'll assume for now it is unique and deal with that issue later.

Step 0:

$$\begin{aligned} m_0 &= \frac{a_0 + b_0}{2} = 1.5 \\ f(a_0) \cdot f(m_0) &= f(1) \cdot f(1.5) = (-1)(0.25) = -0.25 < 0 \\ \Rightarrow r &\in [a_0, m_0] \\ a_1 &\leftarrow a_0 = 1 \\ b_1 &\leftarrow m_0 = 1.5 \end{aligned}$$

Step 1:

$$\begin{aligned} m_1 &= \frac{a_1+b_1}{2} = 1.25 \\ f(a_1)\cdot f(m_1) &= f(1)\cdot f(1.25) = (-1)(-0.4375) = 0.4375 > 0 \\ \Rightarrow r &\notin [a_1, m_1] \;\Rightarrow\; r \in [m_1, b_1] \\ a_2 &\leftarrow m_1 = 1.25 \\ b_2 &\leftarrow b_1 = 1.5 \end{aligned}$$

Step 2:

$$\begin{aligned} m_2 &= \frac{a_2+b_2}{2} = 1.375 \\ f(a_2)\cdot f(m_2) &= f(1.25)\cdot f(1.375) = (-0.4375)(-0.109375) > 0 \\ \Rightarrow r &\notin [a_2, m_2] \;\Rightarrow\; r \in [m_2, b_2] \\ a_3 &\leftarrow m_2 = 1.375 \\ b_3 &\leftarrow b_2 = 1.5 \end{aligned}$$

Step 3:

$$\begin{aligned} m_3 &= \frac{a_3+b_3}{2} = 1.4375 \\ f(a_3)\cdot f(m_3) &= f(1.375)\cdot f(1.4375) = (-0.109375)(0.06640625) < 0 \\ \Rightarrow\; r &\in [a_3, m_3] \\ a_4 &\leftarrow a_3 = 1.375 \\ b_4 &\leftarrow m_3 = 1.4375 \end{aligned}$$

Step 4:

$$\begin{aligned} m_4 &= \frac{a_4+b_4}{2} = 1.4065 \\ &\ldots \end{aligned}$$

The process continues until the desired accuracy is attained. Several things are clear about bisection. First, it always converges, provided the root is trapped at each step. Second, the convergence is not monotonic. At some steps, the value of the midpoint is actually farther from the exact root than was the midpoint of the previous step. In the example above, $m_1 = 1.25$ is farther from $r = 1.414...$ than was the initial midpoint $m_0 = 1.5$. So, bisection is a plodding method: dependable but slow. Let's quantify the error property of the bisection method.

Consider the absolute value of the absolute error after the nth step. The root at each step is approximated by the new midpoint; thus, $|e_n| = |r - m_n|$. However, as Fig. 10 shows, the error can never be worse than half the interval width. That is,

$$|e_n| = |r - m_n| \leq \frac{b_n - a_n}{2} \tag{66}$$

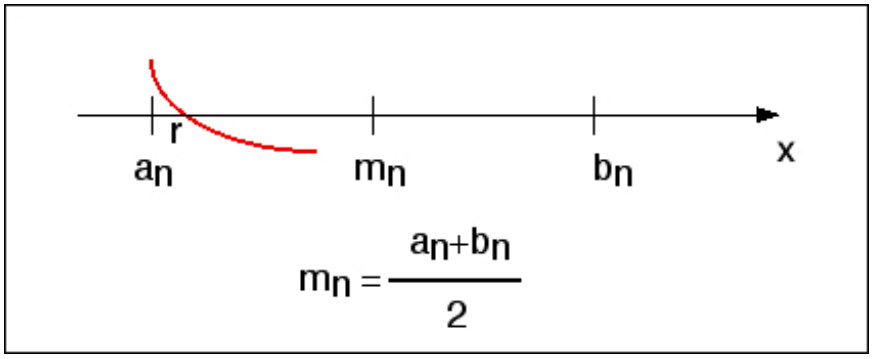


Figure 10: Error of bisection method.

But each new interval is half the width of its predecessor. Thus

$$\begin{aligned} |e_n| &\leq \frac{b_n - a_n}{2} \\ &= \frac{b_{n-1} - a_{n-1}}{4} \\ &= \frac{b_{n-2} - a_{n-2}}{8} \\ &\ldots \\ &= \frac{b_0 - a_0}{2^{n+1}} \end{aligned}$$

To say it succinctly, if $r \in [a,b]$, the absolute error after the n bisection step is bounded as follows:

$$|e_n| \leq \frac{b-a}{2^{n+1}} \tag{67}$$

The following comments derive from Eq. 67:

REMARKS:

1. The error *bound* (not necessarily the error itself) is halved for each additional iteration.
2. This is equivalent to a guarantee of one additional *bit* of precision per iteration.
3. Thus machine accuracy is guaranteed in no more than n iterations, where n is the number of bits in the mantissa.
4. To increase the accuracy from single precision to double precision would require approximately another n iterations!
5. To summarize, bisection always converges (if you follow the rules), but it is painstakingly slow.

Here is the bisection algorithm in pseudocode, the essence of the example above, but all cleaned up.

ALGORITHM 5.1: Rootfinding by Bisection

given $f(x)$ (user defined function)
input interval limits a, b
input absolute error tolerance δ
input number bits in mantissa n
$fa \leftarrow f(a)$ (evaluate f at endpoint a)

if $fa \cdot f(b) > 0$
    output error message
    stop
end if

set fm to something outrageous
$i \leftarrow 0$ (initialize loop counter)

do until $|fm| < \delta$ or $i > n$
| $m \leftarrow (a+b)/2$ (midpoint of interval)
| $fm \leftarrow f(m)$ (evaluate f at midpoint and save)
| if $fa \cdot fm \leq 0$
| $b \leftarrow m$ (keep a, replace b)
| else
| $a \leftarrow m$ (keep b, replace a)
| $fa \leftarrow fm$ (rename fm)
| end if
| $i \leftarrow i+1$
|_____

output m (approximates root r) and fm (indication of error)

Can you see an advantage to storing $f(a)$ and $f(m)$ as fa and fm, respectively, in the algorithm above?

5.3 Newton's Method

Bisection is trustworthy, but it is plodding. What if we need fast convergence to a root, for, say, a real-time application like a flight-control system? Can we do a better job of estimating the root at each step of the iteration? If so, the iteration may converge faster. In this regard, Newton's method is spectacular. But before we use it, we'd better make sure that the root exists and is unique. We've already addressed the existence issue in the previous subsection. Let's now give some attention to uniqueness.

ASIDE–Rolle's THEOREM: (Recall that Rolle's Theorem is a special case of the Mean Value Theorem for Derivatives.) If $f(x)$ is continuous on $[a,b]$ and differentiable on (a,b), and if $f(a) = f(b)$ (this is the

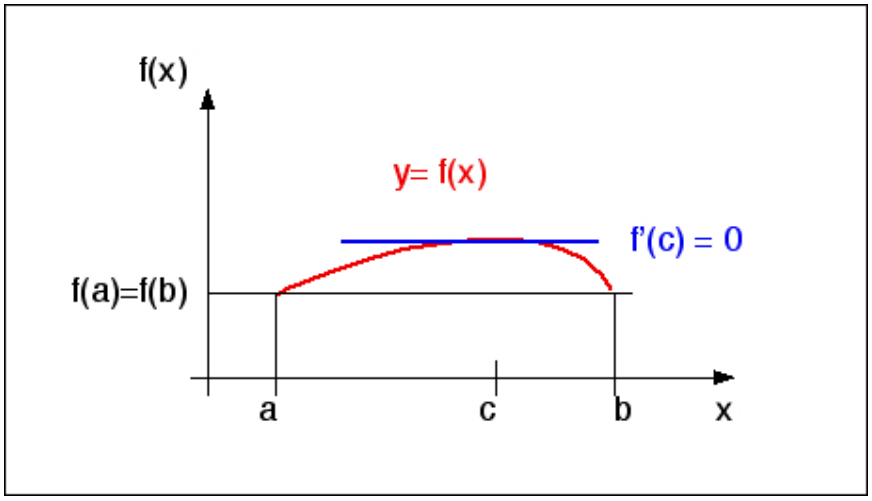


Figure 11: Rolle's Theorem illustrated.

special restriction), then there exists at least one $a < c < b$ such that $f'(c) = 0$. In plain English, Rolle's Theorem states that, if the average rate of change of a differentiable function is zero on an interval, then the instantaneous rate of change on the interval must also be zero at at least one point in the interval. Figure 11 illustrates Rolle's Theorem.

We now make use of Rolle's Theorem to prove the uniqueness of a root in the event that a root exists and the function is continuous and monotonic.

THEOREM (Uniqueness of a Root): If $f(x)$ satisfies the conditions for the existence of a root in the interval $[a,b]$, and in addition $f(x)$ is differentiable on (a,b) with either $f'(x) > 0$ or $f'(x) < 0$ (that is, f is either strictly increasing or it is strictly decreasing, in which case f is said to be *monotonic*), then $f(x)$ has *exactly one* root $a < r < b$. The proof is by contradiction, using Rolle's Theorem.

PROOF: Suppose there are two roots r_1 and r_2 in the interval $[a,b]$. Then, by definition $f(r_1) = 0 = f(r_2)$. By Rolle's Theorem, there must be $a \leq r_1 < c < r_2 \leq b$ such that $f'(c) = 0$. But this cannot be because of our assumption of monotonicity. Therefore, there cannot be two roots (or more than one for that matter).

Figure 12 illustrates how the uniqueness theorem rules out the possibility of multiple roots in the interval.

REMARKS: Existence + Uniqueness $\Rightarrow$ *one and only one* root exists in the interval of interest.

We are now ready to describe Newton's method. *Newton's method should never be applied unless the conditions of existence and uniqueness have both been firmly established.*

Let $f(x)$ be a differentiable and monotonic function with a root r in the interval $[a,b]$, and let x_0 be an initial guess at the root. Let's now construct the tangent line at the point $[x_0, f(x_0)]$ on the function. The tangent line (shown in blue in Fig. 13) has the equation

$$y = f(x_0) + f'(x_0)(x - x_0) \tag{68}$$

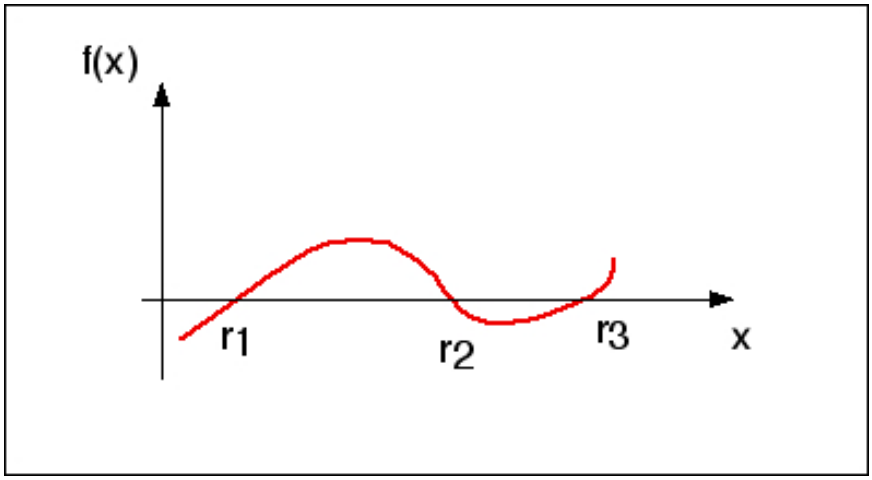


Figure 12: Multiple roots, as shown, are ruled out if function f is monotonic.

(Convince yourself this is true by evaluating y when $x = x_0$ and by computing dy/dx.) It makes sense to use the x intercept of the tangent line as the next guess at the root r. Let's call this value x_1, for which $y = 0$ by the definition of the x intercept. Thus, the intercept is found by solving the following equation for x_1:

$$0 = f(x_0) + f'(x_0)(x_1 - x_0) \tag{69}$$

for which

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{70}$$

Thus $x_1 \approx r$. Now let's repeat the process using x_1 as the new initial guess. There is no need to re-derive everything. The final result is just Eq. 70 with a shift of indices, namely

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \tag{71}$$

whereby $x_2 \approx r$ even better than x_1. Yet another round gives

$$x_3 = x_2 - \frac{f(x_2)}{f'(x_2)} \tag{72}$$

Clearly, we have the beginnings of a new algorithm. In a nutshell, Newton's method is an iteration of the form

$$x_{i+1} \leftarrow x_i - \frac{f(x_i)}{f'(x_i)} \tag{73}$$

Notice that, unlike bisection, Newton's method requires the evaluation of both the function f and its derivative f' at each iteration. Note also that Newton's method would run into trouble if $f'(x_i)$ were zero for some iteration i. However, if the uniqueness criterion (monotonicity) has been established first, then the derivative cannot vanish.

Of primary interest is the rate of convergence of Newton's method. Let's apply it to the same problem to which we applied the bisection method, namely the determination of the $\sqrt{2}$ by finding the positive zero

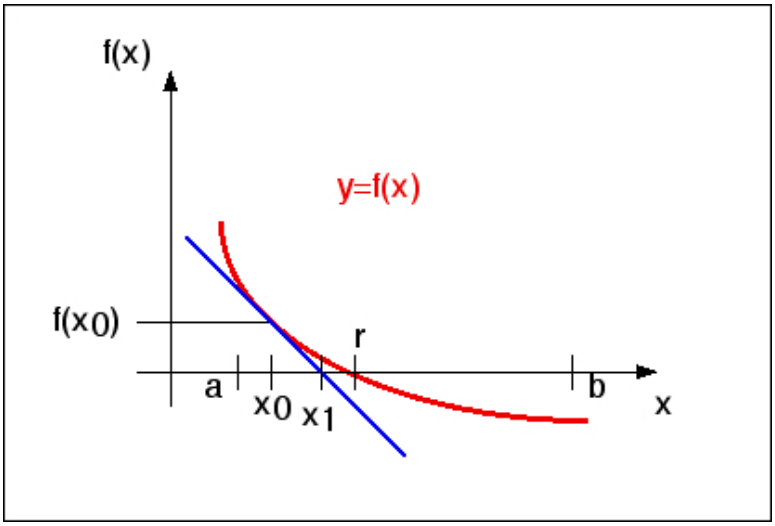


Figure 13: First Newton approximation x_1 of root r.

of $f(x) = x^2 - 2$. For this function $f'(x) = 2x$. We know $1 \leq r = \sqrt{2} \leq 2$ and also that $f'(x) > 0$ on $[1,2]$. So both the existence and uniqueness criteria are satisfied if $[a,b] = [1,2]$.

For this problem, the "nutshell" version of Newton's method looks like

$$x_{i+1} \leftarrow x_i - \frac{x_i^2 - 2}{2x_i} = \frac{x_i}{2} + \frac{1}{x_i} \tag{74}$$

For a head-to-head competition with bisection, let's assume the same initial guess as before, namely $x_0 = 1.5$, the midpoint of the interval. Table 5 shows the first three iterates of Newton's method.

i	x_i	$f(x_i)$
0	1.5	0.25
1	1.41666$\overline{6}$	0.0012817778
2	1.414215686	6.0065285×10^{-6}
3	1.414213562	-1.00553×10^{-9}

Table 5: Convergence typical of Newton's method.

There are several noteworthy comments to make about Newton's method for the example problem. First, 9-digit accuracy is attained in only three iterations! Second, that the iteration is converging to the root is clear from the third column, which should tend toward zero. Third, the number of correct significant digits is indicated by underlining. The initial guess is correct in only one digit, the next iteration in three digits, the next in six, and the last iteration in all digits! It appears that the number of correct significant digits is roughly doubling with each iteration. Is this rapid convergence a fluke for this particular problem or is it an attribute of Newton's method in general? Stay tuned!

But first, you may be wondering why anyone would apply Newton's method to such a trivial problem as computing $\sqrt{2}$. Can't I just get this value from my calculator? you ask. Yes, but did it ever occur to

you to wonder *how* your calculator extracts square roots? Most likely, by Newton's method! Now back to the issue of convergence.

It is also interesting to note that Newton's method is a special type of fixed-point iteration (FPI) for which the iteration function is

$$g(x) = x - \frac{f(x)}{f'(x)} \tag{75}$$

Recall that the convergence rate of FPI is related to the absolute vale of $g'(x)$ at the fixed point (or in this case, the root r). Essentially, the error of each new iteration is approximately $g'(r)$ times the error of the previous iteration. In the best of all possible worlds then, an iterative method would strive for $g'(r) = 0$. This wish becomes a reality for Newton's method!

HW1: From Eq. 75 derive $g'(x)$ for Newton's method, and show that $g'(r) = 0$.

In Math 448, should you continue with numerical analysis, you will learn that this wonderful trait of Newton's method results in what is known as *quadratic convergence*. From a practical point of view, the doubling of the number of significant digits of accuracy with each subsequent iteration is a characteristic of quadratic convergence. Typically Newton's method converges to single precision (6-7 digits) in only three iterations.

HW2: Suppose you know a root r to machine single precision but want to continue computing to obtain the value correct to double precision. Compare bisection and Newton's methods in regard to this task.

ALGORITHM 5.2: Rootfinding by Newton's Method

given $f(x)$ and $f'(x)$ (user-defined functions)
input or compute machine unit round-off error u
$\varepsilon \leftarrow \sqrt{u}$ *(relative error tolerance)*
input x_1 (initial guess at root r)
set x_0 to some outrageous value
input i_{max} (iteration safety limit)
$i \leftarrow 0$ *(initialize iteration counter)*

repeat until $|x_1 - x_0| < \varepsilon |x_1|$ *or* $i > i_{max}$
| $x_0 \leftarrow x_1$
| $fp = f'(x_0)$
| *if* $fp = 0$ *then*
| *output error message (derivative vanishes)*
| *stop*
| *end if*
| $x_1 \leftarrow x_0 - f(x_0)/fp$
| —

if $i > i_{max}$ *then*
output warning message (failed to converge completely)
end if

output x_1 *(which contains estimate of root* r*)*

The astute student may be wondering about the convergence criterion above. Note that for ALG 5.2 above, if $x_0 = r$ exactly, then $f(x_0) = 0$, in which case then $x_1 = x_0$. Thus, it makes sense to test the difference of x_1 and x_0. When the values are almost the same, we must be near the root. Because relative error is more meaningful than absolute error, and x_1 is closer to the exact value than x_0, an appropriate relative error test could assume the form

$$\frac{|x_1 - x_0|}{|x_1|} < \varepsilon \tag{76}$$

This test, however, would experience a problem if $x_1 = 0$. A possible divide by zero can be avoided by recasting Eq. 76 as follows:

$$|x_1 - x_0| < |x_1|\varepsilon \tag{77}$$

Finally, why the mysterious $\varepsilon = \sqrt{u}$ in ALG 5.2? Here's one for you to figure out. (HINT: It has to do with quadratic convergence.)

We close with a summary of Newton's method for rootfinding.

REMARKS:

1. Newton's method requires but a single initial guess at the root.
2. Newton's method converges dramatically fast (quadratically), typically giving double the number of significant digits of accuracy with each subsequent iteration, provided the conditions for existence and uniqueness of a root have first been established.
3. If the condition for uniqueness (monotonicity) has not been established, Newton's method can converge slowly or fail to converge altogether, typically due to a vanishing derivative.
4. Newton's method is computationally expensive in the sense that both the function and its derivative must be evaluated at each iteration. If the function is complicated, its derivative may be horrendously complicated.

The following algorithm is similar to Newton's method; however, it avoids the necessity of computing the derivative by requiring two initial guesses, from which the x intercept of the *secant line* is used to refine the approximation of the root at each iteration.

5.4 Secant Method (Optional)

Newton's method is the *Spitze* of rootfinding methods, but for very complicated functions it can be expensive because of the need to evaluate *two* functions: f and its derivative. If f is complicated, then f' may be outrageously messy. Secant method is similar to Newton's method, but it avoids the need to evaluate the derivative. As the name suggests, a secant line is used to approximate the root r of $f(x)$ rather than

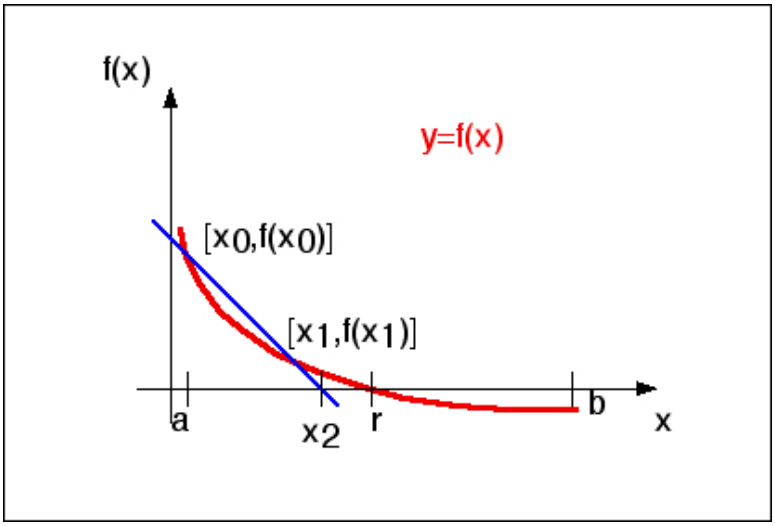


Figure 14: Rootfinding by secant method.

a tangent line, as shown in Fig. 14. Note that *two distinct* initial guesses at the root are required to start secant method: $[x_0, f(x_0)]$, and $[x_1, f(x_1)]$. The slope of the secant line through these two points is

$$m_1 = \frac{f(x_1) - f(x_0)}{x_1 - x_0} \tag{78}$$

in which case, the secant line is given by

$$y = f(x_1) + m_1(x - x_1) \tag{79}$$

Let's define x_2 as the x intercept of the secant line, namely the solution to

$$0 = f(x_1) + m_1(x_2 - x_1) \tag{80}$$

whereby $x_2 = x_1 - f(x_1)/m_1$. If the function f is monotonic on $[a,b]$ containing initial guesses x_0 and x_1, then x_2 will approximate the root r better than either initial guess. The process can then be repeated. In a nutshell, here is the recursion relation for secant method:

$$\begin{aligned} m_i &\leftarrow \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}} \\ x_{i+1} &\leftarrow x_i - \frac{f(x_i)}{m_i} \end{aligned} \tag{81}$$

Let's apply secant method to compute $\sqrt{2}$ by finding the positive root of $f(x) = x^2 - 2$. Note that $f(x)$ is monotonically increasing on the interval $[1,2]$. For our initial guesses let's take $x_0 = 1$ and $x_2 = 2$, neither of which is particularly good. The results of the iterations of secant method for this function are given in Table 6.

Several attributes of secant method are noteworthy. The first is that it converges quickly, not as fast as Newton's method, but typically to machine single precision in 5-6 iterations from decent initial guesses.

i	x_i	$f(x_i)$	m_i
0	1.0	-1.0	NA
1	2.0	2.0	3.0
2	$1.333\overline{3}$	$-0.222\overline{2}$	$3.333\overline{3}$
3	1.4	-0.04	$2.733\overline{3}$
4	1.414634146	0.001189768	2.814634146
5	1.414211438	-6.007×10^{-6}	2.828845584
6	1.414213562	-8.94×10^{-10}	2.828424803
7	1.414213562	-8.94×10^{-10}	ERROR!

Table 6: Iterates of secant method for $f(x) = x^2 - 2$.

However, secant method suffers from a serious flaw. Did you spot it in Table 6? Secant method is susceptible to catastrophic loss of significance near the root because computing the slope m_i involves the subtraction of pairs of numbers: x_{i-1} from x_i, and $f(x_{i-1})$ from $f(x_i)$. Worse, with one iteration too many, the algorithm experiences a divide by zero run-time error. Thus, secant method should be used very carefully, with particular attention to its convergence criterion. Moreover, we recommend that secant method NOT be used for any system where convergence failure can result in the loss of life or limb. With this caveat in mind, here is the full algorithm for secant method:

ALGORITHM 5.3: Rootfinding by Secant Method

given $f(x)$ *(user-defined function)*
input or compute machine unit round-off error u
$\varepsilon \leftarrow u^{2/3}$ *(relative error tolerance)*
input x_0 *and* x_1 *(initial guesses at root* r*)*
$f_0 \leftarrow f(x_0)$
set x_2 *to something outrageous*

repeat until $|x_2 - x_1| < \varepsilon |x_2|$
|
| $f_1 \leftarrow f(x_1)$
| $m \leftarrow \frac{f_1 - f_0}{x_1 - x_0}$
| $x_2 \leftarrow x_1 - \frac{f_1}{m}$
| $x_0 \leftarrow x_1$ *(shift operation)*
| $x_1 \leftarrow x_2$ *(shift operation)*
| $f_0 \leftarrow f_1$ *(shift operation)*
| ——

output (return) x_2 *(which contains estimate of root* r*)*

We close with several remarks about ALG 5.3.

REMARKS:

1. Because secant method uses more than one point at each iteration, it is called a *multistep method*.
2. Each iteration, however, requires only one *new* function evaluation, which is why the method is relatively efficient.
3. The convergence criterion for secant method is *crucial* to prevent a run-time crash!
4. Newton's method converges *quadratically*, which means that the number of digits of precision tends to double with each successive iteration. For secant method, the growth factor of the number of digits of precision is approximately 1.6 per iteration. The choice $\varepsilon = u^{2/3}$, therefore, is carefully chosen to halt the iteration when x_2 is correct in all significant digits, but to prevent a catastrophic additional iteration. Can you see why this choice?

HW1: Write a function $f(x)$, one of whose roots is $4^{1/3}$. Use secant method to find this root to 6 significant digits.

5.5 Exercises

1. Consider $f(x) = e^x - 3x^2$, which has three real roots at approximately -.46, 0.91, and 3.73.
   (a) By solving for x in the second term on the right of the equal sign, devise a fixed point iteration for finding the roots of $f(x)$. Will this algorithm converge to the unique fixed point on [0,2]?
   (b) By solving for x in the first term on the right of the equal sign, devise a fixed point iteration for finding the roots of $f(x)$. Will your algorithm converge to the unique fixed point on [0,2]?
2. Consider the following function which has a *discontinuity* at $x = 1$:
$$f(x) = \begin{cases} x & \text{if } x > 1 \\ 0 & \text{if } x = 1 \\ x-2 & \text{if } x < 1 \end{cases}$$
   (a) Does $f(x)$ have a root?
   (b) If you try the bisection method with bracketing interval $[-1,2]$, what happens? Will the bisection method fail? Will it converge to something?
3. Consider the function $f(x) = x^3 - 4$.
   (a) Show that $f(x)$ has a root $f \in [0,2]$.
   (b) Compute the first three iterates x_0, x_1, and x_2 when using the bisection method to search for the root r.
   (c) DO NOT continue with the bisection method, but if you were to continue, how precise would your approximation x_k be when $k = 10$ and $k = 20$?
   (d) Derive the iteration function $g(x)$ for Newton's method for this problem.

4. (a) Carry out 3 iterations of the fixed point method for solving $x = \frac{1}{x+1}$ with $x_0 = 0$.

   (b) Part (a) corresponds to finding the roots of what polynomial?

   (c) Find 4 different rearrangements of the root finding problem in (b) which pose the problem as a fixed point problem.

5. (a) Show that $f(x) = x^3 - 10$ has a unique root in $[2,3]$.

   (b) Find the root to machine (PC) single precision by the root-finding method of your choice.

   (c) The iteration function $g(x) = x^2 - 2$ has a fixed point $p = 2$. Will the fixed-point iteration $x_{i+1} \leftarrow g(x_i)$ converge to p from an initial guess of $x_0 = 1.5$? Explain why or why not.

6. The saturation concentration of dissolved oxygen in freshwater (O_{sf}) can be calculated with the equation

$$\ln O_{sf} = -139.34411 + \frac{1.575701 \times 10^5}{T_a} - \frac{6.642308 \times 10^7}{T_a^2} + \frac{1.234800 \times 10^{10}}{T_a^3} - \frac{8.621949 \times 10^{11}}{T_a^4}$$

   where T_a is the absolute temperature (K). According to this equation, saturation decreases with increasing temperature. For typical natural waters in temperate climates, the equation can be used to determine that oxygen concentration ranges from 14.621 ml/L at 0 degree Celsius to 6.413 mg/L at 40 degrees Celsius. Given a value of oxygen concentration, this formula and the bisection method can be used to solve for temperature in Celsius.

   If the initial guesses are set at 0 degrees and 40 degrees, how many bisection iterations would be required to determine temperature to an absolute error of 0.05 degrees Celsius?

7. Suppose $f(x)$ has a root at $x = 1$. Suppose that we use the starting interval $[-1,2]$ for the bisection method and the starting point $x_0 = 2$ for Newton's method.

   (a) Draw a continuous function that meets the above criteria where Newton's method succeeds in finding the root, but the bisection method fails.

   (b) Draw a continuous function that meets the above criteria where the bisection method succeeds in finding the root, but Newton's method fails.

6 Part II: Numerical Linear Algebra

We begin this chapter with a couple of problems that can be expressed as *systems of linear equations* or *linear systems* for short. The first is just a math puzzle; the second has practical significance to the topic of *interpolation*, which we will study next.

EX1: I have 12 coins consisting of nickels, dimes, and quarters, whose total value is $1.45, and whose total weight is 22 grams. If nickels, dimes, and quarters weigh 2,1, and 3 grams, respectively (which they don't really), how many of each coin do I have? (Courtesy of Jim Sochacki).

For starters, let n, d, and q symbolize the unknown numbers of nickels, dimes, and quarters, respectively. To find the three unknown quantities, we need three independent pieces of information that can be expressed as equations. We are given information about 1) the number of coins, 2) the value of the coins, and 3) the weight of the coins, so, if we can turn this data into equation form, there is a good chance of solving the problem.

That there are 12 coins in all can be expressed as

$$n+d+q=12 \tag{82}$$

That the value of the coins is $1.45 leads to

$$0.05n+0.10d+0.25q=1.45 \tag{83}$$

It is not necessary, but it is certainly OK to clear the value equation of decimal fractions by multiplying each side of the equation by 20, in which case

$$n+2d+5q=29 \tag{84}$$

It is an important cornerstone of linear algebra that the information content of an equation is not altered whenever both sides of the equation are multiplied by the same non-zero constant. Finally, that the total weight of the coins is 22 grams yields the equation

$$2n+1d+3q=22 \tag{85}$$

The three equations above are each *linear* in that the unknowns appear only to the first power. For example, we don't see n^2, $1/q$, or $\cos(d)$ in any of the equations. Collectively the three linear equations in three unknowns comprise a linear system, in particular the following system:

$$\begin{aligned} 1n+1d+1q &= 12 \\ 1n+2d+5q &= 29 \\ 2n+1d+3q &= 22 \end{aligned} \tag{86}$$

Perhaps you have had experience solving 2×2 and 3×3 linear systems in high school. Regarding the latter, you may have also had the experience of "going around in circles," while chasing one of the unknowns. A systematic approach to the solution can prevent such circular waste of effort. In particular,

let's use the first equation to eliminate the first variable in both the second and the third equations. We do this by subtracting the first equation from the second and twice the first equation from the third, as follows:

$$\begin{aligned} 1n+1d+1q &= 12 \;\Rightarrow\; 1n+1d+1q=12 \\ \text{[(Eq. 2)-(Eq. 1)]}\; 1n+2d+5q &= 29 \;\Rightarrow\; 0n+1d+4q=17 \\ \text{[(Eq. 3)-2(Eq. 1)]}\; 2n+1d+3q &= 22 \;\Rightarrow\; 0n-1d+1q=-2 \end{aligned}$$

Note that the second and third equations above now involve only the unknowns d and q and thus comprise a 2×2 *subsystem* of the original equation system. Let's now focus on solving the subsystem.

$$\begin{aligned} 1d+4q &= 17 \\ -1d+1q &= -2 \end{aligned}$$

By now adding the subsystem's first equation to its second equation, we eliminate the variable d as follows:

$$\begin{aligned} 1d+4q &= 17 \;\Rightarrow\; 1d+4q=17 \\ \text{[(Eq. 2)+(Eq. 1)]}-1d+1q &= -2 \;\Rightarrow\; 0d+5q=15 \end{aligned}$$

The last equation of the resulting subsystem is now trivial. The solution is $q=3$. Knowing q, we can now work backwards with the first equation immediately above to find d. Specifically, by substituting the value of q into $1d+4q=17$, we obtain $1d+4(3)=17$, or equivalently, $1d=5$. At this stage, we know both q and d. By substituting these known values into the very first equation of the original system, namely $n+d+q=12$, we get $n+5+3=12$, in which case $n=4$. The process of working backwards once the last unknown (q) was obtained is known as *back substitution*.

Our next example will be germaine to the next chapter, which concerns polynomial interpolation.

EX2: Find the parabola that passes through the points (1,0), (2,1), and (3,4). Note that a parabola is the graph of a quadratic function. A general form (of many possible forms) of a quadratic is

$$q(x)=a_0+a_1x+a_2x^2 \tag{87}$$

Our job is to determine the specific coefficients a_0, a_1, and a_2, so that the resulting quadratic "fits" the data. Because there are three unknown coefficients (three *degrees of freedom*), we need three equations so that the problem is well-posed. The three equations follow from the requirements that the graph of $q(x)$ passes through all three points. Specifically, $q(1)=0$, $q(2)=1$, and $q(3)=4$. That is,

$$\begin{aligned} a_0+a_1(1)+a_2(1)^2 &= 0 \\ a_0+a_1(2)+a_2(2)^2 &= 1 \qquad (88) \\ a_0+a_1(3)+a_2(3)^2 &= 4 \end{aligned}$$

(89)

This however is nothing but the following 3×3 linear system for the unknowns a_0, a_1, and a_2.

$$\begin{aligned} 1a_0+1a_1+1a_2 &= 0 \\ 1a_0+2a_1+4a_2 &= 1 \\ 1a_0+3a_1+9a_2 &= 4 \end{aligned}$$

(90)

Let's again find the solution by our structured approach, but this time we will keep the system intact at each step. In the first step, we use the first equation to eliminate the first unknown in the second and third equations. Here is Step 1:

$$\begin{aligned} 1a_0+1a_1+1a_2 &= 0 \Rightarrow 1a_0+1a_1+1a_2=0 \\ \text{[(Eq. 2)-(Eq. 1)]}\ 1a_0+2a_1+4a_2 &= 1 \Rightarrow 0a_0+1a_1+3a_2=1 \\ \text{[(Eq. 3)-(Eq. 1)]}\ 1a_0+3a_1+9a_2 &= 4 \Rightarrow 0a_0+2a_1+8a_2=4 \end{aligned}$$

In Step 2, we use *new* Eq. 2 above to eliminate the second unknown in Eq. 3 above. Here is Step 2.

$$\begin{aligned} 1a_0+1a_1+1a_2 &= 0 \Rightarrow 1a_0+1a_1+1a_2=0 \\ 0a_0+1a_1+3a_2 &= 1 \Rightarrow 0a_0+1a_1+3a_2=1 \\ \text{[(Eq. 3)-2(Eq. 2)]}\ 0a_0+2a_1+8a_2 &= 4 \Rightarrow 0a_0+0a_1+2a_2=2 \end{aligned}$$

Note that the system is now in *upper-triangular form*. That is, if we remove all terms with zero coefficients, the following remains:

$$\begin{aligned} 1a_0+1a_1+1a_2 &= 0 \\ 1a_1+3a_2 &= 1 \\ 2a_2 &= 2 \end{aligned}$$

We now proceed with back susbstitution. The last equation has the trivial solution $a_2 = 1$. Substituting this result into the middle equation yields $1a_1+3(1)=1 \Rightarrow a_1=-2$. Substituting the known values of a_2 and a_1 into the first equation yields $1a_0+1(-2)+1(1)=0 \Rightarrow a_0=1$. Thus the desired quadratic is

$$q(x) = 1-2x+x^2 \tag{91}$$

The reader can verify that $q(x)$ fits the given data.

The sample problems above each represented 3×3 linear systems; that is, three linear equations for three unknowns. However, there is nothing to preclude systems of 4 equations in 4 unknowns (4×4), 11 equations in 11 unknowns (11×11), or for that matter, n equations in n unknowns ($n\times n$). In general, an $n\times n$ system is written

$$\begin{aligned} a_{11}x_1+a_{12}x_2+a_{13}x_3+\ldots+a_{1n}x_n &= b_1 \\ a_{21}x_1+a_{22}x_2+a_{23}x_3+\ldots+a_{2n}x_n &= b_2 \\ a_{31}x_1+a_{32}x_2+a_{33}x_3+\ldots+a_{3n}x_n &= b_3 \\ \vdots \quad\quad \vdots \quad\quad \vdots \quad\quad\quad\quad \vdots \quad\quad \vdots & \\ a_{n1}x_1+a_{n2}x_2+a_{n3}x_3+\ldots+a_{nn}x_n &= b_n \end{aligned} \tag{92}$$

The coefficients a_{ij} ("a eye jay") of the linear system are identified by two indices. The first index, i in this case, is the equation index, also called the *row index*. The second index, j in this case, is called the variable

index or the *column index*. For example, in Eq. 86 above, $a_{23} = 5$ ("a two three"), is the coefficient of the third variable in the second equation. What value does a_{33} have in the same equation?

Our primary objective in this chapter is to develop an efficient and accurate method for solving systems of n linear equations in n unknowns. The algorithm that will emerge from this endeavor is called *Gaussian elimination*, once again in honor of the master himself, Karl Friedrich Gauss. The widespread importance of Gaussian elimination cannot be overstated. Ultimately, most problems in science, engineering, or biomathematics boil down to solving large linear systems, sometimes with tens of thousands of equations. If you don't believe this, check out www.netlib.org, a repository for efficient variations of Gaussian elimination for every conceivable situation. As of the end of 2006, the website had received 400 million hits, probably more even than Paris Hilton's!

6.1 Graphical Interpretation of Systems of Linear Equations

The graph of a linear equation in two independent variables (say, x and y, or x_1 and x_2) is a straight line in the plane; hence the terminology *linear*. Two co-planar straight lines must either intersect or be parallel. If they intersect, they do so at a single point, which represents graphically the solution of a 2×2 linear system, whose first equation defines one line and whose second equation defines the other. Provided the lines are not parallel, the solution is *unique*, as shown in Fig. 15. If the lines are parallel, the plot thickens somewhat. They may be *coincident*, in which case every point on the line satisfies the equation (as there is now only one equation). On the other hand, if they are truly parallel, with no shared points, then there is no solution whatsoever. Under what conditions then are two lines either coincident or parallel? Whenever they have the same slope, of course. To have the same slope, the equations need not have the same coefficients. For example, the lines $l_1 : \ x - 2y = 7$ and $l_2 : \ -3x + 6y = 10$ have the same slope but not the same coefficients nor y intercepts. However, because the coefficients of l_2 are a scalar (-3) multiple of those of l_1, the lines have a common slope. In such cases the coefficient sets of the two equations are *linearly dependent*. That is, one set is a disguised version of the other set. Such systems are said to be *singular*. Singular linear systems either have no solution or an infinity of solutions. Our primary interest lies in non-singular linear systems, which have unique (one and only one) solutions.

What graphical interpretation do we give 3×3 systems of linear equations? A linear function of three independent variables, say $ax + by + cz = d$ where a, b, c, and d are constants, graphs a plane in R^3. Two non-parallel planes intersect in a line. A third mutually non-parallel plane that does not contain the line of intersection of the other two must intersect that line in a single point. This point represents graphically the unique solution of the system. On the other hand, a 3×3 system can fail in one of several ways to have a unique solution. If two or more of the three planes are parallel and share no common points, then there is no solution. If all three planes are coincident, on the other hand, then any point on that plane is a solution, so that the solution space is two-dimensional. Finally, suppose the third plane contains the line of intersection of the first two planes. Then any point on that line is a solution and the solution space is one dimensional. In either of the last two scenarios, the system is singular because the coefficient data are linearly dependent. We will define this concept more precisely later.

For systems larger than 3×3, we lose the ability to interpret the solution in graphical terms. However,

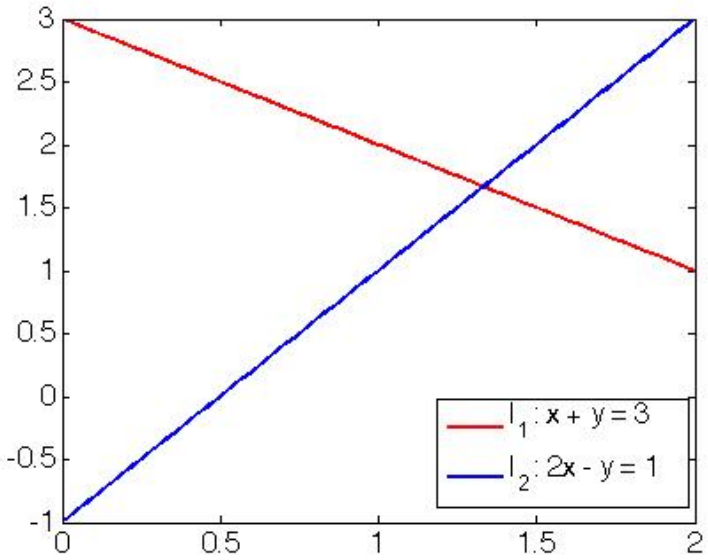


Figure 15: Graphical interpretation of solution of 2×2 linear system.

the algebraic meaning is retained, as are the notions of singularity, linear dependence, and uniqueness, etc.

6.2 Matrices and Vectors: Basic Nomenclature and Properties

When solving an $n \times n$ system of linear equations, as we did previously in two examples with $n = 3$, one eventually realizes that all the algebraic manipulations are being performed only on the coefficients a_{ij} or on the right-hand side data b_i. Thus, for brevity, we could just arrange the coefficients in a table of values and ignore the unknowns x_i for the moment. The coefficient table (or array) for a general $n \times n$ system is shown below.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ . & . & . & & . \\ . & . & . & & . \\ . & . & . & & . \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix}$$

DEF: A rectangular array of real (or complex) numbers enclosed in square brackets is termed a *matrix* (as in the movie). The numbers of a matrix are termed its *elements* or *entries*. Matrices are denoted by uppercase letters, as, for example

$$A = \begin{bmatrix} 3 & 2 & 1 \\ 4 & 5 & 6 \end{bmatrix} \tag{93}$$

Here A is a 2×3 matrix, having two rows and three columns. The number 4 is which element of matrix A? In general, a matrix of m rows and n columns is said to be $m \times n$, which denotes the *size* of the matrix.

Moreover, we will use the nomenclature $M_{m\times n}$ to denote the set of all $m \times n$ matrices. Thus, for example, of matrix A above it can be said $A \in M_{2\times 3}$.

DEF: If $m = n$, then an $m \times n$ matrix is said to be *square*. For example, matrix T below is a square matrix with $m = n = 3$.

$$T = \begin{bmatrix} -2 & 1 & 0 \\ 1 & -2 & 1 \\ 0 & 1 & -2 \end{bmatrix}$$

We are primarily concerned here with the square matrices that result from linear systems of n equations in n unknowns.

DEF: A degenerate matrix for which either $m = 1$ or $n = 1$, but not both, is termed a *vector*. If $m = 1$, we say that the matrix is a *row vector*. In contrast, if $n = 1$, we say that the matrix is a *column vector*. The elements of a vector are more commonly referred to as *components*. Moreover, to distinguish vectors from matrices different symbolism is exploited. Vectors, whether row or column vectors, are distinguished in some texts by boldface lowercase letters or by lowercase letters with a vector sign. We will use the latter. If $m = n = 1$, there is but a single element in the matrix, in which case one refers to the number as a *scalar*.

$$\vec{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \quad ; \quad \vec{d} = [3,2,1]$$

In the example above, $\vec{b}$ is a column 4-vector, and $\vec{d}$ is a row 3-vector.

DEF: Let A be a square matrix with elements a_{ij}. The elements a_{11}, a_{22}, ..., a_{nn} are called the *diagonal elements* of A. For example, the diagonal elements of the 3×3 matrix T above (that is, t_{11}, t_{22}, and t_{33}) each have the value -2.

REMARKS: The diagonal elements of the coefficient matrix will play a "pivotal" role later in the Gaussian elimination algorithm to be developed (pun intended).

DEF: If $A \in M_{m\times n}$, then $A^T \in M_{n\times m}$ ("*A transpose*") is the matrix whose rows are the columns of A and vice versa. That is, $a^T_{ji} = a_{ij}$. For example, for the 2×3 matrix A of Eq. 93,

$$A^T = \begin{bmatrix} 3 & 4 \\ 2 & 5 \\ 1 & 6 \end{bmatrix} \tag{94}$$

A matrix that is equal to its own transpose is said to be *symmetric*. Symmetric matrices must, of course, be square. Matrix T above is a symmetric matrix. Note that transposition of square matrices leaves the diagonal elements, for which $i = j$, unchanged. Regardless of the size of the matrix A, $(A^T)^T = A$. Finally, the transpose of a row vector is a column vector, and vice versa. That is, if $\vec{b} = [-3,5,2]$, then

$$\vec{b}^T = \begin{bmatrix} -3 \\ 5 \\ 2 \end{bmatrix}$$

To close this subsection, we define a *zero matrix* to be a matrix each of whose entries is the number 0. Thus, for example, the 2×3 zero matrix is

$$A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

6.2.1 Matrix Operations

Having defined matrices, we are now in a position to *define* some operations with matrices, among them equality, addition, negation, subtraction, and scalar multiplication.

DEF:

1. Equality: If $A \in M_{m\times n}$, and $B \in M_{m\times n}$, then $A = B$ iff $a_{ij} = b_{ij}$ for all i and j. (Note that matrices can be equated only if they have the same size, in which case equality holds if and only if corresponding elements are equal.)

2. Scalar multiplication: If $A = [a_{ij}]_{m\times n}$, and c is a scalar, then $cA \equiv [ca_{ij}]_{m\times n}$. (Note that, if $c = 0$, the result is the $m \times n$ zero matrix.)

3. Addition: If $A = [a_{ij}]_{m\times n}$, and $B = [b_{ij}]_{m\times n}$, then $A + B \equiv [a_{ij} + b_{ij}]_{m\times n}$. (Note that matrix addition is defined only if the matrices are the same size, in which case the resultant matrix is formed by adding corresponding elements of the constituent matrices.)

4. Opposition (negation): If $A = [a_{ij}]_{m\times n}$, then $-A = cA$ with $c = -1$. (Note that this has the effect of reversing the sign of each element of A.)

5. Subtraction: If A and B are of commensurate size, then $A - B \equiv A + (-B)$.

6.2.2 Properties of Matrices

Suppose the p and q are scalars, and that A, B, and C are matrices of commensurate sizes. Then the following properties are readily proved from the definitions of the matrix operations presented immediately above:

PROPERTIES:

1. $B + A = A + B$ (Commutativity under addition)

2. $[0] + A = A + [0] = A$ ([0], the additive identity)

3. $A - A = A + (-A) = [0]$ ($-A$, the additive inverse)

4. $(A + B) + C = A + (B + C)$ (Associativity of addition)

5. $(p+q)A = pA + qA$ (First distributive property of scalar multiplication)

6. $p(A+B) = pA + pB$ (Second distributive property of scalar multiplication)

7. $p(qA) = (pq)A$ (Associative property of scalar multiplication)

At this point, you may be tempted to yawn, thinking of course that matrices behave just like numbers and that listing the operations and properties above is a complete waste of time. Not so fast. There is something missing from the list of operations. Can you "find" it? If you said matrix multiplication, you are keenly observant. We have not yet defined the product AB of two matrices A and B. It turns out that AB is not even defined for most matrices, and even when it is defined, it is rare that BA is also defined and rarer still that $AB = BA$. So, matrices DO NOT behave just like numbers.

6.3 Matrix Multiplication

If you have no previous experience with linear algebra, the definition of matrix multiplication to follow will seem entirely bizarre. Only after the fact will it begin to make some sense. So bare with us for the moment, trusting that this is leading ultimately to something useful.

6.3.1 The Dot (Inner) Product of Two Vectors

To understand matrix multiplication, it is first helpful to define the *dot product* (or *inner product*), which applies only to vectors.

DEF: Suppose $\vec{x} = [x_1, x_2, ..., x_n]^T$ and $\vec{y} = [y_1, y_2, ..., y_n]^T$ are two column n-vectors. Their *dot product*, denoted $\vec{x} \cdot \vec{y}$ ("x dot y") is defined

$$\vec{x} \cdot \vec{y} = \vec{x}^T \vec{y} = x_1 y_1 + x_2 y_2 + ... + x_n y_n = \sum_{i=1}^{n} x_i y_i \tag{95}$$

In words, the dot product of two vectors that have the same length is formed by multiplying corresponding components and summing the products. Note that the dot product returns a scalar.

EX: If $\vec{x} = [3, 2, 1]^T$ and $\vec{y} = [4, -2, 1]^T$, then $\vec{x} \cdot \vec{y} = (3)(4) + (2)(-2) + (1)(1) = 9$.

On a machine, summations are accomplished by a loop operation; hence, a dot product of two vectors requires little more than a single loop, as shown in Algorithm 6.1.

ALGORITHM 6.1: Dot Product

input size n
input n components x_i, $i = 1, 2, ..., n$ *(fill a 1D array)*
input n components y_i, $i = 1, 2, ..., n$ *(fill a 1D array)*

$t \leftarrow 0$ *(initialize temporary storage register)*

repeat for $i = 1, 2, ..., n$

$\quad t \leftarrow t + x_i y_i$

output t *(which contains value of* $\vec{x} \cdot \vec{y}$*)*

What can we say about the operation count of ALG 6.1? There is a single loop, which is executed n times. Each pass through the loop requires one multiplication and one addition, or two floating-point operations. Thus, the operation count is exactly $2n$.

Note that the dot product is also called the *inner product*. A common application of a dot product is in the computation of the *Euclidean length* (or *norm*) of a vector.

DEF: The Euclidean length (norm) of a column vector $\vec{x}$ is denoted $||\vec{x}||$ and is defined

$$||\vec{x}|| = (\vec{x} \cdot \vec{x})^{1/2} \tag{96}$$

EX: Let $\vec{x} = [3, -2, 1]^T$. Then $||\vec{x}|| = [(3)(3) + (-2)(-2) + (1)(1)]^{1/2} = \sqrt{14}$. In R^2 or R^3, $||\vec{x}||$ is the physical length of the vector whose tail resides at the origin and whose head resides at the point whose coordinates are the components of the vector. Thus, in the example, $||\vec{x}||$ is the physical length of the vector whose tail is at (0,0,0) and whose head is at (3,-2,1) in Cartesian 3-space. For $n > 3$, the physical meaning of the norm is lost, but the mathematical definition is retained.

6.3.2 Matrix Products

We are now in a position to define the product of two matrices.

DEF: If $A \in M_{m \times p}$ and $B \in M_{q \times n}$, then $C = AB$ is defined if and only if $p = q$, in which case

$$\begin{aligned} C &= [c_{ij}]_{m \times n} \text{ with} \\ c_{ij} &= \sum_{k=1}^{p} a_{ik} b_{kj} \\ &= a_{i1} b_{1j} + a_{i2} b_{2j} + a_{i3} b_{3j} + ... + a_{ip} b_{pj} \end{aligned} \tag{97}$$

How can we possibly give meaning to such a strange definition? First, it helps to recognize that the definition above contains a hidden dot product. Note that in the computation of coefficient c_{ij} of the resultant C matrix, both indices i and j remain fixed; the summation is over the index k. Thus a_{ik} with i fixed refers to the ith row of the matrix A. Similarly, b_{kj} with j fixed refers to the jth column of the matrix B. Hence, the coefficient c_{ij} of the C matrix is formed by the dot product of the ith row of matrix A and the (transpose of the) jth column of matrix B, as shown in Fig. 16.

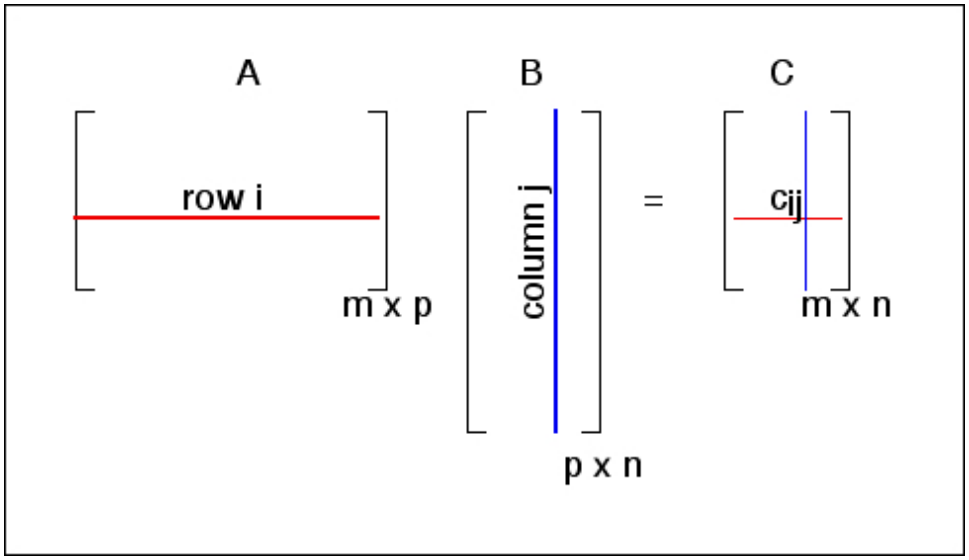


Figure 16: Matrix multiplication via dot products.

EX: Let

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}_{2\times 2} \quad \text{and} \quad B = \begin{bmatrix} 1 & 0 & 1 \\ 1 & -2 & 1 \end{bmatrix}_{2\times 3}$$

Then

$$AB \in M_{2\times 3} \quad \text{with} \quad AB = \begin{bmatrix} 3 & -4 & 3 \\ 7 & -8 & 7 \end{bmatrix}$$

Note however that BA is not even defined because the number of columns (3) of matrix B does not match with the number of rows (2) of matrix A! However, even though BA is undefined, B^TA is well defined as shown below.

$$B^TA = \begin{bmatrix} 1 & 1 \\ 0 & -2 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 4 & 6 \\ -6 & -8 \\ 4 & 6 \end{bmatrix}_{3\times 2}$$

Thus, if AB is defined, BA may or may not be defined.

It is worth mentioning that the definition of matrix multiplication above also applies to vectors. For example, the product of a row vector, which can be viewed as a matrix of size $1 \times n$, and a column vector of the same length, which can be viewed as a matrix of size $n \times 1$, is a 1×1 matrix, which is just a scalar. Thus if $\vec{x}$ and $\vec{y}$ are two column n vectors, $\vec{x}^T\vec{y}$ is their dot product. In this case, $\vec{y}\vec{x}^T$ is also defined. However, the product of an $n \times 1$ matrix with a $1 \times n$ matrix is an $n \times n$ matrix, not a scalar! This is termed the *outer or tensor product* of two vectors. For example, if $\vec{x} = [1,2,3]^T$ and $\vec{y} = [1,0,-1]^T$, then

$$\vec{x}\vec{y}^T = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 3 & 0 & -3 \end{bmatrix}_{3\times 3}$$

All this goes to show that it is important to keep the following ever in mind when dealing with matrix multiplication: *matrix multiplication is not, in general, commutative*. Matrices DO NOT behave like real numbers.

Matrix multiplication is a computationally intensive process. If matrix A has m rows and matrix B has n columns, the resultant matrix C has mn coefficients, each of which is computed by a dot product of length p that requires $2p$ operations. Thus the operation count for matrix multiplication is $2mpn$. If matrices A and B are square and each $n \times n$, then $2n^3$ operations are required to form their product. It is not unusual for n to be 1000 or larger for problems of engineering or scientific interest. Thus, to multiply two matrices, each of size 1000×1000 requires some two *billion* operations. Indeed a primary impetus for the development of fast computers was the necessity of manipulating large matrices in reasonable clock time.

Before closing this subsection, let's define a very special square matrix, the *identity matrix*.

DEF: The $n \times n$ identity matrix, denoted I_n, has elements defined as follows:

$$\begin{aligned} I_n &= [\delta_{ij}]_{n\times n} \text{ where} \\ \delta_{ij} &\equiv \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \end{aligned} \tag{98}$$

The symbol δ_{ij} is known as the "kronecker delta," which can take on only one of two values: 0 or 1. If all this mathematical formalism seems a bit over the top to you, then just remember that an identity matrix is a square matrix all of whose entries are 0, with the exceptions of those lying along the diagonal, whose values are each 1. Here, for example, is the 3×3 identify matrix:

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}_{3\times 3}$$

Note that the kronecker delta is one if and only if the row and column indices are the same, which is true only for diagonal elements. All other entries are zero.

6.3.3 Properties of Matrix Multiplication

Suppose that A, B, C, and D are matrices, $I = I_n$ denotes an identity matrix, $[0]$ denotes a zero matrix, and c is a scalar. From Eq. 97, which defines matrix multiplication, Eq. 98, which defines the identity matrix, and the definition of the zero matrix, follow all the following properties, assuming that the size of the matrices is such that their products are well defined:

1. (*) $AB \neq BA$ (Non-commutativity of matrix multiplication)
2. (*) $AB = [0] \not\Rightarrow A = [0]$ or $B = [0]$
3. $AI = IA = A$ (I_n is the multiplicative identity for matrices)
4. $cAB = (cA)B = A(cB)$ (Associativity of scalar multiplication of matrices)
5. $ABC = (AB)C = A(BC)$ (Associativity of matrix multiplication)
6. $A(B+C) = AB + AC$ (1st distributive property of matrix multiplication)

7. $(A+B)C = AC+BC$ (2nd distributive property of matrix multiplication)

WARNING: Although matrix multiplication is distributive and associative, the order of multiplication must be preserved because of non-commutativity. Because the first two properties differ from the situation with real numbers, we have flagged these properties each with an asterisk.

At this point, you may still be wondering how the concept of matrix multiplication fits into the grand scheme of things and/or why the definition of matrix multiplication seems so strange. OK, it is time to put all our cards on the table. Consider the $n \times n$ linear system of equations expressed in Eq. 92. Let $\vec{x} = [x_1, x_2, ..., x_n]^T$ be the column n-vector of unknowns, let $\vec{b} = [b_1, b_2, ..., b_n]^T$ be the column n-vector of right-hand-side values, and let A be the $n \times n$ matrix of coefficients given in Eq. 6.2. With these definitions, plus the definition of matrix multiplication, the full $n \times n$ system expressed in Eq. 92 can be concisely expressed as

$$A\vec{x} = \vec{b} \tag{99}$$

That this simple matrix-vector expression encodes an entire system of linear equations may temporarily escape you. So let's spell it out in detail. On the left-hand side of Eq. 99 is the product of $n \times n$ matrix A with a degenerate $n \times 1$ matrix known as vector $\vec{x}$, the result of which is a column n-vector. The right-hand side of Eq. 99 is also a column n-vector. As these two vectors are equal, their respective components must be equal. So let's look at the ith component of each. On the left-hand side, the ith component is given by the dot product of the (transpose of the) ith row of the coefficient matrix A with the column vector $\vec{x}$. That dot product is the scalar $a_{i1}x_1 + a_{i2}x_2 + ...a_{in}x_n$. Setting this equal to b_i, the corresponding component from the right-hand side, we get $a_{i1}x_1 + a_{i2}x_2 + ...a_{in}x_n = b_i$, which is the ith equation of the original linear system. Keeping in mind that i remains arbitrary, we could do this for $i = 1, 2, ..., n$ so as to recover all n linear equations of our original system from Eq. 99.

The remainder of this chapter concerns primarily the development of efficient and accurate algorithms for solving systems of linear equations expressed in the matrix-vector form of Eq. 99, which is shorthand for the following:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & ... & a_{1n} \\ a_{21} & a_{22} & a_{23} & ... & a_{2n} \\ . & . & . & & . \\ . & . & . & & . \\ . & . & . & & . \\ a_{n1} & a_{n2} & a_{n3} & ... & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ . \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ . \\ . \\ . \\ b_n \end{bmatrix}$$

6.4 Gaussian Elimination

The standard algorithm for solving systems of linear equations of the form $A\vec{x} = \vec{b}$ for which the coefficient matrix A is *dense* is called *Gaussian elimination*. A dense matrix, by the way, is one whose elements are (nearly) all non-zero. (The identity matrix, in contrast, is *sparse*.) The great edifice of Gaussian

elimination is predicated upon three basic operations known as the *elementary row operations.* Moreover, Gaussian elimination is comprised of two separate algorithms implemented in succession: *forward elimination* followed by *back substitution*. The latter is the simpler of the two, so we will describe these algorithms in *reverse* order. But first, let's define the elementary row operations.

6.4.1 Elementary Row Operations

In short, Gaussian elimination is a process of converting a linear system of equations into an equivalent system, whose solution is much easier to obtain than that of the original system.

DEF: *Equivalent linear systems* are two linear systems of the same size that have the same solution. For example, the two 2×2 systems below are equivalent, because both have the solution $(x_1, x_2) = (1, 2)$, which can readily be verified:

$$\begin{aligned} x_1 + x_2 &= 3 \\ x_1 - x_2 &= -1 \end{aligned} \qquad\qquad \begin{aligned} x_1 + x_2 &= 3 \\ x_2 &= 2 \end{aligned}$$

Given the choice of solving the first or the second of the two equivalent linear systems, which would you choose? The second, of course, because it is easier to solve.

Equivalent linear systems are related by the following elementary row operations (ERO):

1. Interchange of any two equations (rows).
2. Multiplication of an equation (row) by a non-zero scalar (real number).
3. Addition of any two equations (rows).

The first ERO recognizes that swapping the order of the equations in a system of equations does not change the result. The second follows from the fact that equals multiplied by equals remain equal. Similarly, the third recognizes that when equals are added to equals, the two sides of an equation remain equal.

The EROs apply to equation systems. However, as observed previously, when solving systems of linear equations, the algebraic manipulations involve only the coefficients of the unknowns and the right-hand-side values. As a consequence, we can solve a linear system by dealing only with its *augmented matrix* form, which is comprised of the coefficient matrix to which an additional column containing the right-hand-side values is appended. The augmented matrix of a general $n \times n$ system looks like the following:

$$\left[\begin{array}{ccccc|c} a_{11} & a_{12} & a_{13} & \ldots & a_{1n} & b_1 \\ a_{21} & a_{22} & a_{23} & \ldots & a_{2n} & b_2 \\ . & . & . & & . & . \\ . & . & . & & . & . \\ . & . & . & & . & . \\ a_{n1} & a_{n2} & a_{n3} & \ldots & a_{nn} & b_n \end{array}\right]$$

The EROs get their name because they apply expressly to the rows of the augmented matrix above. As an example of the application of the EROs to an augmented matrix, let's return to the second example (EX2) at the very beginning of the chapter, which asked us to find the quadratic polynomial that passes through three distinct points. Expressed in augmented matrix form, Eq. 88 becomes

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 0 \\ 1 & 2 & 4 & 1 \\ 1 & 3 & 9 & 4 \end{array}\right]$$

Now consider the following sequence of EROs:

$$\begin{array}{l} (R2-R1) \\ (R3-R1) \end{array} \left[\begin{array}{ccc|c} 1 & 1 & 1 & 0 \\ 1 & 2 & 4 & 1 \\ 1 & 3 & 9 & 4 \end{array}\right] \rightarrow \quad (R3-2R1) \left[\begin{array}{ccc|c} 1 & 1 & 1 & 0 \\ 0 & 1 & 3 & 1 \\ 0 & 2 & 8 & 4 \end{array}\right] \rightarrow \left[\begin{array}{ccc|c} 1 & 1 & 1 & 0 \\ 0 & 1 & 3 & 1 \\ 0 & 0 & 2 & 2 \end{array}\right]$$

Note that the *linear combination* of two rows, for example $(R3-2R1)$, is actually the combination of EROs (2) and (3). Note also that the resulting augmented matrix on the far right above is in *upper-triangular* or *row echelon* form, in that all coefficients below the diagonal are zero. Because the final and the initial augmented matrix forms are related only by EROs, their respective linear systems are equivalent. Hence, the solution to the system represented at the far right is mathematically identical to the solution of the original system. But because of the row echelon form, the system at the far right is much easier to solve than the original system. If we restore the unknowns to the system at the far right, we obtain

$$\begin{aligned} 1a_0 + 1a_1 + 1a_2 &= 0 \\ 1a_1 + 3a_2 &= 1 \\ 2a_2 &= 2 \end{aligned}$$

Working our way *backwards* from the last equation to the first, we obtain $a_2 = 1$, $a_1 = -2$, and finally, $a_0 = 1$. This process of working backwards from row echelon form is known as *back substitution*.

6.4.2 Back Substitution

Suppose $A\vec{x} = \vec{b}$ and $U\vec{x} = \vec{d}$ are equivalent $n \times n$ linear systems, where matrix U is *upper triangular*. That is, the latter system has the form

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ 0 & u_{22} & u_{23} & \dots & u_{2n} \\ . & . & . & & . \\ . & . & . & & . \\ . & . & . & & . \\ 0 & 0 & 0 & \dots & u_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ . \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ . \\ . \\ . \\ d_n \end{bmatrix}$$

Given the choice to solve the original system or the upper-triangular system, we would be crazy to solve the former because the latter is much easier system to solve. Note that all coefficients of U below the diagonal are zero. So, let's pretend that the Linear Algebra Fairy has waved her magic wand and converted our original system to the form $U\vec{x} = \vec{d}$. Let's now find an algorithm for back substitution. We'll develop this for the 4×4 case and then generalize to $n \times n$. Here's an upper-triangular system of size 4×4 with the variables restored:

$$\begin{aligned} u_{11}x_1 + u_{12}x_2 + u_{13}x_3 + u_{14}x_4 &= d_1 \\ u_{22}x_2 + u_{23}x_3 + u_{24}x_4 &= d_2 \\ u_{33}x_3 + u_{34}x_4 &= d_3 \\ u_{44}x_4 &= d_4 \end{aligned}$$

Let's work our way backwards. The last equation is trivial. From EQ4, we obtain

$$u_{44}x_4 = d_4 \;\;\Rightarrow\;\; x_4 = \frac{d_4}{u_{44}} \tag{100}$$

The third equation (EQ3) now becomes $u_{33}x_3 + u_{34}\underline{x_4} = d_3$, where we have underlined x_4 to indicated that its value is now known, in which case the second term on the left side can migrate to the right to give

$$u_{33}x_3 = d_3 - u_{34}x_4 \;\;\Rightarrow\;\; x_3 = \frac{d_3 - u_{34}x_4}{u_{33}} \tag{101}$$

The last two variables are now known, in which case EQ2 becomes $u_{22}x_2 + u_{23}\underline{x_3} + u_{24}\underline{x_4} = d_2$, from which we conclude

$$u_{22}x_2 = d_2 - u_{23}x_3 - u_{24}x_4 \;\;\Rightarrow\;\; x_2 = \frac{d_2 - u_{23}x_3 - u_{24}x_4}{u_{22}} \tag{102}$$

Finally, we arrive at EQ1, with known values for the last three variables. That is, $u_{11}x_1 + u_{12}\underline{x_2} + u_{13}\underline{x_3} + u_{14}\underline{x_4} = d_1$, from which we infer

$$u_{11}x_1 = d_1 - u_{12}x_2 - u_{13}x_3 - u_{14}x_4 \;\;\Rightarrow\;\; x_1 = \frac{d_1 - u_{12}x_2 - u_{13}x_3 - u_{14}x_4}{u_{11}} \tag{103}$$

Let's now generalize to $n \times n$ systems and construct an algorithm. First, it is clear that to run backwards, backward substitution must incorporate a decrementing loop. Moreover, we must first "prime the pump;" that is, the last of the unknowns must be solved for first in order that the backward substitution process can commence. Thus, x_n must be obtained *outside* the main loop. Finally, note that the number of terms

subtracted from d_i to solve for x_i grows as the algorithm proceeds, and that the first term subtracted contains the variable previously just solved for, namely x_{i+1}. Here then is a bare-bones back-substitution algorithm:

$x_n \leftarrow d_n/u_{nn}$ (outside the loop)
repeat for $i = n-1, n-2, ..., 1$ (decrementing loop)
| $x_i \leftarrow (d_i - \sum_{k=i+1}^{n} u_{ik}x_k)/u_{ii}$
| ___

We now recognize that the summation inside the loop above is itself a loop in disguise. If we "unroll" the summation, the back substitution process is revealed to harbor a loop within a loop, or as we say in programming parlance, a pair of *nested loops*. Note that the length of the inner loop depends upon the index of the outer loop. Here then is the back substition algorithm in its full glory.

ALGORITHM 6.2: Back Substitution for Gaussian Elimination

input (or pass) order n, coefficients $[u_{ij}]_{n\times n}$, *n-vector* $\vec{d}$
$x_n \leftarrow d_n/u_{nn}$ *(outside the loop)*

for $i = n-1, n-2, ..., 1$ *(descending order)*
|
| $s \leftarrow d_i$ *(initialize accumulator to right-hand-side value*
|
| *for* $k = i+1, i+2, ..., n$ *(ascending order)*
| |
| | $s \leftarrow s - u_{ik}x_k$
| | ___
|
| $x_i \leftarrow s/u_{ii}$
| ___

return $\vec{x}$

How many operations are required to complete back substitution for an $n \times n$ triangular system? Here, a table of operations might help. Table 7 reflects the fact that each time the inner loop is executed a multiplication ($\times$) and a subtraction ($-$) are performed. However, the length of the inner loop depends upon the index of the outer loop. In addition, each time the outer loop is executed, a division is performed. From the table, we surmise that the total operation count for back substitution is $S = 1+3+5+...+(2n-1)$. Using the trick of Gauss once again, we write the sum backwards to obtain $S = (2n-1)+(2n-3)+(2n-5)+...+1$. Adding columnwise the two sums we get $2S = 2n+2n+2n+...+2n = n(2n) = 2n^2$. Thus, $S = n^2$; that is, back substitution for an $n \times n$ triangular system requires exactly n^2 operations.

What can go wrong with back substitution? Clearly, if $u_{ii} = 0$ for one or more values of i, then there will be a run-time "divide by zero" error. How is this possibility to be interpreted?

DEF: Let $U = [u_{ij}]_{n\times n}$ be a square upper-triangular matrix. Then the *determinant of* U, denoted $\det(U)$,

Equation	Operations
n	(1) $\div$
$n-1$	(3) $\times - \div$
$n-2$	(5) $\times - \times - \div$
$n-3$	(7) $\times - \times - \times - \div$
.	.
.	.
.	.
1	(2n-1)

Table 7: Operations in back substitution.

is defined

$$\det(U) = u_{11}u_{22}...u_{nn} = \Pi_{i=1}^{n} u_{ii} \tag{104}$$

Here Π is a mathematical symbol (the Greek uppercase "P") denoting *product* analogous to the symbol $\sum$ (the Greek uppercase "S") for *sum*. A matrix for which $\det(U) = 0$ is said to be *singular*.

Note that if $u_{ii} = 0$ for *any* index i, then $\det(U) = 0$, and U is a singular matrix. If U is singular, then the system $U\vec{x} = \vec{d}$ does *not* have a unique solution.

HW1: Consider $U\vec{x} = \vec{d}$ for

$$U = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{bmatrix} \qquad \vec{d} = \begin{bmatrix} 6 \\ 8 \\ 12 \end{bmatrix}$$

Compute $\det(U)$ and determine whether or not U is singular. If U is non-singular, find the solution of the system $U\vec{x} = \vec{d}$.

6.4.3 Forward Elimination

In the previous subsection, we presumed that some good fairy had magically transformed our linear system $A\vec{x} = \vec{b}$ into an equivalent upper-triangular system $U\vec{x} = \vec{d}$, which we could easily solve in n^2 operations by back substitution. Life is rarely so kind. Alas, there is no Linear Algebra Fairy, and we ourselves must perform this transformation, which is accomplished by a process known as *forward elimination*. Gaussian elimination therefore is forward elimination followed by back substitution. Symbolically we might say $GE = FE + BS$, where, here, BS conveys "back substitution" rather than its usual scatological meaning. Here then is the goal of forward elimination:

$$\left[\begin{array}{ccccc|c} a_{11} & a_{12} & a_{13} & ... & a_{1n} & b_1 \\ a_{21} & a_{22} & a_{23} & ... & a_{2n} & b_2 \\ . & . & . & & . & . \\ . & . & . & & . & . \\ . & . & . & & . & . \\ a_{n1} & a_{n2} & a_{n3} & ... & a_{nn} & b_n \end{array}\right] \quad \begin{array}{c} \\ \\ \\ \rightarrow \\ (FE) \\ \\ \\ \end{array} \quad \left[\begin{array}{ccccc|c} u_{11} & u_{12} & u_{13} & ... & u_{1n} & d_1 \\ 0 & u_{22} & u_{23} & ... & u_{2n} & d_2 \\ . & . & . & & . & . \\ . & . & . & & . & . \\ . & . & . & & . & . \\ 0 & 0 & 0 & ... & u_{nn} & d_n \end{array}\right]$$

But the devil is in the details. Let's develop FE by examining its major steps in sequence along with the algorithm for that step.

STEP 1: Zero the coefficients below the diagonal in column 1

In the first step of FE, EROs involving linear combinations of the first row with each other row are used to eliminate (zero) all coefficients below the diagonal in the first column. If you have printed a color version of this document, or if you are accessing it on line, you will find that the color blue has been used to highlight all the coefficients that are to be zeroed in the first step. The color red has been used to identify the *pivot element*. In general, the pivot element of STEP k of FE is a_{kk}. Pivot elements live along the diagonal of the matrix. Notice that the pivot element is the divisor for each linear combination of rows in STEP 1 below. Note also that, for each i, $a_{i1} - (a_{i1}/a_{11})a_{11} = 0$. That is, the multiplier $(t \leftarrow a_{i1}/a_{11})$ of row 1 (R1) is chosen expressly for the purpose of eliminating (zeroing) the element a_{i1}.

$$
\begin{array}{r}
\\
(R2 - \frac{a_{21}}{a_{11}}R1) \\
(R3 - \frac{a_{31}}{a_{11}}R1) \\
. \\
. \\
. \\
(Rn - \frac{a_{n1}}{a_{11}}R1)
\end{array}
\left[\begin{array}{ccccc|c}
a_{11} & a_{12} & a_{13} & \ldots & a_{1n} & b_1 \\
a_{21} & a_{22} & a_{23} & \ldots & a_{2n} & b_2 \\
a_{31} & a_{32} & a_{33} & \ldots & a_{3n} & b_3 \\
. & . & . & & . & . \\
. & . & . & & . & . \\
. & . & . & & . & . \\
a_{n1} & a_{n2} & a_{n3} & \ldots & a_{nn} & b_n
\end{array}\right]
\rightarrow
\left[\begin{array}{ccccc|c}
a_{11} & a_{12} & a_{13} & \ldots & a_{1n} & b_1 \\
0 & a'_{22} & a'_{23} & \ldots & a'_{2n} & b'_2 \\
0 & a'_{32} & a'_{33} & \ldots & a'_{3n} & b'_3 \\
. & . & . & & . & . \\
. & . & . & & . & . \\
. & . & . & & . & . \\
0 & a'_{n2} & a'_{n3} & & a'_{nn} & b'_n
\end{array}\right]
$$

The linear combination of rows necessary to eliminate (zero) the coefficient a_{i1} will likely modify all the other elements of row i as well as the right-hand-side value b_i. Here primes denote matrix elements or vector components that have been modified from their original values. The algorithm for the first step of FE is given below. Because the modified coefficients simply replace the original coefficients in the same storage registers, there is no need to retain the primes in the algorithm itself. Note also that, although the purpose of the first step is to zero elements that fall below the diagonal in column one, it is actually unnecessary to set $a_{i1} \leftarrow 0$ in the algorithm. This is because a_{i1} will never again be used in solving the linear system.

ALG: FE Step 1

```
for i = 2,3,...,n (row index)
|    t ← a_i1/a_11
|    (a_i1 ← 0) (extraneous)
|    for j = 2,3,...,n (column index)
|    |    a_ij ← a_ij − t a_1j
|    |__
|    b_i ← b_i − t b_1
|__
```

STEP 2: Zero the coefficients below the diagonal in column 2

STEP 2 of FE is similar to STEP 1, except that EROs involving linear combinations of the second

row (R2) with each subsequent row are used to eliminate (zero) all coefficients below the diagonal in the second column. In STEP 2, a_{22} is the pivot element, and the divisor in each multiplier of R2.

$$
\begin{array}{r}
\\
(R3-\frac{a_{32}}{a_{22}}R2)\\
.\\
.\\
.\\
(Rn-\frac{a_{n2}}{a_{22}}R2)
\end{array}
\left[\begin{array}{ccccc|c}
a_{11} & a_{12} & a_{13} & \ldots & a_{1n} & b_1\\
0 & a'_{22} & a'_{23} & \ldots & a'_{2n} & b'_2\\
0 & a'_{32} & a'_{33} & \ldots & a'_{3n} & b'_3\\
. & . & . & & . & .\\
. & . & . & & . & .\\
. & . & . & & . & .\\
0 & a'_{n2} & a'_{n3} & \ldots & a'_{nn} & b'_n
\end{array}\right]
\rightarrow
\left[\begin{array}{ccccc|c}
a_{11} & a_{12} & a_{13} & \ldots & a_{1n} & b_1\\
0 & a'_{22} & a'_{23} & \ldots & a'_{2n} & b'_2\\
0 & 0 & a''_{33} & \ldots & a''_{3n} & b''_3\\
. & . & . & & . & .\\
. & . & . & & . & .\\
. & . & . & & . & .\\
0 & 0 & a''_{n3} & & a''_{nn} & b''_n
\end{array}\right]
$$

Here double primes denote the matrix elements or vector components that have been *twice* modified from their original values. For clarity in the remaining steps, we will use double primes to denote values that have been modified *at least twice*.

ALG: FE Step 2

for $i = 3,4,\ldots,n$ *(row index)*
  $t \leftarrow a_{i2}/a_{22}$
  $(a_{i2} \leftarrow 0)$ *(extraneous)*
  for $j = 3,4,\ldots,n$ *(column index)*
    $a_{ij} \leftarrow a_{ij} - t a_{2j}$
  ___
  $b_i \leftarrow b_i - t b_2$
___

Forward elimination proceeds in similar fashion, focusing on the elements below the diagonal in column k in STEP k. We now fast forward to the last step of FE, namely STEP $n-1$.

STEP $n-1$**:** Zero the coefficients below the diagonal in column $n-1$

In STEP $n-1$ of FE, a single ERO involving a linear combination of R$n-1$ with Rn is used to eliminate the remaining non-zero coefficient below the diagonal in column $n-1$. In the last step, $a_{n-1,n-1}$ is the pivot element.

$$
\begin{array}{r}
\\
\\
\\
\\
\\
(Rn-\frac{a_{n,n-1}}{a_{n-1,n-1}}Rn-1)
\end{array}
\left[\begin{array}{ccccc|c}
a_{11} & a_{12} & \ldots & a_{1,n-1} & a_{1n} & b_1\\
0 & a'_{22} & \ldots & a'_{2,n-1} & a'_{2n} & b'_2\\
. & . & & & . & .\\
. & . & & & . & .\\
. & . & & & . & .\\
0 & 0 & \ldots & a''_{n-1,n-1} & a''_{n-1,n} & b''_{n-1}\\
0 & 0 & \ldots & a''_{n,n-1} & a''_{nn} & b''_n
\end{array}\right]
\rightarrow
\left[\begin{array}{ccccc|c}
a_{11} & a_{12} & \ldots & a_{1,n-1} & a_{1n} & b_1\\
0 & a'_{22} & \ldots & a'_{2,n-1} & a'_{2n} & b'_2\\
. & . & & & . & .\\
. & . & & & . & .\\
. & . & & & . & .\\
0 & 0 & \ldots & a''_{n-1,n-1} & a''_{n-1,n} & b''_{n-1}\\
0 & 0 & \ldots & 0 & a''_{nn} & b''_n
\end{array}\right]
$$

ALG: FE Step $n-1$

```
for i = n (row index)
|     t ← a_{i,n-1}/a_{n-1,n-1}
|     (a_{i,n-1} ← 0) (extraneous)
|     for j = n (column index)
|     |     a_{ij} ← a_{ij} − t a_{n-1,j}
|     |___
|     b_i ← b_i − t b_{n-1}
|___
```

Note that in each successive step of FE, the lengths of the loops in the algorithm for that step diminish by one. Consequently, in the last step, the loops are degenerate. That is, each loop is executed only once, the outer loop for $i = n$ only and the inner loop for $j = n$ only.

Hopefully it is now clear to the reader that the FE algorithm, when fully assembled, consists of *triply nested* loops. To assemble the final algorithm, we simply implement an outer loop with index k whose job is to perform STEP k. Hence, k will run from 1 to $n-1$. Inside the outer loop is the algorithm for STEP k, whose indices i and j each run from $k+1$ to n. Without further ado, here is the algorithm that we have been working toward for some twenty pages.

ALGORITHM 6.3: Forward Elimination (FE) without Pivoting

```
input (pass) system size n
input (pass) matrix [a_{ij}]_{n×n}
input (pass) right-hand-side vector b⃗ = [b_i]_{n×1}

for k = 1,2,...,n−1 (k denotes column to be zeroed)
|
|     for i = k+1,k+2,...,n (row index)
|     |
|     |     t ← a_{ik}/a_{kk}
|     |     (a_{ik} ← 0) (extraneous)
|     |
|     |     for j = k+1,k+2,...,n (column index)
|     |     |
|     |     |     a_{ij} ← a_{ij} − t a_{kj}
|     |     |___
|     |
|     |     b_i ← b_i − t b_k
|     |___
|___

return matrix A (containing U) and vector b⃗ (now d⃗)
```

Upon return to the main program, the matrix A contains U in its upper-triangular components, and $\vec{b}$, having been modified, is now $\vec{d}$ of the system $U\vec{x} = \vec{d}$. These values are then passed to the backward

substitution (BS) algorithm, which ignores the extraneous lower-triangular components of U.

The time has come to put it all together with an example.

EX: a) Show that the following two linear systems are equivalent, and then b) find the unique solution.

$$\begin{aligned} 2x_1+4x_2-4x_3+0x_4 &= 12 \\ x_1+5x_2-5x_3-3x_4 &= 18 \\ 2x_1+3x_2+x_3+3x_4 &= 8 \\ x_1+4x_2-2x_3+2x_4 &= 8 \end{aligned} \qquad \begin{aligned} 2x_1+4x_2-4x_3+0x_4 &= 12 \\ 3x_2-3x_3-3x_4 &= 12 \\ 4x_3+2x_4 &= 0 \\ 3x_4 &= -6 \end{aligned}$$

The augmented matrix for the original system is

$$\left[\begin{array}{cccc|c} 2 & 4 & -4 & 0 & 12 \\ 1 & 5 & -5 & -3 & 18 \\ 2 & 3 & 1 & 3 & 8 \\ 1 & 4 & -2 & 2 & 8 \end{array}\right]$$

We now perform elementary row operations to drive the original system toward upper-triangular form. This is the forward elimination (FE) component of Gaussian elimination.

$$\begin{array}{l} R2-\frac{1}{2}R1 \\ R3-\frac{2}{2}R1 \\ R4-\frac{1}{2}R1 \end{array} \left[\begin{array}{cccc|c} 2 & 4 & -4 & 0 & 12 \\ 1 & 5 & -5 & -3 & 18 \\ 2 & 3 & 1 & 3 & 8 \\ 1 & 4 & -2 & 2 & 8 \end{array}\right] \rightarrow \begin{array}{l} \\ R3-\frac{-1}{3}R2 \\ R4-\frac{2}{3}R2 \end{array} \left[\begin{array}{cccc|c} 2 & 4 & -4 & 0 & 12 \\ 0 & 3 & -3 & -3 & 12 \\ 0 & -1 & 5 & 3 & -4 \\ 0 & 2 & 0 & 2 & 2 \end{array}\right]$$

$$\rightarrow \begin{array}{l} \\ \\ \\ R4-\frac{2}{4}R3 \end{array} \left[\begin{array}{cccc|c} 2 & 4 & -4 & 0 & 12 \\ 0 & 3 & -3 & -3 & 12 \\ 0 & 0 & 4 & 2 & 0 \\ 0 & 0 & 2 & 4 & -6 \end{array}\right] \rightarrow \left[\begin{array}{cccc|c} 2 & 4 & -4 & 0 & 12 \\ 0 & 3 & -3 & -3 & 12 \\ 0 & 0 & 4 & 2 & 0 \\ 0 & 0 & 0 & 3 & -6 \end{array}\right] \tag{105}$$

This completes forward elimination. Note that the resulting upper-triangular system is the augmented-matrix form of $U\vec{x}=\vec{d}$ in the example. Because the original and final systems are related through EROs, they must be equivalent. Because $\det(U)=(2)(3)(4)(2)=48\neq 0$, the system is non-singular and has a unique solution. It remains to solve the upper-triangular system by back substitution (BS).

$$\begin{aligned} 3x_4=-6 &\Rightarrow & x_4&=-2 \\ 4x_3+2(-2)=0 &\Rightarrow 4x_3=4 &\Rightarrow\ x_3&=1 \\ 3x_2-3(1)-3(-2)=12 &\Rightarrow 3x_2=9 &\Rightarrow\ x_2&=3 \\ 2x_1+4(3)-4(1)+0(-2)=12 &\Rightarrow 2x_2=4 &\Rightarrow\ x_1&=2 \end{aligned}$$

The final result, $\vec{x}=[2,3,1,-2]^T$, is the unique solution to the original system, which can be verified by substitution.

6.4.4 Checking by Residuals

There is another way to check the correctness of the candidate solution $\vec{x}$ of linear system $A\vec{x} = \vec{b}$. If $\vec{x}$ is correct exactly, then the product $A\vec{x}$ should exactly equal $\vec{b}$, in which case $\vec{b} - A\vec{x} = 0$. This is indeed the case for the previous example, for which

$$\vec{b} - A\vec{x} = \begin{bmatrix} 12 \\ 18 \\ 8 \\ 8 \end{bmatrix} - \begin{bmatrix} 2 & 4 & -4 & 0 \\ 1 & 5 & -5 & -3 \\ 2 & 3 & 1 & 3 \\ 1 & 4 & -2 & 2 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ 1 \\ -2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

In general, for large linear systems in which round-off error plays a role, the computed numerical solution will not be exact. A measure of the fidelity of the computed solution is given by the *residual vector* $\vec{r}$, defined as

$$\vec{r} = \vec{b} - A\vec{x} \tag{106}$$

For an exact solution, $\vec{r} = \vec{0}$. Thus, the departure of the residual from zero is an indirect measure of the error inherent in the numerical solution. How large a residual is tolerable? Recall that absolute error is not very meaningful; relative error is the important concept. Hence, a rule of thumb might be that $||\vec{r}||/||\vec{x}|| < u$, where the double vertical bars denote the Euclidean norm defined previously, and u is the unit round-off error of the machine, also known as machine epsilon. If the residual exceeds this rule of thumb, or if the matrix A is severely *ill-conditioned*, a term meaning almost singular, certain precautions are necessary, among them *iterative refinement*. These topics, beyond the scope of this course, are grist, however, for Math 448.

6.4.5 Operation Count of Gaussian Elimination

How "expensive" is Gaussian elimination (GE) from a computational point of view? Recall that GE = FE + BS, and that for an $n \times n$ system, BS requires exactly n^2 operations. It may not have escaped your notice that the dot product, which requires a single loop of length n, has an $O(n)$ operation count, and that BS, which involves doubly nested loops, has an operation count of $O(n^2)$. This suggests that FE, which requires triply nested loops, has an operation count of $O(n^3)$. This is indeed the case, but let's see if we can put a finer point on it than that.

For STEP 1 of GE, the i and j loops both range from 2 to n. Thus the assignment statement in the innermost loop is executed $(n-1) \times (n-1)$ times. In STEP 2 of GE, the i and j loops each diminsh in length by one; therefore, the assignment statement in the innermost loop is executed $(n-2) \times (n-2)$ times. Proceeding in similar fashion, we can construct a table of operations for FE, as shown in Table 8.

We wish to sum all the operations implied in Table 8. There are neat mathematical formulas to do this for us, but we can get sufficiently close to the answer by simple reasoning. Imagine that we are constructing a pyramid with a square base out of perfectly cubic blocks all of equal dimensions. In the first layer we lay $(n-1) \times (n-1)$ blocks. In the second layer we lay $(n-2) \times (n-2)$ blocks, and so on, $(n-1)$ layers in all. At the top, we will need to lay only a single capstone block. How many blocks are there in total? The volume of a pyramid is $\frac{1}{3}Bh$, where B is the area of the base and h is the height.

outer loop index k	executions of innermost assignment statement
1	$(n-1)^2$
2	$(n-2)^2$
3	$(n-3)^2$
.	.
.	.
.	.
$n-2$	2^2
$n-1$	1^2

Table 8: Operation count for outer steps of Forward Elimination.

The area of the base is $(n-1)^2$ square blocks, and the height is $(n-1)$ blocks, so the volume is (very nearly) $\frac{1}{3}(n-1)^3$ cubic blocks. If n is a large number, then the number of blocks is approximately $\frac{1}{3}n^3$. In this analogy, the number of blocks is a metaphor for the number of times the assignment statement in the innermost loop is executed. That assignment statement involves two operations, a multiplication by t and a subtraction. Hence, the total operation count of FE is approximately $\frac{2}{3}n^3$. But wait, you say. We forgot to count the divisions involved in $t \leftarrow a_{ik}/a_{kk}$ and the multiplications and subtractions involved in $b_i \leftarrow b_i - tb_k$. Right you are. But these assignment statements reside not in the innermost loop, so each is executed fewer times and involves only $O(n^2)$ operations. Once again, if n is large, then FE is dominated by the $O(n^3)$ contribution and not by the paltry $O(n^2)$ contributions. Hence we say that FE has an $O(n^3)$ operation count, or more specifically that the count is approximately $\frac{2}{3}n^3$ operations.

Finally, what is the operation count for Gaussian elimination (GE), which consists of FE followed by BS? Well, once again the $O(n^2)$ operation count of BS pales in comparison to the effort of FE, and so we say the GE requires approximately $\frac{2}{3}n^3$ operations.

The moral of this story is that solving linear systems of equations is computationally expensive. Indeed, one of the driving factors for the prevalance of fast computers in today's world is the necessity of solving large systems of linear equations in reasonable clock time.

6.4.6 Pivoting

What can go wrong during the forward elimination (FE) process of Gaussian elimination (GE)? The worst thing that can happen is for the pivot element a_{kk} to vanish at outer STEP k, in which case there will be a divide by zero run-time error; i.e., a crash. What then do we do if a pivot element vanishes? Does that mean the matrix system is singular? Consider the following simple example, expressed in augmented-matrix form:

$$\left[\begin{array}{ccc|c} 0 & 1 & 2 & 3 \\ 1 & 0 & 4 & 5 \\ 0 & 2 & 1 & 3 \end{array}\right]$$

The very first step of FE fails because pivot element $a_{11} = 0$. There is nothing wrong with the system, however. All it takes to get us out of this corner is to swap rows one and two (R1 and R2) to obtain

$$\left[\begin{array}{ccc|c} 1 & 0 & 4 & 5 \\ 0 & 1 & 2 & 3 \\ 0 & 2 & 1 & 3 \end{array}\right]$$

From here, GE proceeds normally to the solution $x_1 = x_2 = x_3 = 1$. So, clearly, the vanishing of a pivot element does not always spell disaster. Then again, sometimes it does.

DEF: The interchange of two rows during the FE process of GE to avoid a zero pivot element is known as *pivoting*.

Algorithm 6.4 below, which is to be inserted at the top of the outer (k) loop of FE (ALG 6.3), represents one of the simplest pivoting strategies.

ALGORITHM 6.4: Partial Pivoting for Forward Elimination

```
if (a_kk = 0) then

    SINGULAR ← TRUE (SINGULAR is a logical variable)

    for i = k+1, k+2, ..., n
    |
    |    if (a_ik ≠ 0) then
    |        swap rows k and i
    |        SINGULAR ← FALSE
    |        EXIT loop
    |    end if
    |__

end if

if (SINGULAR) then
    output error message
    stop
end if
```

A few words about ALG 6.4 are in order. First, the algorithm inserts a test of the pivot element to ensure that FE does not encounter a zero divisor. If the pivot element at STEP k is indeed zero, then one of two situations prevails: 1) either the matrix is singular, or 2) the matrix is non-singular, and the problem can be corrected by interchanging rows. The presumption of singularity is made, but this presumption is reversed if the search below the diagonal of column k turns up a non-zero value. If no non-zero value can be found, the matrix truly is singular, an error message is output, and the computation is brought gracefully to a halt (rather than a crash).

REMARKS:

1. A matrix that is almost singular is called *ill-conditioned*. During FE of ill-conditioined matrices the pivot element almost vanishes in some relative sense.

2. Thus, pivoting is also used to minimize the accumulation of round-off error during GE, particularly for ill-conditioned systems.

3. Most linear-algebra software packages involve more sophistocated pivoting algorithms than ALG 6.4, such as *scaled partial pivoting* or even *full pivoting*. These are somewhat beyond the scope of this course. Another great reason to take Math 448!

6.4.7 The Determinant (Revisited)

In linear algebra courses, one is often taught to find the determinant of a square matrix by a method called *expansion by minors*. The method is OK in theory, but, carrying an $O(n!)$ operation count, it could hardly be worse from an algorithmic point of view. Here, we consider a much more practical way of computing the determinant. It begins with a question: given square matrix A and corresponding upper-triangular matrix U related to A through EROs, what is the relationship between $\det(A)$ and $\det(U)$, if any?

First, consider that, when solving linear systems by hand, it is often convenient to clear fractions in a given equation (row) by multiplying the row by a non-zero scalar. For machine implementations of GE, however, the machine really doesn't care whether a number is whole or fractional, and so we can rule out the ERO of simply multiplying a row by a scalar. Note that a linear combination of rows (say, $\mathrm{R}i + c\mathrm{R}j$) is still allowed, but the operation $c\mathrm{R}j$ alone is now ruled out, the later of which multiplies $\det(A)$ by c.

If we also temporarily prohibit pivoting, then a wonderful thing happens: $\det(A) = \det(U)$. (We will omit the proof of this.) But pivoting may be absolutely necessary, then what? Each row interchange simply changes the sign of the determinant. So then, if the number of row interchanges during FE is odd, the sign of the determinant is reversed. On the other hand, if the number of interchanges is even, then the sign remains unchanged. Putting all of this together, we can get $\det(A)$ nearly for free during FE by the following rubric:

1. During FE of square matrix A, keep track of p, the number of row interchanges during pivoting.

2. If the pivoting strategy fails, then A is singular and $\det(A) = 0$.

3. If the pivoting strategy succeeds, at the end of FE, compute $\det(U) = \Pi_{k=1}^{n} u_{kk}$.

4. Finally, $\det(A) = (-1)^p \det(U)$.

EX: Compute $\det(A)$ for the matrix below:

$$A = \begin{bmatrix} 2 & 4 & -4 & 0 \\ 1 & 5 & -5 & -3 \\ 2 & 3 & 1 & 3 \\ 1 & 4 & -2 & 2 \end{bmatrix}$$

We have already shown in an example above that A is related to the following upper-triangular matrix U through EROs without pivoting (or pure multiplication of rows by scalars):

$$U = \begin{bmatrix} 2 & 4 & -4 & 0 \\ 0 & 3 & -3 & -3 \\ 0 & 0 & 4 & 2 \\ 0 & 0 & 0 & 3 \end{bmatrix}$$

Hence, $p = 0$, and $\det(A) = (-1)^0 \det(U) = (2)(3)(4)(3) = 72 \neq 0$. Matrix A is non-singular and any system $A\vec{x} = \vec{b}$ will have a unique solution.

6.5 The Matrix Inverse

To set up an analogy, let's consider the simplest scalar algebraic equation imaginable:

$$ax = b \tag{107}$$

where a, b, and x are scalars, x is the unknown and a and b are the *data*. Even this simple equation presents three different scenarios. First, suppose $a \neq 0$. Then $a^{-1} = 1/a$ exists, in which case we can multiply both sides of the equation by a^{-1} to obtain $x = a^{-1}b$ because $a^{-1}a = 1$. However, if $a = 0$, there remain two possible scenarios to consider. If $b = 0$, then *any* value of x satisfies $ax = 0$ (provided, of course $a = 0$). On the other hand, *no* value of x satisfies $ax = b$ if $a = 0$ but $b \neq 0$. So, in this trivial example, there exist a unique solution, an infinity of solutions, or no solution at all, depending upon the values of the data a and b. In particular, if $a \neq 0$, a unique solution exists.

Now consider the matrix-vector analog of the scalar algebraic equation previously considered:

$$A\vec{x} = \vec{b} \tag{108}$$

Our experience with the scalar analogy begs the question: Given matrix A, under what circumstances, if any, does A^{-1} exist, in which case the unique solution to Eq. 108 should be $\vec{x} = A^{-1}\vec{b}$?

DEF: Given $n \times n$, matrix A, if there exists $n \times n$ matrix B such that $AB = BA = I_n$, then B is called the inverse of A and is denoted A^{-1}. A matrix A for which A^{-1} exists is said to be *invertible*.

The definition above is a little unusual. We have defined the inverse of a matrix, but it is not yet clear that such things exist. It is a little bit like defining an elephant as a large four-legged animal with a "tail" at both ends. If we see such an animal, we'll know what to call it. However, just defining the creature doesn't mean it exists. (Dragons are pretty well defined too, and not many of those have been seen.)

THEOREM (Uniqueness of the matrix inverse:) Suppose A is invertible. Then A^{-1} is unique.

PROOF (by contradiction): Suppose matrix A, which is invertible, has two distinct inverses, B and C. Then, by definition of the inverse

$$AB = I$$

$$
\begin{aligned}
\Rightarrow CAB &= CI \\
\Rightarrow (CA)B &= CI \text{ (associativity of matrix multiplication)} \\
\Rightarrow IB &= CI \text{ (because } C \text{ is presumed an inverse of } A) \\
\Rightarrow B &= C \text{ (because } I \text{ is the multiplicative identity)}
\end{aligned}
$$

However, the last line above is a contradiction of the original assumption that A had two *distinct* inverses. Hence, there must be one and only one inverse.

Despite the fact that the matrix inverse, if it exists, must be unique, we still haven't concluded that such things exist. Here then is proof positive that at least one such inverse exists. Consider the 2×2 matrix A given below.

$$A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \tag{109}$$

The reader can verify that $AA^{-1} = A^{-1}A = I_2$ for

$$A^{-1} = \begin{bmatrix} 2/3 & -1/3 \\ -1/3 & 2/3 \end{bmatrix} \tag{110}$$

So, in at least this instance, A^{-1} is for real. But, how, in general, do we know when the inverse exists and, moreover, how do we compute the inverse? A 3×3 example should suffice to shed light on the situation.

Consider invertible $A \in M_{3\times 3}$, given below with generic elements:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \tag{111}$$

Because A is invertible, we know that A^{-1} exists, but we don't know its elements. So, let's also define A^{-1} generically as

$$A^{-1} = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} \tag{112}$$

Furthermore, consider the three vectors $\vec{e}_1 = [1,0,0]^T$, $\vec{e}_2 = [0,1,0]^T$, and $\vec{e}_3 = [0,0,1]^T$, which are known as the *standard basis vectors* in R^3 because any vector can be readily expressed as a linear combination of these three basis vectors. Finally, consider the solutions of the following three related linear systems:

$$
\begin{aligned}
A\vec{c}_1 &= \vec{e}_1 = [1,0,0]^T \\
A\vec{c}_2 &= \vec{e}_2 = [0,1,0]^T \\
A\vec{c}_3 &= \vec{e}_3 = [0,0,1]^T
\end{aligned} \tag{113}
$$

Because A is invertible, we know that the solution of the first linear system is $\vec{c}_1 = A^{-1}\vec{e}_1$. But $A^{-1}\vec{e}_1$ is

$$\begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} b_{11} \\ b_{21} \\ b_{31} \end{bmatrix} \tag{114}$$

which is the first column of A^{-1}. That is, $\vec{c}_1$ is the first column of A^{-1}. Similarly, the solution $\vec{c}_2$ of system $A\vec{c}_2 = \vec{e}_2$ is the second column of A^{-1}, and the solution $\vec{c}_3$ of system $A\vec{c}_3 = \vec{e}_3$ is the third column of A^{-1}. Thus, we can construct the inverse of A one column at a time by solving the three linear systems of Eq. 113 in succession.

This game plan will indeed work provided A is invertible. But it is wasteful because FE, which is costly, is being replicated three times. It is far more efficient to do the FE only once, which can be done by considering the following augmented matrix:

$$\left[\begin{array}{ccc|c|c|c} a_{11} & a_{12} & a_{13} & 1 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 1 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & 1 \end{array}\right] \tag{115}$$

Note that the right half of the augmented matrix above is the 3×3 identity matrix, whose columns are $\vec{e}_1$, $\vec{e}_2$, and $\vec{e}_3$. That is, we begin with $[A|I]$. From here we proceed with FE, operating on all three right-hand sides simultaneously. When A has been driven to upper-triangular form, we then do three back substitutions, one each for $\vec{c}_1$, $\vec{c}_2$, and $\vec{c}_3$. Let's put the theory to practice with a numerical 3×3 example.

EX: Compute A^{-1} for matrix A below:

$$\left[\begin{array}{ccc} 5 & -1 & 2 \\ 5 & -1 & 7 \\ -5 & 2 & 4 \end{array}\right]$$

We begin with

$$\left[\begin{array}{ccc|ccc} 5 & -1 & 2 & 1 & 0 & 0 \\ 5 & -1 & 7 & 0 & 1 & 0 \\ -5 & 2 & 4 & 0 & 0 & 1 \end{array}\right]$$

Proceeding with FE, we obtain

$$\begin{array}{l} (R2-R1) \\ (R3+R1) \end{array} \left[\begin{array}{ccc|ccc} 5 & -1 & 2 & 1 & 0 & 0 \\ 5 & -1 & 7 & 0 & 1 & 0 \\ -5 & 2 & 4 & 0 & 0 & 1 \end{array}\right] \rightarrow \left[\begin{array}{ccc|ccc} 5 & -1 & 2 & 1 & 0 & 0 \\ 0 & 0 & 5 & -1 & 1 & 0 \\ 0 & 1 & 6 & 1 & 0 & 1 \end{array}\right]$$

$$(\text{Swap } R2\&R3)$$

$$\rightarrow \left[\begin{array}{ccc|ccc} 5 & -1 & 2 & 1 & 0 & 0 \\ 0 & 1 & 6 & 1 & 0 & 1 \\ 0 & 0 & 5 & -1 & 1 & 0 \end{array}\right]$$

Note that the original system is now in the form $U\vec{c}_j = \vec{d}_j$, where $\vec{d}_j$ is the vector in the jth column of the rightmost three columns of the augmented matrix following FE. Now that FE is complete, we may proceed with three back substitutions, the first of which solves $U\vec{c}_1 = \vec{d}_1$, where $\vec{c}_1 = [b_{11}, b_{21}, b_{31}]^T$. Here is the system with unknowns restored:

$$\begin{aligned} 5b_{11} - 1b_{21} + 2b_{31} &= 1 \\ 1b_{21} + 6b_{31} &= 1 \\ 5b_{31} &= -1 \end{aligned}$$

Back substitution leads to $b_{31} = -1/5$, $b_{21} = 11/5$, and $b_{11} = 18/25$, or $\vec{c}_1 = [18/25, 11/5, -1/5]^T$. Similarly, back substitutions with right-hand-side data $\vec{d}_2$ and $\vec{d}_3$ lead to $\vec{c}_2 = [-8/25, -6/5, 1/5]^T$ and $\vec{c}_3 = [1/5, 1, 0]^T$, respectively. Each of these three solutions gives a column of A^{-1}; hence

$$A^{-1} = \begin{bmatrix} 18/25 & -8/25 & 1/5 \\ 11/5 & -6/5 & 1 \\ -1/5 & 1/5 & 0 \end{bmatrix}$$

which can be verified by computing AA^{-1} and $A^{-1}A$.

To generalize to invertible $n \times n$ matrices, the columns $\vec{c}_j$ of the inverse of A can be generated one by one by solving $A\vec{c}_j = \vec{e}_j$ in succession for $j = 1, 2, ..., n$, where $\vec{e}_j$ is the jth standard basis vector in R^n. As such, it consists of all 0's with the sole exception of 1 as the jth component. To avoid repeating the computationally intensive FE n times, FE is done only once, for the augmented matrix $[A|I_n]$, where the columns of I_n are the standard basis vectors in R^n. Following the completion of FE, n back substitions yield the column vectors of A in succession.

6.5.1 Gauss-Jordan Elimination

Although less efficient computationally than Gaussian elimination, Gauss-Jordan elimination is often preferred for finding the matrix inverse by hand computation when the size of the matrix n is small, say 3 or 4. Gauss-Jordan elimination consists of forward elimination followed by *back elimination*, or symbolically GJ = FE + BE. The FE part is the same as for GE. To illustrate backward elimination (BE), let's return to our previous example at the end of the FE step (Eq. 116).

$$\left[\begin{array}{ccc|ccc} 5 & -1 & 2 & 1 & 0 & 0 \\ 0 & 1 & 6 & 1 & 0 & 1 \\ 0 & 0 & 5 & -1 & 1 & 0 \end{array}\right]$$

At this stage we now perform EROs on the left-hand matrix to eliminate the entries *above* the diagonal in a way similar to that during FE in which we eliminated entries *below* the diagonal. The primary difference is that the pivot elements $a''_{kk} = u_{kk}$ are used in reverse order, from last to next-to-last, and so on; hence, *backward* elimination.

$$\begin{array}{l} (R1 - \frac{2}{5}R3) \\ (R2 - \frac{6}{5}R3) \end{array} \left[\begin{array}{ccc|ccc} 5 & -1 & 2 & 1 & 0 & 0 \\ 0 & 1 & 6 & 1 & 0 & 1 \\ 0 & 0 & 5 & -1 & 1 & 0 \end{array}\right] \rightarrow \begin{array}{l} (R1 + R2) \\ \\ \end{array} \left[\begin{array}{ccc|ccc} 5 & -1 & 0 & 7/5 & -2/5 & 0 \\ 0 & 1 & 0 & 11/5 & -6/5 & 1 \\ 0 & 0 & 5 & -1 & 1 & 0 \end{array}\right]$$

$$\rightarrow \begin{array}{l} (\frac{1}{5}R1) \\ \\ (\frac{1}{5}R3) \end{array} \left[\begin{array}{ccc|ccc} 5 & 0 & 0 & 18/5 & -8/5 & 1 \\ 0 & 1 & 0 & 11/5 & -6/5 & 1 \\ 0 & 0 & 5 & -1 & 1 & 0 \end{array}\right] \rightarrow \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 18/25 & -8/25 & 1/5 \\ 0 & 1 & 0 & 11/5 & -6/5 & 1 \\ 0 & 0 & 1 & -1/5 & 1/5 & 0 \end{array}\right] \quad (116)$$

The last step of GJ elimination is simply a scaling operation that produces the identity matrix on the left-hand side of the augmented system.

To summarize Gauss-Jordan elimination, we use EROs to drive the augmented matrix $[A|I_n] \rightarrow [I_n|B]$. When the $n \times n$ identity matrix appears on the left side of the augmented matix, the matrix B on the right-hand side is A^{-1}.

Solutions to linear systems, or matrix inverses, can be found either by GE or by GJ. Of the two, GE is the more efficient. Because BE is nearly as computationally intensive as FE, GJ has an asymptotic operation count of n^3, one and one-half times that of GE for large systems.

6.6 Linear Algebra Summary

Before proceeding to our last subsection, let's summarize what we have learned about solving linear systems of the form $A\vec{x} = \vec{b}$. The following statements are equivalent; that is, the truth of any one implies the truth of all:

1. $A \in M_{n\times n}$ is invertible.
2. A^{-1} exits, and it is unique.
3. $\det(A) \neq 0$.
4. $A\vec{x} = \vec{b}$ has the unique solution $\vec{x} = A^{-1}\vec{b}$.
5. $A\vec{x} = \vec{0}$ has only the trivial solution $\vec{x} = \vec{0}$.
6. The rows of matrix A are linearly independent.

Item 6 above needs some explanation, because we have not yet defined *linear independence*.

DEF: A set of n-vectors $\{\vec{v}_1, \vec{v}_2, ..., \vec{v}_p\}$ is *linearly independent* iff the linear combination $c_1\vec{v}_1 + c_2\vec{v}_2 + ... + c_p\vec{v}_p = \vec{0} \Rightarrow c_1 = c_2 = ... = c_p = 0$, where the c_k are scalars. Conversely, if the linear combination $c_1\vec{v}_1 + c_2\vec{v}_2 + ... + c_p\vec{v}_p = \vec{0}$ with not all the $c_k = 0$, then the vectors are *linearly dependent*.

From an intuitive point of view, a linearly dependent set of vectors is deficient in information. For example, suppose $c_1\vec{v}_1 + c_2\vec{v}_2 + ... + c_p\vec{v}_p = \vec{0}$ with not all the $c_k = 0$, and in particular, suppose $c_1 \neq 0$. Then $\vec{v}_1 = -\frac{1}{c_1}\sum_{k=2}^{p} c_k\vec{v}_k$; that is, $\vec{v}_1$ can be expressed as a linear combination of the remaining vectors. This implies that some information contained in the set of vectors is redundant. In regard to linear systems of equations, if the rows of matrix A are not linearly independent, then the set of equations is redundant in some sense. This sort of redundancy will manifest itself during forward elimination by the failure of pivoting to find a non-zero pivot element, in which case A is singular.

Finally, we wish to summarize the operation counts of some of the algorithms that we have discussed. This is done in Table 9, where A and B are $n \times n$ matrices and $\vec{x}$, and $\vec{y}$ are n-vectors.

We wish to close this section on a very important note. Suppose we are faced with the task of solving the linear system $A\vec{x} = \vec{b}$ where A is an invertible square matrix. According to the theory of linear algebra, in particular Item 4 above, we *could* solve the system by first computing the inverse A^{-1} and then multiplying $A^{-1}\vec{b}$ to get $\vec{x}$. This is a *terrible* idea from a computational point of view. Why?

Matrix and/or Vector Operation	Operation Count (Approximate)
$\vec{x} \cdot \vec{y}$	$2n$
$A\vec{x}$	$2n^2$
AB	$2n^3$
BS	n^2
FE	$\frac{2}{3}n^3$
GE=FE+BS	$\frac{2}{3}n^3 + n^2 \approx \frac{2}{3}n^3$
GJ=FE+BE	n^3

Table 9: Operation counts of selected matrix, vector, and matrix-vector operations.

6.7 Special Matrices

The matrices that arise from physical systems or engineering problems often exhibit symmetries. Symmetry can be exploited to simplify Gaussian elimination, sometimes dramatically. In this final subsection, we discuss a few types of special matrices; that is, matrices that manifest some type of symmetry or unusual structure.

First, we remind the reader of a few definitions already discussed, to which we add a few more:

DEF:

1. A *dense* matrix is one all (or nearly all) of whose entries are non-zero.
2. Conversely, a *sparse* matrix is one with mostly zero entries that are more or less randomly distributed.
3. A *symmetric* matrix is one that is equal to its own transpose.
4. A *diagonal* matrix is one whose non-zero elements lie only along the diagonal. That is, $a_{ij} = 0$ except when $i = j$.
5. A *banded* matrix is one whose non-zero elements lie only near the diagonal. That is, $a_{ij} = 0$ except when $|i - j| \leq b$, where $0 \leq b$ is the *semi bandwidth*, and $2b + 1$ is the bandwidth. By this definition, a diagonal matrix is banded with $b = 0$.
6. A *tridiagonal* matrix is a banded matrix for which $b = 1$. That is, a tridiagonal matrix has non-zero entries only for elements $a_{i,i-1}$, $a_{i,i}$, and $a_{i,i+1}$.

For our last algorithm of the chapter, we adapt Gaussian elimination to tridiagonal systems of equations, which arise when solving ordinary differential equations (ODEs) of boundary-value type (BVP) by the finite-difference method (FDM). The FDM will be discussed later in the context of numerical differentiation. For now, we will simply focus on developing an appropriate algorithm for tridiagonal systems.

In general, a tridiagonal matrix assumes the following form.

$$\begin{bmatrix} d_1 & u_1 & 0 & 0 & \dots & 0 \\ l_2 & d_2 & u_2 & 0 & \dots & 0 \\ 0 & l_3 & d_3 & u_3 & \dots & 0 \\ & & . & . & \dots & . \\ & & & . & . & . \\ 0 & \dots & 0 & l_{n-1} & d_{n-1} & u_{n-1} \\ 0 & \dots & 0 & 0 & l_n & d_n \end{bmatrix}$$

Because most of the elements are zero, only the non-zero matrix elements are stored, typically in three n-vectors: $\vec{l}$, $\vec{d}$, and $\vec{u}$. The diagonal elements are stored in $\vec{d}$. *Subdiagonal* and *superdiagonal* elements are stored in vectors $\vec{l}$ and $\vec{u}$, respectively. Note that elements l_1 of the subdiagonal and u_n of the superdiagonal are unused.

Let's now consider a 4×4 tridiagonal system and run through the steps of Gaussian elimination, after which we can readily generalize to $n \times n$ tridiagonal systems.

$$\left[\begin{array}{cccc|c} d_1 & u_1 & 0 & 0 & b_1 \\ l_2 & d_2 & u_2 & 0 & b_2 \\ 0 & l_3 & d_3 & u_3 & b_3 \\ 0 & 0 & l_4 & d_4 & b_4 \end{array}\right] \xrightarrow{(R2 - \frac{l_2}{d_1} R1)} \left[\begin{array}{cccc|c} d_1 & u_1 & 0 & 0 & b_1 \\ 0 & d_2' & u_2 & 0 & b_2' \\ 0 & l_3 & d_3 & u_3 & b_3 \\ 0 & 0 & l_4 & d_4 & b_4 \end{array}\right] \xrightarrow{(R3 - \frac{l_3}{d_2'} R2)} \left[\begin{array}{cccc|c} d_1 & u_1 & 0 & 0 & b_1 \\ 0 & d_2' & u_2 & 0 & b_2' \\ 0 & 0 & d_3' & u_3 & b_3' \\ 0 & 0 & l_4 & d_4 & b_4 \end{array}\right] \xrightarrow{(R4 - \frac{l_4}{d_3'} R3)} \left[\begin{array}{cccc|c} d_1 & u_1 & 0 & 0 & b_1 \\ 0 & d_2' & u_2 & 0 & b_2' \\ 0 & 0 & d_3' & u_3 & b_3' \\ 0 & 0 & 0 & d_4' & b_4' \end{array}\right]$$

Forward elimination is now complete. As per our previous conventions, red is used to highlight the pivot element at each step, and blue is used to show the elements that need to be zeroed. Primes denote elements that have been modified from their original values. Note that the vectors $\vec{d}$ and $\vec{b}$ are "clobbered" (that is, overwritten with new values), but the vectors $\vec{l}$ and $\vec{u}$ remain unchanged. (We know, it looks like the elements l_i of the subdiagonal have been clobbered, but in the actual algorithm below, it is unnecessary to set the values to zero because the values l_i are never again used.) Note that for each step of FE, only a single element below the diagonal needs to be zeroed. For this reason, tridiagonal GE is vastly more efficient than standard GE. When FE has been completed, the resulting upper-triangular matrix is *bidiagonal*, with non-zero elements confined to the diagonal and the superdiagonal only.

It is now time for back substitution. That too is extremely simple, considering that U is bidiagonal. Here is the full BS process.

$$\begin{aligned} x_4 &= b_4'/d_4' \\ x_3 &= (b_3' - u_3 x_4)/d_3' \\ x_2 &= (b_2' - u_2 x_3)/d_2' \\ x_1 &= (b_1' - u_1 x_2)/d_1' \end{aligned}$$

Here is an algorithm GE of and $n \times n$ tridiagonal system, based on our experience with a 4×4 system.

ALGORITHM 6.5: Tridiagonal Gaussian Elimination

input (pass) order n
input (pass) n*-vectors* $\vec{l}$, $\vec{d}$, *and* $\vec{u}$
input (pass) right-hand-side n*-vector* $\vec{b}$

for $i = 2,3,\ldots,n$ *(Forward Elimination)*
|
| $t \leftarrow l_i/d_{i-1}$ *(scalar temporary)*
| $d_i \leftarrow d_i - tu_{i-1}$ *(*$\vec{d}$ *is overwritten)*
| $(l_i \leftarrow 0)$ *(extraneous)*
| $b_i \leftarrow b_i - tb_{i-1}$ *(*$\vec{b}$ *is overwritten)*
|__

$x_n \leftarrow b_n/d_n$ *(last unknown outside the loop)*

for $i = n-1, n-2, \ldots, 1$ *(Back Substitution)*
|
| $x_i \leftarrow (b_i - u_i x_{i+1})/d_i$
|__

return $\vec{x}$

An alternate version of BS for tridiagonal matrices, shown below, overwrites the right-hand-side vector $\vec{b}$ with the solution $\vec{x}$ to save one array of storage.

$b_n \leftarrow b_n/d_n$ *(last unknown outside the loop)*

for $i = n-1, n-2, \ldots, 1$ *(Back Substitution)*
|
| $b_i \leftarrow (b_i - u_i b_{i+1})/d_i$
|__

return $\vec{b}$ *(upon return contains solution* $\vec{x}$*)*

HW1: Compute the operation count for GE of an $n \times n$ tridiagonal system, and compare with the operation count for solving an $n \times n$ dense system.

To conclude the chapter, we mention a few other special matrix-vector systems.

The solution of linear systems of equations is so important in science and industry that specialized subprograms have been written for almost every conceivable matrix type. One of the most extensive collections of linear algebra subroutines is that maintained by AT&T Bell Labs, Oak Ridge National Laboratory, and the University of Tennessee at Knoxville. These subroutines are made available for free at www.netlib.org. At last count, the site had received some 400,000,000 hits. Within this vast col-

lection are several prominent linear-algebra packages deserving of brief mention: LAPACK, EISPACK, SPARSEPAK, and FISHPACK. LAPACK, short for Linear Algebra PACKage, is a collection of generic linear-algebra routines for most common purposes. It contains routines that return single or double precision results and routines for both real and complex matrices. SPARSEPAK contains routines specialized to sparse matrices. The names EISPACK and FISHPACK are both puns. The former contains routines for efficiently finding the eigenvalues of large matrices. We haven't talked at all about eigenvalues, but they figure prominently in all sorts of scientific problems, especially those that involve determining the dynamic *stability* of structures; e.g., an aircraft wing. Eigenvalues and eigenvectors will be defined and addressed in Math 238 and Math 448. A type of partial differential equation (PDE) that arises frequently in mathematics is Poisson's equation. It's numerical approximation often involves solving a linear system of equations with a *block-tridiagonal* structure. The word "fish" in English is "poisson" in French. Hence, some computational scientist with a wry sense of humor dubbed the package of routines for solving Poisson's equation with the odiforous moniker FISHPACK.

The reader is encouraged to browse www.netlib.org to gain some appreciation for the extreme importance of numerical linear algebra to the world of science and engineering.

6.8 Exercises

1. Consider the linear system of equations given in Eq. 86. Express this system in the matrix-vector form of Eq. 99. Be sure to identify A, $\vec{x}$, and $\vec{b}$ explicitly.

2. Find two square matrices A and B for which $AB \neq BA$.

3. Find two square matrices A and B for which $AB = [0]$, but for which *neither* $A = [0]$ *nor* $B = [0]$.

4. By considering operation counts explain why it is usually a terrible idea to compute the inverse of a matrix to solve a linear system of equations.

5. Assuming that A is an invertible $n \times n$ matrix, prove Item 5 above.

6. Find the inverse of the matrix
$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$
in two different ways: 1) FE followed by 3 back substitutions, and 2) FE followed by BE.

7 Part II: Polynomial Interpolation and Approximation

At the very end of the section on Rootfinding, we learned how Newton's method for finding roots can be used behind the scenes to create a square-root function; that is, to give the output $\sqrt{x}$ to any input $x \geq 0$. The square-root function is but one of a panoply of intrinsic functions on a standard scientific calculator. You probably have not lost sleep wondering how your calculator or your computer evaluates intrinsic functions such as $\cos(x)$, $\sin(x)$, $\ln(x)$, e^x, and so on. Nevertheless, it is an interesting question. And the truth is that your calculator does *not* evaluate such functions because it *cannot*! What it does do is to closely *approximate* these functions by other functions that are easy to evaluate.

In this section we will consider only *polynomial approximants*. We remind you of the definition of a polynomial, which was presented previously in Section 5.

DEF: If $a_n \neq 0$, then a function of the form

$$\begin{aligned} P_n(x) &= a_0 + a_1 x + a_2 x^2 + \ldots a_{n-1} x^{n-1} + a_n x^n \\ &= \sum_{k=0}^{n} a_k x^k \end{aligned} \tag{117}$$

is called a *polynomial* in x of *degree* (order) n.

Polynomials have some nice properties. The first is that they are easy to evaluate. Recall that a polynomial of degree n can be evaluated in only $2n$ operations by ALG 1.1 (Horner's Method). Moreover, derivatives and antiderivatives of polynomials are also polynomials, so they too are easy to compute and easy to evaluate. For example, by differentiating $P_n(x)$ in Eq. 117, we obtain

$$\begin{aligned} P_n'(x) &= a_1 + 2a_2 x + \ldots (n-1) a_{n-1} x^{n-2} + n a_n x^{n-1} \\ &= \sum_{k=1}^{n} k a_k x^{k-1} \end{aligned} \tag{118}$$

Similarly, by integrating Eq. 117 term-by-term, we obtain the indefinite integral of a polynomial, namely

$$\begin{aligned} \int P_n(x) dx &= C + a_0 x + a_1 \frac{x^2}{2} + a_2 \frac{x^3}{3} + a_{n-1} \frac{x^n}{n} + a_n \frac{x^{n+1}}{n+1} \\ &= C + \sum_{k=0}^{n} \frac{a_k}{k+1} x^{k+1} \end{aligned} \tag{119}$$

But by far the most useful attribute of polynomials is their ability at mimicry. Polynomials it seems can "look like" virtually any continuous function. This property is codified in the famous Weierstrass Approximation Theorem, which we state here without proof.

Weierstrass Approximation THEOREM (WAT): Suppose f is defined and continuous on closed interval $[a,b]$. For each $\delta > 0$, there exists a polynomial $P(x)$ also defined on $[a,b]$ for which

$$|f(x) - P(x)| < \delta \text{ for all } x \in [a,b]$$

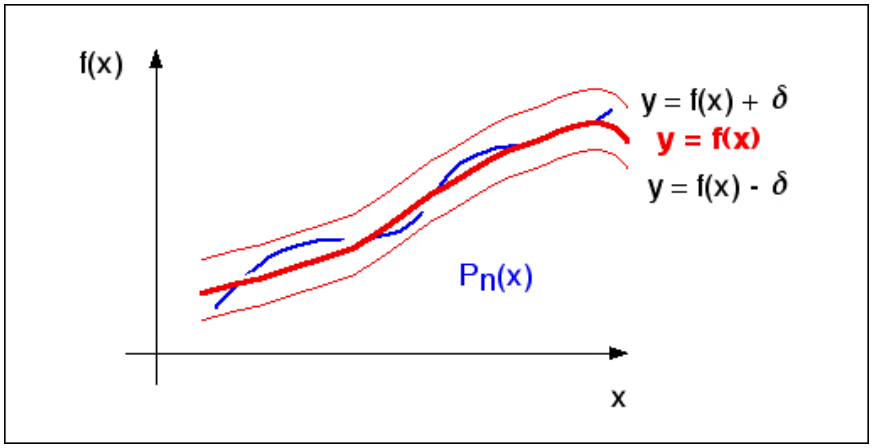


Figure 17: Illustration of Weierstrass Approximation Theorem.

Notice that, if $f(x)$ is the exact function and $P(x)$ is the *approximant*, then their difference is the error of the approximant. The WAT states that, for any bound $\delta > 0$ on the approximation error, no matter how small, a polynomial approximant to f can be found that satisfies the specified error constraint (Fig. 17). In plain English, any continuous function can be approximated as accurately as you please by a polynomial.

To summarize, the game plan for this section is as follows. Given a function $f(x)$, an interval $[a,b]$, and an absolute error tolerance $\delta > 0$, find polynomial $P(x)$ that lies within a distance δ of f on the entire interval. Then use $P(x)$ instead of f because P is easy and quick to evaluate.

The WAT tells us that our game plan is possible, but it does not tell us *how* to find the polynomial approximant $P(x)$. In general, there are several different criteria for finding $P(x)$, as illustrated in Figs. 18, 19, and 20.

One schema is polynomial *interpolation*. In this scenario, the polynomial approximant $P(x)$ is forced to exactly match the function $f(x)$ at a finite set of points $[x_i, f(x_i)]$, $i = 0,1,...,n$ within the interval $[a,b]$. The abscissas (x values) of these points are called *nodes*. Thus, $P(x_i) = f(x_i)$ at each of the $n+1$ nodes, as shown in Fig. 18.

The second scenario is quite different. Here, we focus on a single point $[a, f(a)]$. In a small vicinity of the node $x = a$, given by the interval $[a-\delta, a+\delta]$, $\delta > 0$, we want the approximant $T(x)$ to "look like" the function $f(x)$ as much as possible. Outside this small interval, we allow the approximant to depart radically from the function. Let the approximation error be denoted $e(x) = f(x) - T(x)$. The criterion for choosing $T(x)$ is $|e(x)| << 1$ for $x \in [a-\delta, a+\delta]$. In this case, we force $T(x)$ to "look like" $f(x)$ by requiring $T(a) = f(a)$, $T'(a) = f'(a)$, $T''(a) = f''(a)$, etc. This idea leads to the *Taylor polynomial*; hence, the use of the symbol $T(x)$.

Finally, let's consider one other scenario, altogether different from the first two. In this case, the function $f(x)$ is not known explicitly, but is given by a discrete data set known to contain measurement errors, also called *noise*. Thus if $f(x)$ is the exact but unknown function, our data consists of the set (x_i, y_i), $i = 0,1,2...,n$, where $y_i = f(x_i) + e_i$, and e_i is the error of measurement i. In this case, it would

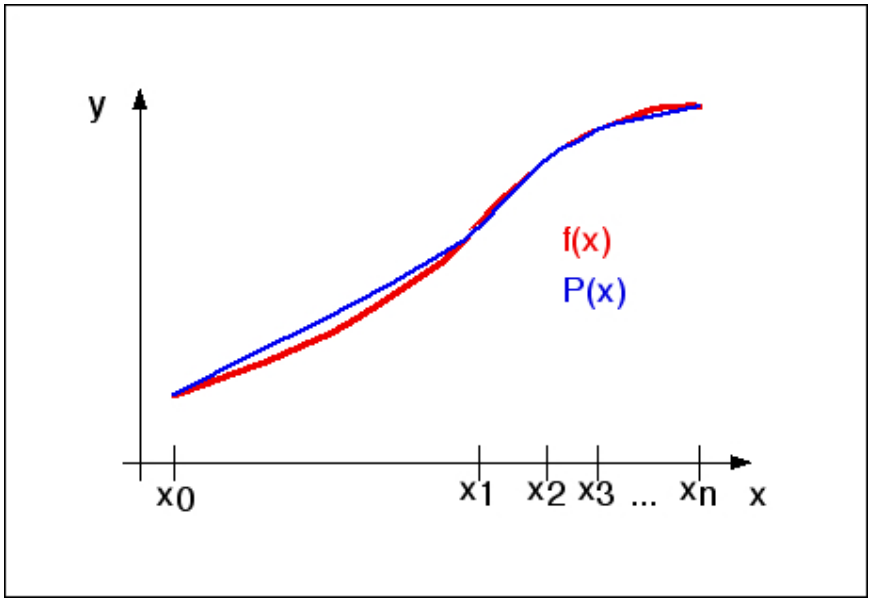


Figure 18: Polynomial interpolation.

be foolish to fit the data exactly, because the data are in error. Thus, the goal is to fit the *trend* of the data without fitting each data point precisely. This schema leads to the *least-squares polynomial* approximant.

There is yet a fourth schema, *spline interpolants*, not shown by a representative figure here. Splines are piecewise, low-order polynomials that are designed to fit the data precisely while maintaining continuity of low-order derivatives at the nodes. Splines are magnificent numerical devices, but slightly beyond the scope of this course.

In Math 248, we will study in detail the first two of these approximation tools: the interpolating polynomial and the Taylor polynomial. Least squares and splines are relegated to Math 448 as an enticement for you to continue the study of numerical methods.

7.1 Polynomial Interpolation

Consider continuous $y = f(x)$ on the closed interval $[a,b]$, and *partition* the interval such that $a \leq x_0 < x_1 < x_2 ... < x_{n-1} < x_n \leq b$. From now on, we will refer to the values x_i of the partition as *nodes*. Finally, let's sample the function $f(x)$ at the nodes, such that $y_i \equiv f(x_i)$. Thus we have, as our starting point, a finite subset (x_i, y_i), $i = 0,1,2,...,n$ of $n+1$ data points that represent function $f(x)$, as shown in Fig. 21.

DEF: Let $P(x)$ be a polynomial of degree at most n. If $P(x_i) = y_i = f(x_i)$ for $i = 0,1,2,...,n$, then $P(x)$ is called an *interpolating polynomial*. If $x_0 \leq x \leq x_n$, then $P(x)$ *interpolates* $f(x)$. If, on the other hand, $x < x_0$ or $x > x_n$, then $P(x)$ *extrapolates* $f(x)$.

REMARKS: Extrapolation is to be avoided like the plague! We'll explain why later.

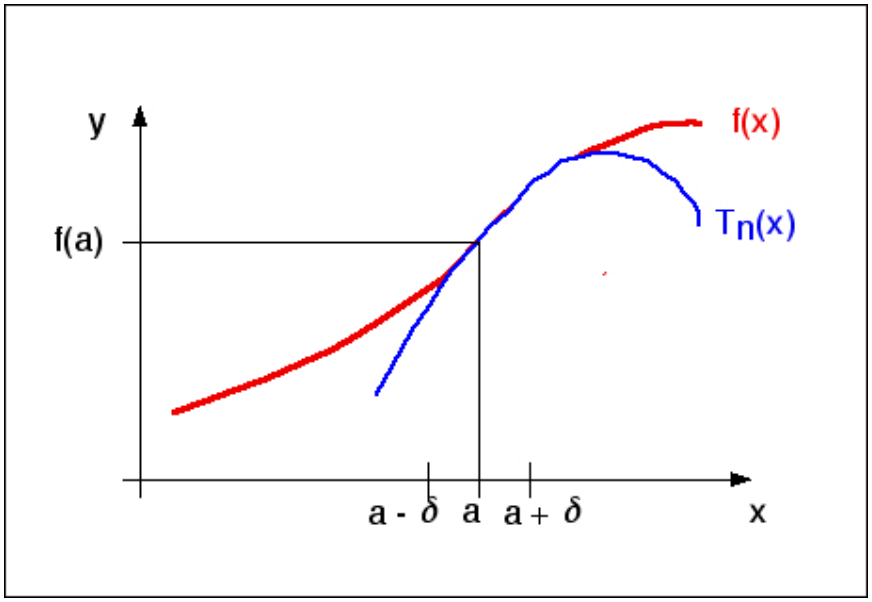


Figure 19: Taylor polynomial.

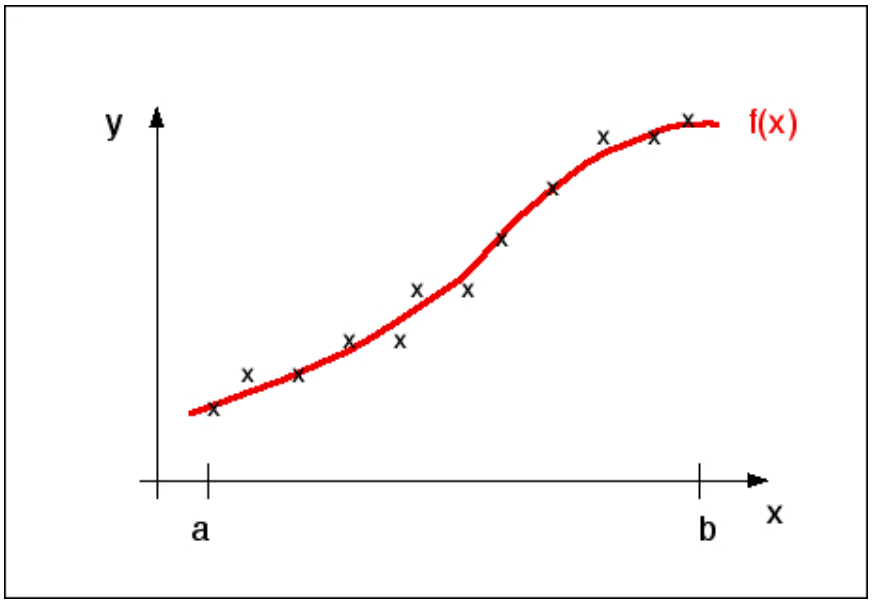


Figure 20: Least squares interpolating polynomial.

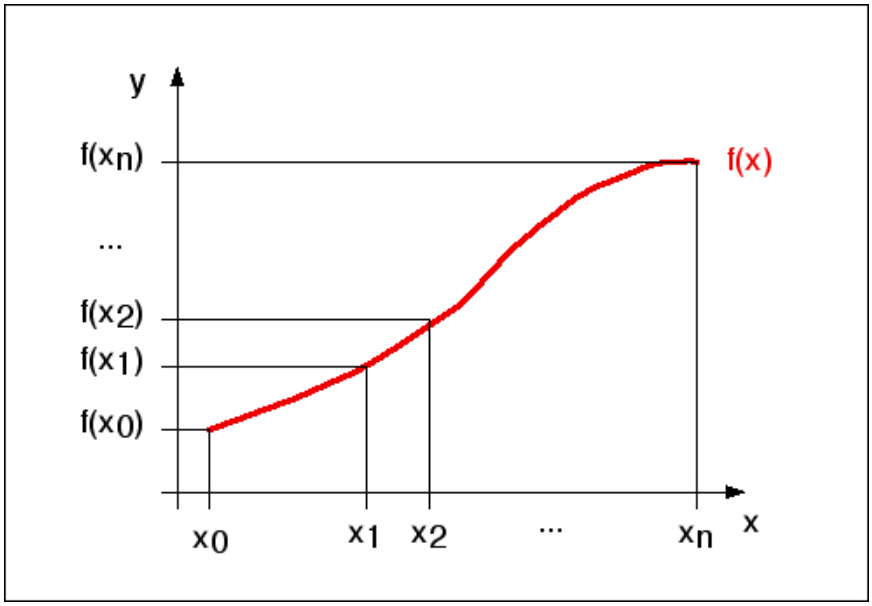


Figure 21: Sampling $f(x)$ at nodal points.

Before we try to find the interpolating polynomial that fits our data, we'd better make sure that such a beast exists. So once again, issues of *existence* and *uniqueness* raise their ugly heads.

7.1.1 Existence and Uniqueness

THEOREM (Existence and Uniqueness of the Interpolating Polynomial): Given $n+1$ distinct points (x_i, y_i) $i = 0, 1, 2, ..., n$, there exists one and only one polynomial $P(x)$ of degree less than or equal to n such that $P(x_i) = y_i$ for all $n+1$ values i.

PROOF: Let $P(x) = a_0 + a_1 x + a_2 x^2 + ... + a_{n-1} x^{n-1} + a_n x^n$. (Note that $P(x)$ is a polynomial, but because we do not require that a_n be non-zero, or any other a_k be non-zero for that matter, we know only that $P(x)$ is of degree less than or equal to n.) Let's require $P(x_0) = y_0$. Thus,

$$P(x_0) = a_0 + a_1 x_0 + a_2 x_0^2 + ... + a_n x_0^n = y_0 \tag{120}$$

Similarly, let's require $P(x_1) = y_1$, in which case

$$P(x_1) = a_0 + a_1 x_1 + a_2 x_1^2 + ... + a_n x_1^n = y_1 \tag{121}$$

Continuing to add requirements, we get to the last requirement

$$P(x_n) = a_0 + a_1 x_n + a_2 x_n^2 + ... + a_n x_n^n = y_n \tag{122}$$

These $n+1$ requirements on $P(x)$ can be written as the following linear system of equations in the $n+1$ unknowns a_i:

$$\begin{bmatrix} 1 & x_0 & x_0^2 & ... & x_0^n \\ 1 & x_1 & x_1^2 & ... & x_1^n \\ ... & & & & \\ 1 & x_n & x_n^2 & ... & x_n^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ ... \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ ... \\ y_n \end{bmatrix}$$

The system above can be written succinctly as $V\vec{a} = \vec{y}$, where $\vec{a}$ is the $n+1$ vector of unknowns, $\vec{y}$ is the $n+1$ vector of sampled function values, and V is an $(n+1) \times (n+1)$ matrix known as the *Vandermonde matrix*. From the theory of linear algebra, we know that $V\vec{a} = \vec{y}$ has a unique solution provided the rows of V are *linearly independent*. Thus, all the constraints imposed upon $P(x)$ can be satisfied uniquely provided the rows of the resulting linear system are independent ($|V| \neq 0$). Notice that the rows of V are each made up of powers of the nodal values. Provided the $n+1$ nodes are distinct, which is assumed by hypothesis, then the rows are independent, the system has a solution $\vec{a}$, and that solution is unique.

You may be wondering why the unique polynomial $P(x)$ is of degree less than or equal to n, rather than simply of degree n. It is because some or all of the coefficients a_i may turn out to be zero. For example, if $f(x)$ were a linear function, then the only (possibly) non-zero coefficients would be a_0 and a_1.

Not only does the Vandermonde matrix approach address the issues of existence and uniqueness, it also provides a way of actually computing the coefficients a_i of the polynomial $P(x)$. Were the Vandermonde matrix method a *good* way of doing this, then the story would end here, and we would move on to the Taylor polynomial. However, it turns out that the Vandermonde matrix is great for theory and terrible as a practical method. When n is even moderately large, say 5, the Vandermonde matrix method magnifies round-off error dramatically in a phenomenon known as *ill-conditioning*. This trait stems from the fact that the Vandermonde matrix is almost *singular*. To find out more about ill-behaved matrices and how to tame them, take Math 448. Suffice it to say here that the Vandermonde matrix method should not be used for $n > 3$ without precautions.

HW: Find the quadratic polynomial that passes through the points (0,3), (1,4), and (2,3) by the Vandermonde matrix method.

Before proceding, we want to caution that the Lagrange and Newton forms of the interpolating polynomial to be addressed in the next two subsections are not different polynomials. They are merely different *forms* of the same interpolating polynomial. For a given set of $n+1$ points (x_i, y_i), as discussed above, the interpolating polynomial of degree $< n$ is *unique*. How we best find that unique polynomial, however, is the issue. Because the Vandermonde method fails in practice, let's examine two other methods of finding the interpolating polynomial.

7.1.2 Lagrange's form of the Interpolating Polynomial

We begin by re-visiting the notion of linear interpolation, which should be familiar to just about everyone who has taken some high-school mathematics.

Linear Lagrange Interpolation

Two distinct points determine a line. Thus, for linear interpolation we consider only two nodes (x_0 and x_1) and their respective function values (y_0 and y_1). Let's refer to the interpolating polynomial as $P_1(x)$, as a recognition that it is at most first-order (linear).

By the point-slope form of a line, we have

$$\begin{aligned} P_1(x) &= y_0 + m(x - x_0) \\ &= y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x - x_0) \\ &= y_0 \left[1 - \frac{(x - x_0)}{(x_1 - x_0)} \right] + y_1 \frac{(x - x_0)}{(x_1 - x_0)} \\ &= y_0 \frac{(x - x_1)}{(x_0 - x_1)} + y_1 \frac{(x - x_0)}{(x_1 - x_0)} \end{aligned} \tag{123}$$

Let's now define two functions, the (linear) Lagrange functions $L_0(x)$ and $L_1(x)$, respectively:

$$L_0(x) \equiv \frac{x - x_1}{x_0 - x_1} \tag{124}$$

$$L_1(x) \equiv \frac{x - x_0}{x_1 - x_0} \tag{125}$$

in which case

$$P_1(x) = y_0 L_0(x) + y_1 L_1(x) \tag{126}$$

That Eq. 126 must be valid is almost, but not quite, obvious. First, consider that, by definition,

$$L_0(x) = \begin{cases} 0 \text{ if } x = x_1 \\ 1 \text{ if } x = x_0 \end{cases} \tag{127}$$

$$L_1(x) = \begin{cases} 0 \text{ if } x = x_0 \\ 1 \text{ if } x = x_1 \end{cases} \tag{128}$$

Now, because $P_1(x)$ is a linear combination of the two linear functions L_0 and L_1, it is (at most) linear. Second, $P_1(x_0) = y_0 L_0(x_0) + y_1 L_1(x_0) = y_0$, and $P_1(x_1) = y_0 L_0(x_1) + y_1 L_1(x_1) = y_1$. Thus, $P_1(x)$ interpolates $f(x)$ and it is at most linear, so it is indeed the polynomial we have been looking for.

EX: Use linear Lagrange interpolation to interpolate $f(x) = \ln(x)$ on $[1/2, 1]$. Here $x_0 = 1/2$ and $x_1 = 1$. Thus,

$$\begin{aligned} L_0(x) &= \frac{(x-1)}{(\frac{1}{2} - 1)} = -2(x-1) \\ L_1(x) &= \frac{(x - \frac{1}{2})}{(1 - \frac{1}{2})} = 2x - 1 \end{aligned}$$

Moreover,

$$\begin{aligned} y_0 &= f(x_0) = f(\frac{1}{2}) = \ln(\frac{1}{2}) = -\ln(2) \\ y_1 &= f(x_1) = f(1) = \ln(1) = 0 \end{aligned}$$

Thus

$$\begin{aligned} P_1(x) &= -\ln(2)(-2)(x-1) + 0(2x-1) \\ \Rightarrow P_1(x) &= 2\ln 2(x-1) \end{aligned}$$

x	$f(x) = \ln(x)$	$P_1(x)$	$e_1(x)$
0.5	-0.693147...	-0.693147...	0.0
0.6	-0.510825...	-0.554517...	0.043692...
0.7	-0.356674...	-0.415888...	0.059213...
0.8	-0.223143...	-0.277258...	0.054115...
0.9	-0.105360...	-0.138629...	0.033268...
1.0	0.0	0.0	0.0

Table 10: Error of linear Langrange interpolation.

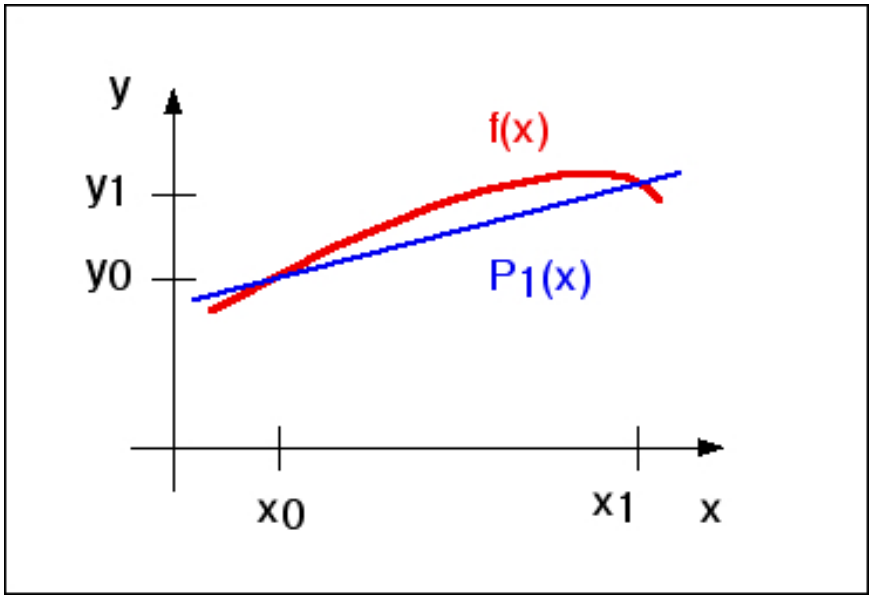


Figure 22: Linear Lagrange interpolation.

Table 10 compares $f(x) = \ln(x)$ with its linear interpolant $P_1(x)$ for several equally spaced values of x between 1/2 and 1.

The last column of Table 10 provides the interpolation error $e_1(x) = f(x) = P_1(x)$. As expected, the error is zero at the nodes, because the interpolating polynomial is required to exactly match the function at these values. In general, errors tend to be worst between adjacent nodes, as is the case here. The worst absolute error in the Table occurs at $x = 0.7$, approximately midway between the two nodes.

Admittedly the error properties above are "nothing to write home about." In fact, the worst *relative* error in the Table is nearly 30 percent at $x = 0.9$! Surely we can do better than this. Any idea how? Two possibilities should be relatively obvious: 1) move the nodes closer together, and/or 2) use higher order interpolants. We now consider the latter of these two options.

Quadratic Lagrange Interpolation

In this case, we consider three nodes in order to fit a quadratic through three distinct points (x_0, y_0),

(x_1, y_1), and (x_2, y_2). Rather than reinvent the wheel, can we mimic what we did for linear interpolation and simply adapt appropriately? That is, let's try to construct $P_2(x)$ such that

$$P_2(x) = y_0 L_0(x) + y_1 L_1(x_1) + y_2 L_2(x) \tag{129}$$

In this case, the Lagrange functions L_0, L_1, and L_2 should each be quadratic and should satisfiy the following constraint equations:

$$\begin{aligned} L_0(x) &= \begin{cases} 0 \text{ if } x = x_1 \text{ or } x = x_2 \\ 1 \text{ if } x = x_0 \end{cases} \\ L_1(x) &= \begin{cases} 0 \text{ if } x = x_0 \text{ or } x = x_2 \\ 1 \text{ if } x = x_1 \end{cases} \\ L_2(x) &= \begin{cases} 0 \text{ if } x = x_0 \text{ or } x = x_1 \\ 1 \text{ if } x = x_2 \end{cases} \end{aligned} \tag{130}$$

The conditions above are all satisfied by the following definitions:

$$\begin{aligned} L_0(x) &\equiv \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} \\ L_1(x) &\equiv \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} \\ L_2(x) &\equiv \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} \end{aligned} \tag{131}$$

in which case Eq. 129 clearly holds, because $P_2(x_0) = y_0$, $P_2(x_1) = y_1$, and $P_2(x_2) = y_2$ by design.

EX: Appoximate $\cos(x)$ on $[-\frac{\pi}{4}, +\frac{\pi}{4}]$ by quadratic Lagrange interpolation using 3 equally spaced nodes.

In general nodes need not be equally spaced. However, in this case the problem specification requires this. Thus $x_0 = -\frac{\pi}{4}$, $x_1 = 0$, and $x_2 = +\frac{\pi}{4}$. Accordingly $y_0 = y_2 = \frac{\sqrt{2}}{2}$ and $y_1 = 1$. The Lagrange functions involve only the nodal values. That is,

$$\begin{aligned} L_0(x) &= \frac{(x-0)(x-\frac{\pi}{4})}{(-\frac{\pi}{4}-0)(-\frac{\pi}{4}-\frac{\pi}{4})} = \frac{8}{\pi^2} x(x-\frac{\pi}{4}) \\ L_1(x) &= \frac{[x-(-\frac{\pi}{4})](x-\frac{\pi}{4})}{[0-(-\frac{\pi}{4})](0-\frac{\pi}{4})} = -(\frac{4}{\pi})^2 [x^2 - (\frac{\pi}{4})^2] \\ L_2(x) &= \frac{[x-(-\frac{\pi}{4})](x-0)}{[\frac{\pi}{4}-(-\frac{\pi}{4})](+\frac{\pi}{4}-0)} = \frac{8}{\pi^2} x(x+\frac{\pi}{4}) \end{aligned} \tag{132}$$

The following list contains the ouput of a computer program that makes use of Eq. 129 and Eq. 132 to create a table of interpolated values for the cosine function. The 21 values are equally spaced in x on $[-\pi/4, \pi/4]$. The fourth column gives the interpolation error $e_2(x) = \cos(x) - P_2(x)$. Once again, by design, interpolation is exact at the nodes. The absolute error is at its worst between nodes, but a bit closer to the ends of the interval than the middle. We'll find out why this is so later.

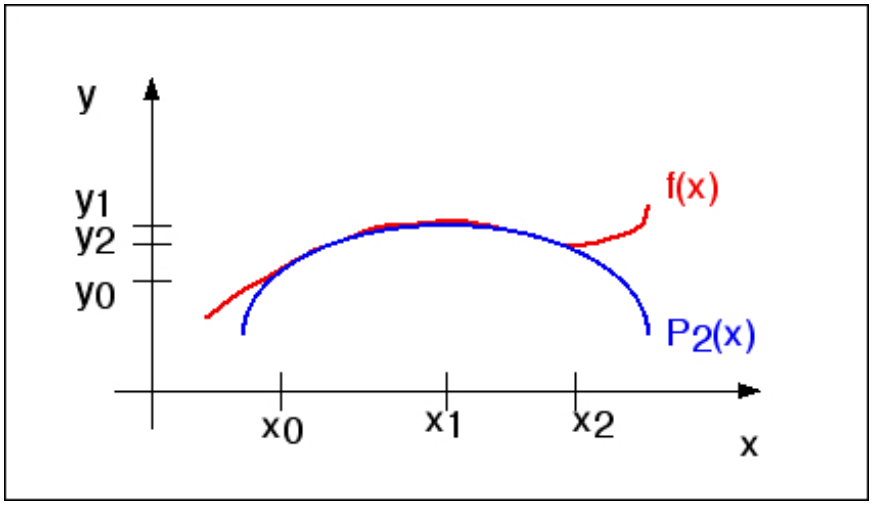


Figure 23: Quadratic Lagrange interpolation.

```
  x          Cos(x)     P2(x)      e2(x)

-0.785398   0.707107   0.707107   0.0000E+00
-0.706858   0.760406   0.762756  -0.2350E-02
-0.628319   0.809017   0.812548  -0.3531E-02
-0.549779   0.852640   0.856482  -0.3842E-02
-0.471239   0.891007   0.894558  -0.3552E-02
-0.392699   0.923880   0.926777  -0.2897E-02
-0.314159   0.951057   0.953137  -0.2081E-02
-0.235619   0.972370   0.973640  -0.1270E-02
-0.157080   0.987688   0.988284  -0.5959E-03
-0.078540   0.996917   0.997071  -0.1537E-03
 0.000000   1.000000   1.000000   0.0000E+00
 0.078540   0.996917   0.997071  -0.1537E-03
 0.157080   0.987688   0.988284  -0.5959E-03
 0.235619   0.972370   0.973640  -0.1270E-02
 0.314159   0.951057   0.953137  -0.2081E-02
 0.392699   0.923880   0.926777  -0.2897E-02
 0.471239   0.891007   0.894558  -0.3552E-02
 0.549779   0.852640   0.856482  -0.3842E-02
 0.628319   0.809017   0.812548  -0.3531E-02
 0.706858   0.760406   0.762756  -0.2350E-02
 0.785398   0.707107   0.707107   0.0000E+00
```

Lagrange Interpolation of Degree *n*

It should be relatively clear by now that Lagrange interpolation can be implemented at arbitrary degree *n*. With higher order, errors tend to diminish to a point, but not, as you might think, indefinitely.

For Lagrange interpolation of degree n, we seek

$$\begin{aligned} P_n(x) &= y_0 L_0(x) + y_1 L_1(x_1) + ... + y_n L_n(x) \\ &= \sum_{i=0}^{n} y_i L_i(x) \end{aligned} \tag{133}$$

where each $L_i(x)$ is a Lagrange polynomial of degree n that satisfies the constraints

$$L_i(x_j) = \delta_{ij} = \begin{cases} 0 \text{ if } i \neq j \\ 1 \text{ if } i = j \end{cases} \tag{134}$$

where δ_{ij} is the Kronecker delta function. It is readily shown that the following form of $L_i(x)$ satisfies the constraints:

$$L_i(x) = \frac{\Pi_{j=0\ (j\neq i)}^{n}(x - x_j)}{\Pi_{j=0\ (j\neq i)}^{n}(x_i - x_j)} \tag{135}$$

Recall that Π means "product." Thus, the numerator of each $L_i(x)$ is the product of n linear factors, each of the form $(x - x_j)$, so each $L_i(x)$ is a polynomial of degree n. The unusual notation $(j \neq i)$ signals that one linear factor is omitted for each i, namely the factor $x - x_i$.

Lagrange interpolation is a practical method that does not suffer from the problems with round-off error that plague the Vandermonde matrix method. Nevertheless, Lagrange interpolation suffers from two shortcomings, the second of which is quite serious.

1. First, the Lagrange method is not very computationally efficient. For degree n, there are n Lagrange functions $L_i(x)$, each of which is formed by the product of n linear factors in the numerator and n similar constant factors in the denominator. Each factor requires two operations: a subtraction and a multiplication. Thus, there are approximately $4n$ operations per $L_i(x)$, or $4n^2$ operations in total to evaluate the L_i's *once*, and another $2n$ operations to sum them up for $P_n(x)$.

2. Second, Lagrange interpolation is particularly bad if you need to add an additional node to form $P_{n+1}(x)$, for example, for better accuracy of the interpolant. With the Lagrange method, you must start over if another node is added.

In the best of all worlds, the method of interpolation would allow us to build up the interpolating polynomial one order at a time until the desired accuracy was attained. That is, given $P_n(x)$, $P_{n+1}(x)$ could be constructed by simply adding the next higher order term.

Once again, we have Isaac Newton to thank for developing an ideal method: Newton's divided-difference method of interpolation. (This is the same Newton, by the way, who developed differential and integral calculus, the theory of gravition, the theory of optics, the binomial theorem, Newton's method for rootfinding, and much more. He did all of this during a two-year period, 1665-1666, when he was 22-24 years of age and working in isolation at home. The bubonic plague had closed many European universities, including Cambridge, where Newton had been a student. Pretty humbling to think that one human being could accomplish so much of importance to the world in such a short time.)

7.1.3 Newton's Form of the Interpolating Polynomial

Newton's method of interpolation begins with the question: Can we construct interpolating polynomial $P_n(x)$ such that $P_{n+1}(x)$ is readily obtained if higher order is necessary?

The Newton Polynomial

Consider the n+1 points $[x_i, f(x_i)],\ i = 0, 1, ..., n$, whose x values are called nodes or *centers*. Let

$$\begin{aligned} P_0(x) &= a_0 \\ P_1(x) &= a_0 + a_1(x - x_0) \\ &= P_0(x) + a_1(x - x_0) \\ P_2(x) &= a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) \\ &= P_1(x) + a_2(x - x_0)(x - x_1) \\ P_3(x) &= a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + a_3(x - x_0)(x - x_1)(x - x_2) \\ &= P_2(x) + a_3(x - x_0)(x - x_1)(x - x_2) \end{aligned} \tag{136}$$

Notice, for example, that $P_2(x)$ is $P_1(x)$ plus a new quadratic term, and that $P_3(x)$ is $P_2(x)$ plus a new cubic term. In general,

$$P_n(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + ... + a_n(x - x0)(x - x_1)(x - x_2)...(x - x_{n-1}) \tag{137}$$

and

$$P_{n+1}(x) = P_n(x) + a_{n+1}(x - x_0)(x - x_1)(x - x_2)...(x - x_{n-1})(x - x_n) \tag{138}$$

That is, each polynomial of the next higher order is constructed by adding a higher order term to the polynomial that is its immediate predecessor. (**WARNING:** The last term of $P_n(x)$ never involves node x_n. One node, the last, is always left out. If you try to include it, your polynomial will be off by one degree.)

Given n points, our main job now is to find coefficients $a_i,\ i = 0, 1, 2, ..., n$ so that $P_n(x)$ in Newton form interpolates $f(x)$. Before we turn to that somewhat daunting task, let's consider how best to evaluate a polynomial in Newton form. For specificity, consider the Newton polynomial of degree 4, namely,

$$\begin{aligned} P_4(x) &= a_0 \\ &+ a_1(x - x_0) \\ &+ a_2(x - x_0)(x - x_1) \\ &+ a_3(x - x_0)(x - x_1)(x - x_2) \\ &+ a_4(x - x_0)(x - x_1)(x - x_2)(x - x_3) \end{aligned} \tag{139}$$

which an be written in *nested* form as

$$P_4(x) = a_0 + (x - x_0)\{a_1 + (x - x_1)[|a_2 + (x - x_2)[a_3 + (x - x_3)a_4]|]\} \tag{140}$$

To evaluate $P_4(x)$ in nested form, from the inner nest out, we would perform the following steps, where s is a scalar temporary variable, and x is considered specified:

$$s \leftarrow a_4$$

$$\begin{aligned} s &\leftarrow s(x-x_3)+a_3 \\ s &\leftarrow s(x-x_2)+a_2 \\ s &\leftarrow s(x-x_1)+a_1 \\ s &\leftarrow s(x-x_0)+a_0 \end{aligned} \tag{141}$$

For arbitrary n, $P_n(x)$ could be evaluated in similar fashion, which leads to ALGORITHM 7.1, a generalization of Horner's Method (ALG 1.1).

ALGORITHM 7.1: Newton Polynomial Evalution by Generalized Horner's method

input order n
input $n+1$ *coefficients* a_i, $i = 0,1,...,n$
input $n+1$ *nodes (centers)* x_i, $i = 0,1,...,n$
input value of x
$s \leftarrow a_n$ *(initialize temporary storage register)*

repeat for $i = n-1, n-2, ..., 0$ *(descending order)*
| $s \leftarrow s \cdot (x - x_i) + a_i$
| —

output s (which contains value of $P_n(x)$*)*

Newton's Divided-Difference Table

Now to find the coefficients a_i. If we have but one node x_0, we must require $P_0(x_0) = f(x_0)$. But the zeroth-order Newton polynomial is simply a constant, namely $P_0(x) = a_0$ (Eq. 136). Thus, $a_0 = f(x_0)$. Substituting this value into the form of the linear (first-order) Newton polynomial, we have

$$P_1(x) = f(x_0) + a_1(x - x_0) \tag{142}$$

To find a_1, we impose the new constraint

$$\begin{aligned} P_1(x_1) = f(x_1) &\Rightarrow f(x_0) + a_1(x_1 - x_0) = f(x_1) \\ &\Rightarrow a_1 = \frac{f(x_1) - f(x_0)}{(x_1 - x_0)} \end{aligned} \tag{143}$$

We now substitute the values of a_0 and a_1 just found into $P_2(x)$ of Eq. 136, to obtain

$$P_2(x) = f(x_0) + \frac{f(x_1) - f(x_0)}{(x_1 - x_0)}(x - x_0) + a_2(x - x_0)(x - x_1) \tag{144}$$

and to find a_2, we impose the additional constraint that $P_2(x_2) = f(x_2)$, in which case

$$f(x_0) + \frac{f(x_1) - f(x_0)}{(x_1 - x_0)}(x_2 - x_0) + a_2(x_2 - x_0)(x_2 - x_1) = f(x_2) \tag{145}$$

The algebra now starts to get tedious. Solving the equation immediately above for a_2 yields

$$\begin{aligned} a_2 &= \frac{\frac{f(x_2)-f(x_0)}{(x_2-x_0)} - \frac{f(x_1)-f(x_0)}{(x_1-x_0)}}{x_2-x_1} \\ &= \frac{[f(x_2)-f(x_0)](x_1-x_0)-[f(x_1)-f(x_0)](x_2-x_0)}{(x_2-x_0)(x_1-x_0)(x_2-x_1)} \end{aligned} \tag{146}$$

We now wish to manipulate the form above by writing $(x_2-x_0)=[(x_2-x_1)-(x_1-x_0)]$ to obtain

$$\begin{aligned} a_2 &= \frac{[f(x_2)-f(x_0)](x_1-x_0)-[f(x_1)-f(x_0)][(x_2-x_1)-(x_1-x_0)]}{(x_2-x_0)(x_1-x_0)(x_2-x_1)} \\ &= \frac{[f(x_2)-f(x_0)-f(x_1)+f(x_0)](x_1-x_0)-[f(x_1)-f(x_0)](x_2-x_1)}{(x_2-x_0)(x_1-x_0)(x_2-x_1)} \\ &= \frac{\left[\frac{f(x_2)-f(x_1)}{x_2-x_1}\right] - \left[\frac{f(x_1)-f(x_0)}{x_1-x_0}\right]}{x_2-x_0} \end{aligned} \tag{147}$$

We admit, the process is beginning to look hopeless, especially when we consider the daunting process of trying to solve for a_3 and higher order coefficients. However, it turns out that all we need is a bit of organization, bookkeeping if you will.

Define

$$f[x_0] \equiv f(x_0);\;\; f[x_1] \equiv f(x_1);\;\; f[x_2] \equiv f(x_2);\;\; \text{etc.} \tag{148}$$

We now define the *first divided difference* as follows:

$$f[x_0,x_1] \equiv \frac{f[x_1]-f[x_0]}{x_1-x_0};\;\; f[x_1,x_2] \equiv \frac{f[x_2]-f[x_1]}{x_2-x_1};\;\; \text{etc.} \tag{149}$$

and the *second divided difference* as the difference of first divided differences such that

$$f[x_0,x_1,x_2] \equiv \frac{f[x_1,x_2]-f[x_0,x_1]}{x_2-x_0};\;\; f[x_1,x_2,x_3] \equiv \frac{f[x_2,x_3]-f[x_1,x_2]}{x_3-x_1};\;\; \text{etc.} \tag{150}$$

Similarly, each successive divided difference is a quotient involving previous lower-order divided differences. For example,

$$f[x_0,x_1,x_2,x_3] \equiv \frac{f[x_1,x_2,x_3]-f[x_0,x_1,x_2]}{x_3-x_0};\;\; \text{etc.} \tag{151}$$

The process is easily organized in tabular form, as shown in Table 11 for $n=4$.

The table is completed by beginning with the first two columns on the left and then working across the columns from left to right. For example, the values in the third column are second divided differences, which are computed from the first divided differences in the second column, and so forth. The only tricky part of the computation is figuring out what the divisors should be. For example, the fifth column, which contains the third divided differences, each of which involves four nodes. In particular, the divided difference $f[x_0,x_1,x_2,x_3]$ involves the four nodes beginning with x_0 and ending with x_3. Thus, the divisor is x_3-x_0. The divided-difference table is a bit like a geneological table of ancestry. Each new generation is related to many generations of ancestors. The divisor must span all generations.

x_k	$f[x_k]$	$f[\ ,\ ]$	$f[\ ,\ ,\ ]$	$f[\ ,\ ,\ ,\ ]$	$f[\ ,\ ,\ ,\ ,\ ]$
x_0	$\underline{f(x_0)}$				
x_1	$f(x_1)$	$\underline{f[x_0,x_1]}$			
x_2	$f(x_2)$	$f[x_1,x_2]$	$\underline{f[x_0,x_1,x_2]}$		
x_3	$f(x_3)$	$f[x_2,x_3]$	$f[x_1,x_2,x_3]$	$\underline{f[x_0,x_1,x_2,x_3]}$	
x_4	$f(x_4)$	$f[x_3,x_4]$	$f[x_2,x_3,x_4]$	$f[x_1,x_2,x_3,x_4]$	$\underline{f[x_0,x_1,x_2,x_3,x_4]}$

Table 11: Newton's Divided-Difference Table for $n = 4$.

Where are the coefficients a_i? They are lurking in the divided difference table. Recall that $a_0 = f[x_0]$, $a_1 = \frac{f[x_1]-f[x_0]}{x_1-x_0} = f[x_0,x_1]$, and $a_2 = \frac{f[x_1,x_2]-f[x_1,x_0]}{x_2-x_0} = f[x_0,x_1,x_2]$. Yes! You are right. The coefficients a_i reside along the diagonal of the divided-difference table!

Let's illustrate how it all works with a real numerical example.

EX: Consider the function $f(x) = 3 \cdot 2^x$ on the interval $[-1,3]$. Let $n = 4$ and use the Newton form of the interpolating polynomials to interpolate at $x = 1.5$. Use five equally spaced nodes (but keep in mind that the nodes need not be either equally spaced or even in numerical order!). Thus our data is

i	x_i	$f(x_i)$
0	-1.0	1.5
1	0.0	3.0
2	1.0	6.0
3	2.0	12.0
4	3.0	24.0

Note that $x = 1.5$ is not a node, so interpolation is required to obtain a value for $x = 1.5$. We now transfer the starting data to our triangular array (Table 12) and begin "walking across the columns" in succession.

x_k	$f[x_k]$	$f[\ ,\ ]$	$f[\ ,\ ,\ ]$	$f[\ ,\ ,\ ,\ ]$	$f[\ ,\ ,\ ,\ ,\ ]$
-1.0	$\underline{a_0 = 1.5}$				
0.0	3.0	$\underline{a_1 = 1.5}$			
1.0	6.0	3.0	$\underline{a_2 = 0.75}$		
2.0	12.0	6.0	1.5	$\underline{a_3 = 0.25}$	
3.0	24.0	12.0	3.0	0.50	$\underline{a_4 = 0.0625}$

Table 12: Numerical example of Newton's divided-difference table.

Using these coefficients, we construct the Newton polynomials of degrees 0 to 4, each of which is built upon the previous.

$P_0(x) \quad = \quad 1.5$

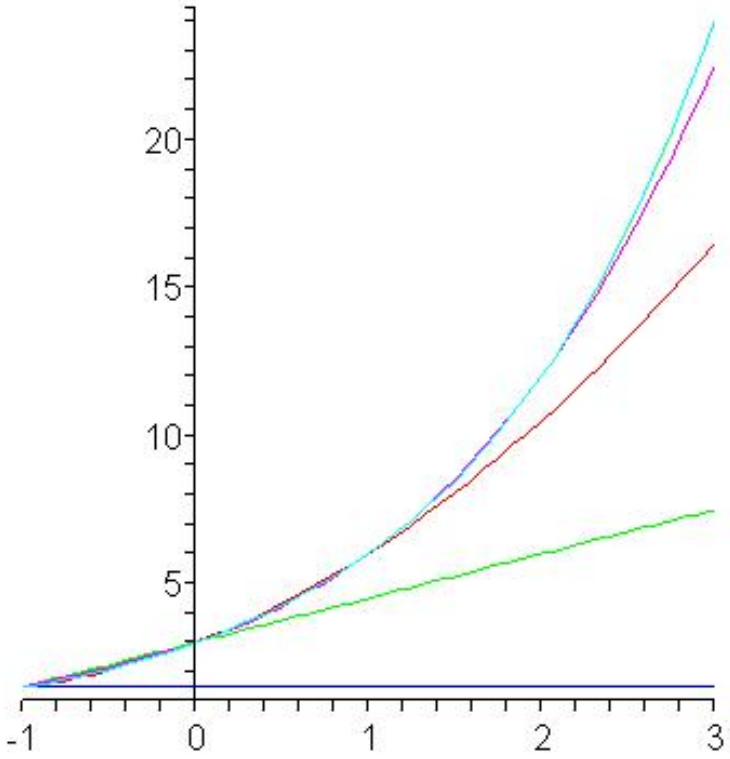


Figure 24: $P_0(x)$ (blue), $P_1(x)$ (green), $P_2(x)$ (red), $P_3(x)$ (magenta), $P_4(x)$ (cyan) compared with $f(x) = 3 \cdot 2^x$ (yellow).

$$\begin{aligned} P_1(x) &= 1.5 + 1.5(x+1) \\ P_2(x) &= 1.5 + 1.5(x+1) + 0.75(x+1)(x) \\ P_3(x) &= 1.5 + 1.5(x+1) + 0.75(x+1)(x) + 0.25(x+1)(x)(x-1) \\ P_4(x) &= 1.5 + 1.5(x+1) + 0.75(x+1)(x) + 0.25(x+1)(x)(x-1) + 0.0625(x+1)(x)(x-1)(x-2) \end{aligned} \tag{152}$$

See Figure 24 for a comparison of the above Newton Polynomials with the function.

Finally, let's evaluate each of these polynomials at $x = 1.5$ (using modified Horner's method, of course). Note that the exact value to 10 significant digits is $f(1.5) = 8.485281375$. Note also that the

n	$P_n(1.5)$	$e_n(1.5) = f(1.5) - P_n(1.5)$
0	1.5	6.985281374
1	5.25	3.235281374
2	8.0625	0.422781374
3	8.53125	-0.045968626
4	8.47265625	0.012625124

constant, linear, and quadratic interpolating polynomials return terrible results, as shown by the error in

the third column of the table, but that the relative error of the cubic and quartic polynomials is small, the latter being on the order of one-tenth of a percent.

The beauty of Newton's method of interpolation is that, if the error of the interpolant is unsatisfactory, a new node can be added. This requires only the appendage of another row at the bottom of the divided-difference table. Previous rows remain unchanged. Moreover, the new node need not be in numerical order, nor in fact, must any of the nodes be in numerical order. Newton's method therefore is practical, accurate, and efficient. What more can you ask of an algorithm?

Speaking of algorithms, here is pseudocode for constructing Newton's divided-difference table.

ALGORITHM 7.2: Newton's Divided-Difference Table

input order n
input $n+1$ *nodes* x_i, $i = 0, 1, ..., n$
input $n+1$ *function values* y_i, $i = 0, 1, ..., n$

for $i = 0, 1, ..., n$
| $d_{i,0} \leftarrow y_i$ *(initialize table)*
| —

for $j = 1, 2, ..., n$ *(j is column index)*
|
| *for* $k = j, j+1, ..., n$ *(k is row index)*
| | $d_{kj} \leftarrow (d_{k,j-1} - d_{k-1,j-1})/(x_k - x_{k-j})$
| |_
|—

Note that Newton's divided-difference table requires the use of an $(n+1) \times (n+1)$ matrix D dimensioned from 0 to n. Upon completion of the algorithm, the a_i's are stored along the diagonal of matrix D. That is, $a_i = d_{ii}$. In actuality, by overwriting, the algorithm can be re-written so as to require only an $n+1$-vector $\vec{d}$, which is used both for computing and storing the values a_i.

The inner assignment statement of the nested loop is executed $\frac{n(n+1)}{2}$ times, and there are 3 operations (two subtracts and one divide) within the statement. Thus, the total count is approximately $\frac{3}{2}n^2$ operations. The operation count for the generalized Horner method (ALG 7.1) to evaluate $P_n(x)$ is only $3n$ operations. In comparing these numbers with the counts for the Lagrange method, we find that Newton's method has a considerable computational advantage over Lagrange interpolation.

In closing, we remind the reader that, given the same input data (and ignoring differences in round-off errors), the Vandermonde matrix method, the Lagrange method, and Newton's method, all yield that same interpolating polynomial in different forms. Why so? Because the polynomial that interpolates is unique.

Our next topic, the Taylor polynomial, is a polynomial of a very different stripe.

7.2 Taylor Polynomials

As shown in Fig. 19, we desire to approximate $f(x)$ on an interval I, but we consider only a single node $x = a$ in the interior of I. The general form of a Taylor polynomial of degree n is

$$T_n(x) = a_0 + a_1(x-a) + a_2(x-a)^2 + \ldots + a_n(x-a)^n \tag{153}$$

Note that the Taylor polynomial looks like the Newton polynomial with all nodes collapsed into the single node $x = a$. Our job is to find the coefficients a_i so that $T_n(x)$ "looks like" $f(x)$.

Taylor polynomial approximants, like Newton polynomials, are built up one order at a time. We begin with the Taylor polynomial of order zero and constrain it to make $T_0(x) = f(x)$ at $x = a$. Thus,

$$T_0(x) = f(a) \quad \Rightarrow a_0 = f(a) \tag{154}$$

We now add a linear term to T_0 to get T_1.

$$T_1(x) = f(a) + a_1(x-a) \tag{155}$$

To find a_1, we impose the new constraint $T_1'(a) = f'(a)$, in which case

$$\begin{aligned} a_1 &= f'(a) \Rightarrow \\ T_1(x) &= f(a) + f'(a)(x-a) \end{aligned} \tag{156}$$

We could continue in this manner, but let it suffice to say that the quadratic Taylor polynomial is

$$T_2(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2}(x-a)^2 \tag{157}$$

where $a_2 = \frac{f''(a)}{2}$. It is easily verified that $T_2(x)$ satisfies $T_2(a) = f(a)$, $T_2'(a) = f'(a)$, and $T_2''(a) = f''(a)$. Thus, each new constraint requires the next Taylor polynomial in succession to match another derivative of $f(x)$ at $x = a$. If it looks like a duck, walks like a duck, and quacks like a duck, it must be a duck. Similarly, if $T_n(x)$ has the same value as $f(x)$ at $x = a$, the same derivative as $f(x)$ at $x = a$, the same second derivative as $f(x)$ at $x = a$, and so on, then it must look a lot like $f(x)$. By requiring the third derivative of the Taylor polynomial to match that of $f(x)$ at $x = a$, we get $T_3(x)$, namely

$$T_3(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2}(x-a)^2 + \frac{f'''(a)}{6}(x-a)^3 \tag{158}$$

Once we see the pattern, we can safely generalize.

$$\begin{aligned} T_n(x) &= f(a) + f'(a)(x-a) + \frac{f''(a)}{2}(x-a)^2 + \frac{f'''(a)}{6}(x-a)^3 + \ldots + \frac{f^{(n)}(a)}{n!}(x-a)^n \\ &= a_0 + a_1(x-a) + a_2(x-a)^2 + a_3(x-a)^3 + \ldots + a_n(x-a)^n \\ &= \sum_{k=0}^{n} a_k(x-a)^k \end{aligned} \tag{159}$$

where $a_k = \frac{f^{(k)}(a)}{k!}$. (Recall the mathematical conventions that $0! \equiv 1$ and $f^{(0)}(a) \equiv f(a)$.)

The power of the Taylor polynomial is best illustrated by an example.

EX: Let $f(x) = \sin(x)$ and $a = 0$. Find the Taylor polynomial approximants of orders 5, 7, and 9.

It suffices to find $T_9(x)$ and then to truncate it at 5 and 7 terms to obtain $T_5(x)$ and $T_7(x)$, respectively. Taylor polynomials are best found with the help of a table.

k	$f^{(k)}(x)$	$f^{(k)}(a)$
0	$f(x) = \sin(x)$	$\sin(0) = 0$
1	$\cos(x)$	$\cos(0) = 1$
2	$-\sin(x)$	$-\sin(0) = 0$
3	$-\cos(x)$	$-\cos(0) = -1$
4	$\sin(x)$	$\sin(0) = 0$
5	$\cos(x)$	$\cos(0) = 1$
6	$-\sin(x)$	$-\sin(0) = 0$
7	$-\cos(x)$	$-\cos(0) = -1$
8	$\sin(x)$	$\sin(0) = 0$
9	$\cos(x)$	$\cos(0) = 1$

Thus,

$$\begin{aligned} T_9(x) &= 0 + 1(x-0) \\ &+ \frac{0}{2!}(x-0)^2 - \frac{1}{3!}(x-0)^3 \\ &- \frac{0}{4!}(x-0)^4 + \frac{1}{5!}(x-0)^5 \\ &+ \frac{0}{6!}(x-0)^6 - \frac{1}{7!}(x-0)^7 \\ &- \frac{0}{8!}(x-0)^8 + \frac{1}{9!}(x-0)^9 \end{aligned} \tag{160}$$

You may be wondering what possessed us to leave all the zeroes in the expansion of $T_9(x)$ above. It is to remind you that, in general, these quantities are *non-zero*. They just happen to vanish for this specific example. That said, cleaning up the result above and truncating at degrees 5, 7, and 9, respectively, we obtain

$$\begin{aligned} T_5(x) &= x - \frac{x^3}{3!} + \frac{x^5}{5!} \\ T_7(x) &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \\ T_9(x) &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} \end{aligned} \tag{161}$$

Finally, let's compare $T_5(\frac{\pi}{4})$, $T_7(\frac{\pi}{4})$, and $T_9(\frac{\pi}{4})$ to see how well each compares with the exact value $f(\frac{\pi}{4}) = \sin(\frac{\pi}{4}) = \frac{\sqrt{2}}{2}$. The results are summarized in Table 13 below.

n	$f(x)$	$T_n(x)$	$e_n(x) = f(x) - T_n(x)$
5	0.7071067812	0.707143046	3.62646×10^{-5}
7	0.7071067812	0.707106781	3.11611×10^{-7}
9	0.7071067812	0.707106783	-1.75×10^{-9}

Table 13: Error of Taylor polynomial approximation of $f(x) = \sin(x)$ at $x = \pi/4$.

The result of Table 13 is really quite amazing. The $\sin(x)$ is approximated to nearly nine significant digits of accuracy on $[\pi/4, \pi/4]$ by a polynomial with only five non-zero terms!

How do we know in advance how many terms of the Taylor polynomial to keep for a desired level of accuracy? Thus far, we don't. But that will change in the next subsection.

7.3 Truncation Error and Taylor's Remainder Theorem

In the study of infinite series in Math 236, we learned that the process of approximation above can be taken to the infinite limit. But first, a review of terminology. Whereas a *polynomial* has a *finite* number of terms of the form $a_k(x-a)^k$, a *power series* has an *infinite number* of terms of the same form. If the process above is taken to the infinite limit, the Taylor polynomial morphs into the *Taylor series*. For example, the Taylor series for $\sin(x)$ (about $a = 0$) is

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \ldots \tag{162}$$

Unlike the Taylor polynomial, which is an approximation, the Taylor series above is *exact* for all $x \in (-\infty, \infty)$. Exact Taylor series can be found for any function that is infinitely differentiable (but the interval of convergence may not be infinite). However, the Taylor series is impractical as a numerical tool. No matter how fast the computer, it cannot possibly sum an infinity of terms in finite time. At some point we must chop off the infinite tail of the Taylor series and be content with the finite polynomial that remains. The \$64,000 question is: How many terms do we need for a desired level of accuracy? Fortunately there is a theorem that (nearly) answers our question.

Taylor's Remainder THEOREM: Let $f(x)$ be $n+1$ times differentiable on some interval I containing $x = a$, and let $T_n(x)$ be the n-order Taylor polynomial approximant of $f(x)$ about $x = a$. For each $x \in I$ there exists some ξ between a and x such that

$$e_n^T(x) \equiv f(x) - T_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x-a)^{n+1} \tag{163}$$

where the superscript T is inserted to remind the reader that the error formula applies only to the Taylor polynomial. How many of you really appreciate the power of this beautiful theorem (whose proof must wait until Math 448)? Taylor's Remainder Theorem tells one *exactly* how much error is incurred if the Taylor series is truncated after n terms. We can thus use the remainder theorem in reverse to determine the number of terms n necessary for a desired level of accuracy in the Taylor-polynomial approximation

of $f(x)$. How cool is that? There is a fly in the ointment, however. The quantity ξ above is somewhat mysterious. All we know is that ξ lives between a and x. What value should we use?

It turns out that a less exact form of Taylor's remainder term is actually more useful to us. Let's derive this form by taking the absolute value of both sides of Eq. 163.

$$\begin{aligned} |e_n^T(x)| &= \left| \frac{f^{(n+1)}(\xi)}{(n+1)!}(x-a)^{n+1} \right| \\ &= \frac{1}{(n+1)!}|f^{(n+1)}(\xi)||x-a|^{n+1} \\ &\leq \frac{M_{n+1}}{(n+1)!}|x-a|^{n+1} \end{aligned} \tag{164}$$

where M_{n+1} is an upper bound on the absolute value of the $n+1$ derivative of $f(x)$ on the interval of interest. That is

$$|f^{(n+1)}(x)| \leq M_{n+1} \quad x \in I \tag{165}$$

By replacing the $n+1$ derivative in Eq. 164 with a bound we get around the thorny problem of not knowing what value to use for ξ. However, there is a price to be paid. Whereas Eq. 163 is an equality, Eq. 164 is now an inequality that provides a bound on the error rather than an estimate of the error. In general, finding a suitable value for M_{n+1} is itself a thorny problem. However, for some functions, it is trivial. Consider that both the sine and cosine functions are bounded by unity. That is, $|\cos(x)| \leq 1$ and $|\sin(x)| \leq 1$ for all $x \in \mathcal{R}$. But all derivatives of sines and cosines are themselves sines or cosines. Consequently, for $f(x) = \sin(x)$ or $f(x) = \cos(x)$, an appropriate bound to use is $M_{n+1} = 1$.

EX: What degree n of Taylor polynomial approximation of $f(x) = \sin(x)$ about $a = 0$ is necessary so that $T_n(x)$ approximates $f(x)$ to 10 significant digits on a) $x \in [-\pi/4, \pi/4]$? and b) $x \in [-\pi/2, \pi/2]$?

As discussed previously, an appropriate value to use for M_{n+1} is 1. Thus,

$$|e_n^T(x)| \leq \frac{1}{(n+1)!}|x-0|^{n+1} = \frac{1}{(n+1)!}|x|^{n+1} \tag{166}$$

From the form of the error term, it is clear that Taylor-polynomial approximations tend to get worse as the polynomial is evaluated further and further from the point of expansion $x = a$. In fact, the aproximation, by design, is exact at $x = a$, and degrades as x moves away from a. Thus, when bounding the error on an entire intervale, we must consider the worse-case scenario when x is as far as possible from a. For the example problem, part a), $a = 0$, and $I = [-\pi/4, \pi/4]$. Thus the furthest reaches of the interval are at either $x = -\pi/4$ or $x = \pi/4$. Either way, $|x| = \pi/4$. Combining this fact with Eq. 166, we get

$$|e_n^T(x)| \leq \frac{1}{(n+1)!}\left(\frac{\pi}{4}\right)^{n+1} \tag{167}$$

Thus, if n is chosen to satisfy

$$\frac{1}{(n+1)!}\left|\frac{\pi}{4}\right|^{n+1} \leq 10^{-10} \tag{168}$$

then by the transitive property of inequality, the absolute error of Eq. 167 will be less than or equal to 10^{-10}. Eq. 168 is an inequality for n, whose solution can be found by trial and error, as shown in Table 14,

which is derived from a Fortran program in which the notation "E-09," for example, means $\times 10^{-9}$. The reader can verify that the lowest value of n for which Eq. 168 holds is $n = 11$.

n	$\frac{1}{(n+1)!}\left\|\frac{\pi}{4}\right\|^{n+1}$
0	0.785398
1	0.308425
2	0.807455E-01
3	0.158543E-01
4	0.249039E-02
5	0.325992E-03
6	0.365762E-04
7	0.359086E-05
8	0.313362E-06
9	0.246114E-07
10	0.175725E-08
11	0.115012E-09
12	0.694845E-11
13	0.389807E-12
14	0.204102E-13
15	0.100189E-14

Table 14: Error bound of the Taylor polynomial for $\sin(x)$ on $[-\pi/4, \pi/4]$.

The (b) part of the example problem is left to the reader. How should Eqs. 167 and 168 be modified for part (b)? Construct a table similar to Table 14 to determine n.

Taylor's Remainder Theorem brings us to a vitally important topic: truncation error. Chapter 3 addressed sources of error, but focused upon round-off error. In numerical computations, the two error sources of greatest concern are round-off error and truncation error. Round-off error was defined as that error incurred when real numbers are stored as floating-point numbers with finite-bit mantissas. It is now time to define *truncation error*.

DEF: *Truncation error* is that error incurred by the computer's inability to perform infinitessimal limit processes or infinite sums.

For example, we have said that the Taylor-series expansion of $\sin(x)$ is mathematically exact. However, it is not useful for a computer because series have an infinite number of terms. By truncating the (infinite) tail of the series, we get a finite Taylor polynomial $T_n(x)$ that can be evaluated by computer, but at the price of an approximation rather than an exact result. Thus, the difference between a Taylor series for $\sin(x)$ and a Taylor polynomial for the same is truncation error. Taylor's Remainder Theorem (Eq. 163) is extremely useful because it gives an explicit formula for the truncation error of the Taylor polynomial. In the next chapter, we will meet truncation error of a different type, that due to the inability of the computer to take a limit. But first, back to interpolation.

7.4 Interpolation Error

Thus far, in this chapter we have talked about two different types of approximations of functions: polynomial interpolation and Taylor polynomials. For the latter, we have discussed an error formula that allows us to determine in advance how many terms are necessary for a pre-specified accuracy. It remains for us to derive an error bound for the interpolating polynomial.

For this purpose we will need Rolle's Theorem, so please refer to Fig. 11 and the discussion in Chapter 5. For all subsequent analysis, we assume that $f(x)$ is differentiable as many times as necessary.

Let's first consider the error of linear interpolation by Newton's method, for which $P_1(x) = f[x_0] + f[x_0,x_1](x-x_0)$. Thus,

$$e_1(x) = f(x) - P_1(x) \tag{169}$$

Note that $e_1(x_0) = e_1(x_1) = 0$, because x_0 and x_1 are nodes. Thus Rolle's theorem applies and there is an ξ between x_0 and x_1 such that

$$\begin{aligned} & e_1'(\xi) = 0 \\ \Rightarrow \quad & f'(\xi) - P_1'(\xi) = 0 \\ \Rightarrow \quad & f'(\xi) = P_1'(\xi) \\ \Rightarrow \quad & f'(\xi) = f[x_0,x_1] \end{aligned} \tag{170}$$

The previous equation relates Newton's first divided difference to the first derivative of the original function; that is, $f[x_0,x_1] = f'(\xi)$, for ξ the interval containing x_0 and x_1.

Let's now consider quadratic interpolation by Newton's method, which will ultimately relate the second divided difference to the second derivative. In this case,

$$P_2(x) = f[x_0] + f[x_0,x_1](x-x_0) + f[x_0,x_1,x_2](x-x_0)(x-x_1) \tag{171}$$

and

$$e_2(x) = f(x) - P_2(x) \tag{172}$$

Note that $e_2(x)$ has at least three zeroes, at the nodes x_0, x_1, and x_2. By Rolle's theorem, there must be an ξ_1 between x_0 and x_1 such that $e_2'(\xi_1) = 0$. But also, there must be an ξ_2 between x_1 and x_2 such that $e_2'(\xi_2) = 0$. Thus, $e_2'(x)$ has at least two zeroes; hence, we can apply Rolle's theorem once again to conclude that there exists ξ between ξ_1 and ξ_2 such that $e_2''(\xi) = 0$. From this fact if follows that

$$\begin{aligned} & e_2''(\xi) = 0 \\ \Rightarrow \quad & f''(\xi) - P_2''(\xi) = 0 \\ \Rightarrow \quad & f''(\xi) = P_2''(\xi) \\ \Rightarrow \quad & f''(\xi) = 2f[x_0,x_1,x_2] \end{aligned} \tag{173}$$

The end result relates the second divided difference and the second derivative as follows:

$$f[x_0,x_1,x_2] = \frac{f''(\xi)}{2} \tag{174}$$

for ξ in the interval containing x_0, x_1, and x_2.

We could continue this process, but hopefully it is clear that, because $e_3(x)$ will have at least four zeroes at the four nodes, then $e_3'''(x)$ will have at least one zero, and thus is subject to Rolle's, from which it is derived that

$$f[x_0,x_1,x_2,x_3] = \frac{f'''(\xi)}{6} \tag{175}$$

The pattern is established and it is now safe to generalize. We present the generalization as a theorem, which could be proven by induction.

THEOREM: Let $f(x)$ be a real-valued function on $[a,b]$ and n times differentiable on (a,b). If x_0, x_1, x_2, ..., x_n are distinct nodes in $[a,b]$, then there exists $\xi \in [a,b]$ such that

$$f[x_0,x_1,...,x_n] = \frac{f^{(n)}(\xi)}{n!} \tag{176}$$

The theorem is important because it makes a connection between Newton's nth divided difference and the nth derivative of the original function $f(x)$. We'll now use this connection to derive an error formula for polynomial interpolation.

Consider the interpolation error

$$e_n(x) = f(x) - P_n(x) \tag{177}$$

where $P_n(x)$ is the Newton form of the polynomial that interpolates $f(x)$ at the $n+1$ nodes $\{x_0,x_1,x_2,...,x_n\}$ with $x_i \in [a,b]$. Suppose we add another node $\bar{x} \in [a,b]$ distinct from all the previous nodes. Then

$$P_{n+1}(x) = P_n(x) + f[x_0,x_1,...,x_n,\bar{x}]\Pi_{k=0}^{n}(x-x_k) \tag{178}$$

Now, because $\bar{x}$ is a node, $P_{n+1}(\bar{x}) = f(\bar{x})$, in which case

$$\begin{aligned} f(\bar{x}) &= P_n(\bar{x}) + f[x_0,x_1,...,x_n,\bar{x}]\Pi_{k=0}^{n}(\bar{x}-x_k) \\ \Rightarrow f(\bar{x}) - P_n(\bar{x}) &= f[x_0,x_1,...,x_n,\bar{x}]\Pi_{k=0}^{n}(\bar{x}-x_k) \qquad (179) \\ \Rightarrow e_n(\bar{x}) &= f[x_0,x_1,...,x_n,\bar{x}]\Pi_{k=0}^{n}(\bar{x}-x_k) \end{aligned}$$

(180)

Our final result comes by replacing the $n+1$ divided difference in Eq. 179 with the $n+1$ derivative from Eq. 176. We obtain the following:

$$e_n^P(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}\Pi_{k=0}^{n}(x-x_k) \;\; \xi \in [a,b] \tag{181}$$

where the superscript P denotes the error of "polynomial" interpolation. You may be wondering where the $\bar{x}$ went. It turns out that there was nothing special about the new node $\bar{x}$ except that it was distinct from the previous nodes; otherwise, $\bar{x}$ could have been any point in the interval $[a,b]$. Because there was nothing special about $\bar{x}$, we have dropped the distinguishing mark (overline), and now just refer to the point as x.

To make sure you understand Eq. 181, let's specialize it to constant, linear, and quadratic interpolation, in which cases

$$\begin{aligned} e_0^P(x) &= f'(\xi)(x-x_0) \\ e_1^P(x) &= \frac{f''(\xi)}{2}(x-x_0)(x-x_1) \\ e_2^P(x) &= \frac{f'''(\xi)}{6}(x-x_0)(x-x_1)(x-x_2) \end{aligned} \tag{182}$$

where, in each case, $\xi \in [a,b]$.

It is instructive to ask the question: Under what circumstances is constant, linear, or quadratic interpolation exact? From the example error formulas of Eqs. 182, it is clear that the error always vanishes at the nodes. This is by design. Note too that constant interpolation would be exact if $f'(x) = 0$ on $[a,b]$, which is the case if the function f were indeed a constant. Similarly, linear interpolation would be exact if $f''(x) = 0$ on $[a,b]$, in which case f is at most linear. Finally, quadratic interpolation is exact if the underlying function is quadratic, in which case $f'''(x) = 0$. So, the error formulas don't just appear out of nowhere. They are self-consistent and they make sense.

As with the Taylor remainder term, the mysterious ξ is a bit of a problem when it comes to evaluating the error of polynomial interpolation. The following inequality is sometimes easier to use:

$$|e_n^P(x)| \leq \frac{M_{n+1}}{(n+1)!}|\Pi_{k=0}^n(x-x_k)| \tag{183}$$

where, once again, M_{n+1} is an upper bound on $f^{(n+1)}(x)$, $x \in [a,b]$. The function $\Pi_{k=0}^n(x-x_k)$ is sometimes referred to as $\psi_n(x)$; note that it depends only upon the placement of the nodes and not upon the function f. It is illustrative and a bit sobering to graph $\psi_n(x)$ for a relatively large value of n under the supposition that the nodes are equally spaced on $[a,b]$. So, for pedagogical purposes, let's consider $n = 6$ and $[a,b] = [-3,3]$, in which case $x_0 = -3$, $x_1 = -2$, $x_2 = -1$, $x_3 = 0$, $x_4 = +1$, $x_5 = +2$, and $x_6 = +3$. Figure 25 shows $\psi_6(x)$ for this distribution of nodes. Note that $\psi_6(x)$ oscillates wildly, especially near the ends of the interval. It may not be obvious, but what Fig. 25 implies is that high-order polynomial interpolation is not necessarily a good idea if the nodes are equally spaced. In fact, going to higher order can actually make the interpolant worse instead of better. The problem is not high-order interpolation per se, but rather the uniform distribution of nodes. When the nodes are uniformly distributed, $\psi_n(x)$ oscillates wildly as shown in the figure. Interpolation error can be tamed only if one is willing and able to redistribute the nodes, with a greater density of nodes toward the ends of the interval than near the middle. In Math 448, we will examine the Chebyshev distribution of nodes, which is in a sense optimal. With this distribution, phenomenal accuracy is possible with high-order interpolation. In contrast, with a uniform distribution of nodes, one should probably not consider interpolation of order higher than, say, five.

Figure 25 also reveals why extrapolation is usually a terrible idea. Recall that extrapolation is the evaluation of the interpolating polynomial *outside* its design interval $[a,b]$. Notice how quickly $\psi_n(x)$ grows outside the interval, which implies that the error $e_n^P(x)$ also grows at a phenomenal rate outside the interval. At best, extrapolation is dangerous and to be avoided.

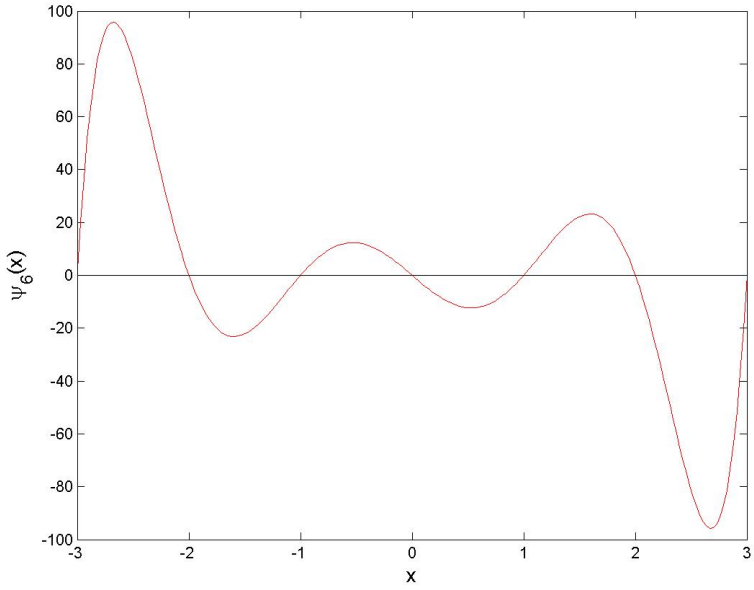


Figure 25: Function ψ of interpolation error formula, for equally spaced nodes.

7.4.1 Interpolation with Equally Spaced Nodes

From the previous discussion, it can be concluded that interpolation should be restricted to relatively low order if the nodes are equally spaced. With this in mind, let's consider the error of linear, quadratic, and cubic interpolation when the nodes are each separated by a uniform interval width of h. That is, the nodal placements are given by

$$x_i = x_0 + ih \quad i = 0, 1, 2, ..., n \tag{184}$$

where $x_0 = a$ and $h = (b-a)/n$. From Eq. 183 for $n = 1$, $n = 2$, and $n = 3$, respectively, we have

$$|e_1^P(x)| \leq \frac{M_2}{2}|\psi_1(x)| \tag{185}$$

$$|e_2^P(x)| \leq \frac{M_3}{6}|\psi_2(x)| \tag{186}$$

$$|e_3^P(x)| \leq \frac{M_4}{12}|\psi_3(x)| \tag{187}$$

where

$$\psi_1(x) = (x-x_0)(x-x_1) \tag{188}$$

$$\psi_2(x) = (x-x_0)(x-x_1)(x-x_2) \tag{189}$$

$$\psi_3(x) = (x-x_0)(x-x_1)(x-x_2)(x-x_3) \tag{190}$$

and where the reader is reminded that M_2, M_3, and M_4 are upper bounds on f'', f''', and $f^{(4)}$, respectively, on the interval I containing all relevant nodes. The error will tend to be worst near where the $\psi_n(x)$ have their extrema.

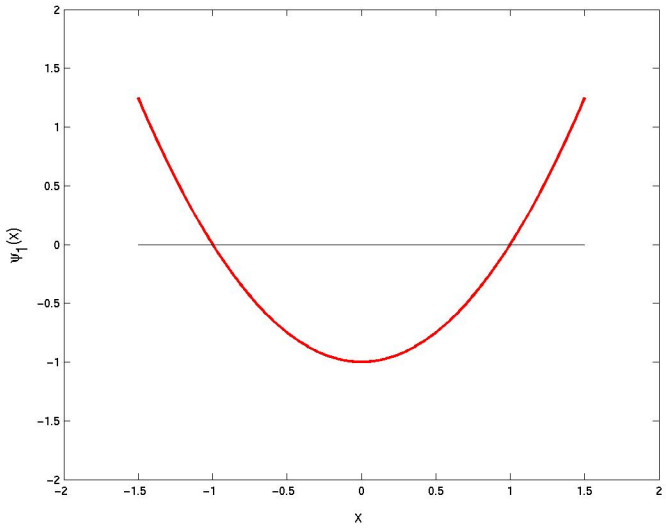


Figure 26: Interpolation error function $\psi_1(x)$, shown for $x_0 = -1$ and $x_1 = +1$.

Let's look first at $\psi_1(x)$, which is quadratic in x, a parabola opening upwards with zeroes at nodes x_0 and x_1 (Fig. 26). By inspection, the absolute minimum of $\psi_1(x)$ occurs midway between x_0 and x_1, that is at $x_0 + \frac{h}{2}$. The minimum is therefore $\psi_1(x_0 + \frac{h}{2}) = (\frac{h}{2})(-\frac{h}{2}) = -\frac{h^2}{4}$. Consequently $|\psi_1(x)| \leq \frac{h^2}{4}$ on $I = [x_0, x_1]$. Combining this fact with Eq. 185 yields

$$|e_1^P(x)| \leq \frac{M_2 h^2}{8} \tag{191}$$

Similarly, let's find the extrema of $\psi_2(x)$, a cubic with three zeroes: at x_0, x_1, and x_2. For convenience, write

$$\begin{aligned} \psi_2(x) &= [x-(x_1-h)](x-x_1)[x-(x_1+h)] \\ &= [(x-x_1)+h](x-x_1)[(x-x_1)-h)] \end{aligned} \tag{192}$$

The problem becomes a bit easier if we now simplify by the change of variables $u = x - x_1$, in which case $\psi_2(u) = (u+h)(u)(u-h) = u(u^2-h^2)$. Relative extrema of $\psi_2(u)$ occur where its first derivative vanishes. Thus, it suffices to solve $\psi_3'(u) = 3u^2 - h^2 = 0$. There are two critical points: $u \in \{-\frac{h}{\sqrt{3}}, +\frac{h}{\sqrt{3}}\}$. Returning to the original variable, critical values occur at $x = x_1 \pm \frac{h}{\sqrt{3}}$. The first gives the absolute maximum of $\frac{2}{3\sqrt{3}}h^3$, and the second, the absolute minimum of $-\frac{2}{3\sqrt{3}}h^3$. Either way, $|\psi_2(x)| \leq \frac{2}{3\sqrt{3}}h^2$. Combining this result with Eq. 186 yields

$$|e_2^P(x)| \leq \frac{M_3 h^3}{9\sqrt{3}} \tag{193}$$

It is interesting to note that the extrema of $\psi_2(x)$ occur slightly closer to the ends of the interval than to the middle of the interval.

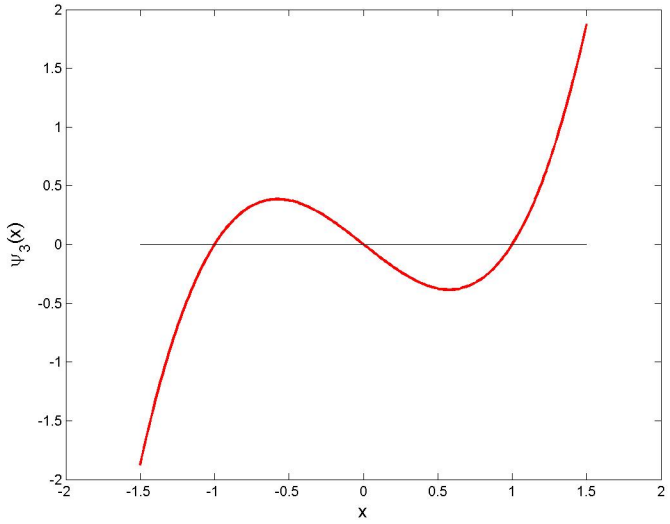


Figure 27: Interpolation error function $\psi_2(x)$, shown for $x_0 = -1$, $x_1 = 0$, and $x_1 = +1$.

Clearly we can continue in this vein to bound $|e_3^P(x)|$. Mercifully, we will skip the details here and simply summarize the results for low-order interpolation with nodes equally spaced with interval width h.

$$\begin{aligned} |e_1^P(x)| &\leq \frac{M_2 h^2}{8} \\ |e_2^P(x)| &\leq \frac{M_3 h^3}{9\sqrt{3}} \\ |e_3^P(x)| &\leq \frac{M_4 h^4}{24} \end{aligned} \tag{194}$$

So, what is the point of this exercise? Interpolation accuracy can be improved in either or both of two ways: 1) going to higher-order interpolation, or 2) placing nodes closer together. Previous discussion has shown the limits of the former: high-order is not always better if the nodes are equally spaced. In that event, Eqs. 194 are useful in determining an appropriate spacing of nodes to maintain a desired interpolation accuracy when the order of the interpolating polynomial is relatively low. The following example illustrates how the error bounds work.

EX: Find spacing h such that $P_3(x)$ interpolates $\cos(x)$ correctly to four places (absolute error). Thus, we want

$$|e_3^P(x)| \leq \underline{\frac{M_4 h^4}{24} \leq 0.0001} \tag{195}$$

The underlined portion of the inequality implicitly specifies h. For this problem $M_4 = 1$. Thus

$$\begin{aligned} \frac{h^4}{24} &\leq 0.0001 \\ \Rightarrow h^4 &\leq 0.0024 \\ \Rightarrow h &\leq 0.2213 \end{aligned}$$

Thus any value of h less than 0.2213, say $h = 0.2$, would guarantee at least the desired accuracy.

To close this chapter, we would like to summarize the two major results: Taylor's Remainder Theorem, and the formula for interpolation error. These are repeated below.

$$e_n^T(x) \equiv f(x) - T_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x-a)^{n+1} \tag{196}$$

$$e_n^P(x) \equiv f(x) - P_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x-x_0)(x-x_1)(x-x_2)...(x-x_n) \tag{197}$$

In the former ξ is between a and x. In the later ξ is somewhere in the interval $[a,b]$ that contains all the nodes. We hope that the similarity of these two error formulas has not escaped the notice of the reader. It turns out, in fact, that the Taylor remainder term is exactly what the interpolation error would be if all $n+1$ distinct nodes x_i coalesced into the single node $x = a$! This is in fact a good way to remember the two formulas. Remember only one, and then think about how they are related. Both error formulas are used again in the following chapters, so it is a good idea to commit them to memory.

7.5 Exercises

1. Let $(x_1,y_1),(x_2,y_2),(x_3,y_3)$ be three data points such that $x_1 < x_2 < x_3$ and $y_1,y_2,y_3 > 0$. Does this imply that the quadratic function interpolating these points is positive on $[x_1,x_3]$? Explain.

2. Find the Taylor Polynomial,$p_n(x)$, of $f(x) = e^{x^2} - \frac{1}{1-x}$ near x=0. Hint: Find the Taylor Polynomial of the two terms to the right of the = sign separately, add the results together, and simplify.

3. How large should n be chosen such that $|e^{x^2} - p_n(x)| \leq 10^{-5}, -1 \leq x \leq 1$?

4. Find a way to calculate accurate values as $x \to 0$ of $f(x) = \frac{x(e^x - \cos(x)}{\sqrt{1+x^2}-1}$.

5. (a) Given the four points (1,2), (3,4), (5,3), and (9,8) write the Lagrangian interpolating polynomial $L(x)$ that passes through each point.
   (b) Give the third degree Newton polynomial $P(x)$ at the above four points.
   (c) Without expanding, can you determine a relationship between $L(x)$ and $P(x)$? If so, what is it?

6. Suppose we have the following two functions

$$f(x) = e^{-2x}$$
$$g(x) = \sin x$$

   Suppose we interpolate each function with a cubic polynomial using equally spaced points in the interval $[0,2]$. Which function, $f(x)$ or $g(x)$, will have the better polynomial interpolant? Explain.

7. (a) Find the third degree Taylor polynomial for $f(x) = ln(x)$ expanded about $c = 2$.
   (b) Approximate $\int_2^3 ln(x)dx$ by integrating the polynomial found in part (a).

(c) Derive an error bound for your result in part (b) by integrating and manipulating the remainder term that would be associated with your polynomial in part (a).

8. (a) Use the Vandermonde matrix method to find the quadratic polynomial $P_2(x)$ that passes through the points (0,1), (1,2), and (2,5).

   (b) Find the determinant of the matrix in part (a).

   (c) Does the matrix have an inverse? Why or why not?

   (d) Also find $P_2(x)$ by Newton's divided difference method. Do you get the same polynomial?

9. (a) For $a = 0$, what order n of Taylor polynomial $T_n(x)$ is required to approximate $f(x) = sin(x)$ to machine single precision on $[0, \frac{\pi}{3}]$?

   (b) Find $T_n(x)$.

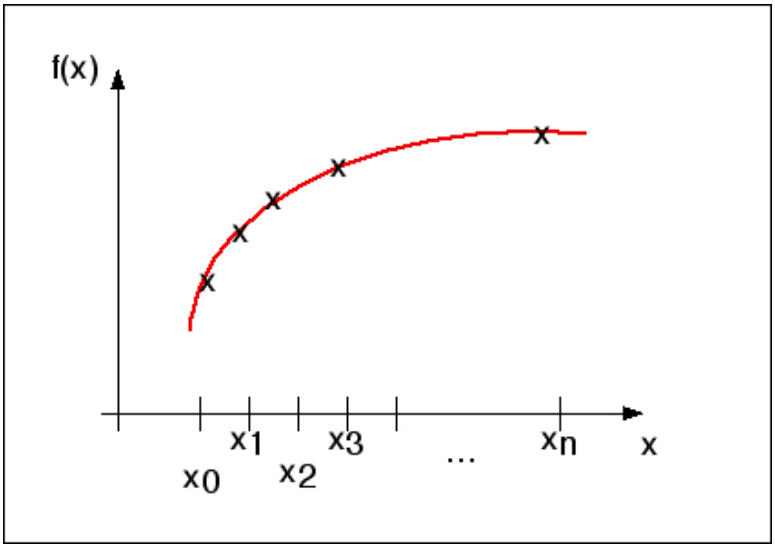


Figure 28: Numerical differentiation from discrete data.

8 Part II: Numerical Differentiation

In this chapter, we wish to consider the approximation of the derivative of a function, when the function is known only as a discrete set of points. More specifically, consider the set of $n+1$ points given by $\{[x_i, f(x_i)],\ i = 0, 1, ..., n\}$. For now assume that the points are equally spaced in x such that $x_i = x_0 + ih$, and h is the length of the interval between adjacent points. Further suppose that these points represent a discrete "sample" of a continuous underlying function $f(x)$, as shown in Fig. 28. Herein lies the rub. We don't know $f(x)$; we know only its value at a few points. More to the point, what we are really after is $f'(x)$, the derivative of $f(x)$. Just differentiate, you say. The problem is, we can't. Differentiability implies continuity; however the function f is known only partially, as a discontinuous set of points that comprise some representative sample of $f(x)$. Derivative rules don't apply to discontinuous sets. What is to be done?

Fortunately we laid all the groundwork in the previous chapter. The game plan is relatively straightforward. It consists of two steps: 1) Fit polynomial $P_k(x)$ through some subset of the $n+1$ points, which implies $k \leq n$. 2) Approximate the derivative of the function at the nodes by the derivative of the interpolating polynomial; that is, $f'(x_i) \approx P_k'(x_i)$.

The devil of course is in the details. The first practical question is: How big a subset should we use? That is, what value should k have? Another question is: At what node x_i do we evaluate the derivative of $P_k(x)$? One way around having to answer the first question is to simply start small and work up to larger values of k. The simplest type of function that has a non-zero derivative is the linear function.Thus, let's start with $k = 1$.

8.1 Two-Point Formulas

Here, we consider only two points: $[x_0, f(x_0)]$ and $[x_1, f(x_1)]$. The Newton form of the linear interpolating polynomial is

$$\begin{aligned} P_1(x) &= f(x_0) + f[x_0, x_1](x - x_0) \\ &= f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0) \quad x_0 \le x \le x_1 \end{aligned} \tag{198}$$

The derivative of $P_1(x)$ is a constant, namely

$$P_1'(x) = \frac{f(x_1) - f(x_0)}{x_1 - x_0} \quad x_0 \le x \le x_1 \tag{199}$$

The approximation above is valid over the interval $[x_0, x_1]$, provided we take a one-sided derivative if at an endpoint of the interval spanned by the two nodes. Thus, there are two ways to interpret the fact that $P_1\prime(x) = \frac{f(x_1) - f(x_0)}{x_1 - x_0} \approx f'(x)$. If we evaluate the derivative at $x = x_0$, then $x_1 = x_0 + h$, $x_1 - x_0 = h$, and

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h} \tag{200}$$

On the other hand, if $x = x_1$, then $x_0 = x_1 - h$, and

$$f'(x_1) \approx \frac{f(x_1) - f(x_1 - h)}{h} \tag{201}$$

Once these formulas are derived, there is no need to retain the subscripts. Generalizing, we obtain

$$f'(x) \approx \frac{f(x + h) - f(x)}{h} \equiv D_{h+} f \tag{202}$$

$$f'(x) \approx \frac{f(x) - f(x - h)}{h} \equiv D_{h-} f \tag{203}$$

The former is known as the *two-point forward-difference approximation* to the first derivative, given the symbol $D_{h+} f$, and the latter is known as the *two-point backward-difference approximation*, whose symbol is $D_{h-} f$. A word on the symbolism. Here D denotes an approximate differentiation *operator*, which operates on the function f, known only discretely, with adjacent nodes separated by by distance h. The plus or minus designators refer to "forward" or "backward," respectively, and the h refers to the interval width between the nodes, also known as the *step size*.

Note that the forward-difference approximation $D_{h+} f$ gives the *exact* derivative in the infinitessimal limit as $h \to 0$, because this is, in fact, the definition of the derivative of $f(x)$! On the other hand, if h remains finite, $D_{h+} f$ only approximates the derivative of $f(x)$. Similarly, it can be easily shown that $D_{h-} f$ is also exact in the infinitessimal limit.

So, how well do two-point approximations work in reality? Let's try an example.

EX: Suppose $f(x) = \sin(x)$. Approximate $f'(\frac{\pi}{3})$ by both $D_{h+}f$ and $D_{h-}f$ with $h = 0.01$. In this pedagogical example, we of course know the exact answer: $f'(\frac{\pi}{3}) = \cos(\frac{\pi}{3}) = \frac{1}{2}$.

$$\begin{aligned} D_{h+}f &= \frac{\sin(\frac{\pi}{3}+0.01)-\sin(\frac{\pi}{3})}{0.01} = 0.495661573 \\ D_{h-}f &= \frac{\sin(\frac{\pi}{3})-\sin(\frac{\pi}{3}-0.01)}{0.01} = 0.504321758 \end{aligned}$$

Neither result is particularly impressive. The forward-difference approximation undershoots the exact result by a bit less than one percent, and the backward-difference approximation overshoots by about the same amount. You might be wondering what happens if we average these two values.

$$\frac{D_{h+}f + D_{h-}f}{2} = 0.499991667 \tag{204}$$

Now that is an impressive result, differing from the true value only in the 6th significant digit. Is it a fluke that the average was so close to the true value, or have we stumbled onto something important? Stay tuned.

8.2 Three-Point Formulas

Two-point formulas suffer from a serious flaw. The second derivative of a linear function is zero. Thus, two-point formulas are inappropriate if both first- and second-derivative approximations are needed. For example, if the original function is $s(t)$ (position s as a function of time t), then its first and second derivatives with respect to time are velocity and acceleration, respectively. A linear approximation of position would always yield zero for the approximation of acceleration. Not too swift. To insure that the interpolating polynomial allows for non-zero first and second derivatives, we need to consider at least quadratic polynomials; that is, $2 \leq k$. So, once again adopting a minimalist approach, let's consider $k = 2$, in which case we need three points: $[x_0, f(x_0)]$, $[x_1, f(x_1)]$, and $[x_2, f(x_2)]$.

The Newton form of the quadratic interpolating polynomial is

$$P_2(x) = f(x_0) + f[x_0,x_1](x-x_0) + f[x_0,x_1,x_2](x-x_0)(x-x_1) \tag{205}$$

in which case

$$\begin{aligned} P_2'(x) &= f[x_0,x_1] + f[x_0,x_1,x_2][2x-(x_0+x_1)] && (206) \\ P_2''(x) &= 2f[x_0,x_1,x_2] && (207) \end{aligned}$$

The expressions above are valid for the interval $[x_0, x_2]$. Thus we have the option of evaluating $P_2'(x)$ at x_0, x_1, or x_2. Let's first pick the center node and evaluate $P_2'(x)$ at $x = x_1$. That is,

$$\begin{aligned} P_2'(x_1) &= f[x_0,x_1] + [2x_1-(x_0+x_1)]f[x_0,x_1,x_2] \\ &= f[x_0,x_1] + (x_1-x_0)f[x_0,x_1,x_2] \\ &= f[x_0,x_1] + h\frac{f[x_1,x_2]-f[x_0,x_1]}{x_2-x_0} \end{aligned}$$

$$
\begin{aligned}
&= f[x_0,x_1] + h\frac{f[x_1,x_2]-f[x_0,x_1]}{2h} \\
&= \frac{1}{2}\{f[x_0,x_1]+f[x_1,x_2]\} \\
&= \frac{1}{2}\left[\frac{f(x_1)-f(x_0)}{h}+\frac{f(x_2)-f(x_1)}{h}\right] \\
&= \frac{f(x_2)-f(x_0)}{2h} \\
&= \frac{f(x_1+h)-f(x_1-h)}{2h}
\end{aligned}
$$

Once the derivative formula has been established, it is no longer necessary to retain the nodal subscript. Thus, in general,

$$f'(x) \approx \frac{f(x+h)-f(x-h)}{2h} \equiv D_h f \tag{208}$$

This formula is known as the *centered- (or central-) difference approximation* of f' at x. Note that the centered-difference approximation is the average of the forward- and backward-difference approximations. That is $D_h f = \frac{1}{2}[D_{h+}f + D_{h-}f]$. It was this approximation that gave such a good result in our previous numerical example.

What would have happened had we evaluated $P_2'(x)$ at x_0 rather than x_1? In this case,

$$
\begin{aligned}
P_2'(x_0) &= f[x_0,x_1] + [2x_0-(x_1+x_0)]f[x_0,x_1,x_2] \\
&= f[x_0,x_1] - (x_1-x_0)f[x_0,x_1,x_2] \\
&= f[x_0,x_1] - h\frac{f[x_1,x_2]-f[x_0,x_1]}{x_2-x_0} \\
&= f[x_0,x_1] - h\frac{f[x_1,x_2]-f[x_0,x_1]}{2h} \\
&= \frac{3}{2}f[x_0,x_1] - \frac{1}{2}f[x_1,x_2] \\
&= \frac{3}{2}\left[\frac{f(x_1)-f(x_0)}{h}\right] - \frac{1}{2}\left[\frac{f(x_2)-f(x_1)}{h}\right] \\
&= \frac{-3f(x_0)+4f(x_1)-f(x_2)}{2h}
\end{aligned}
$$

Once again, dropping the subscript and generalizing, we obtain the *three-point forward difference approximation* of the first derivative, namely

$$f'(x) \approx \frac{-3f(x)+4f(x+h)-f(x+2h)}{2h} \equiv D_{h++}f \tag{209}$$

We leave it to the reader to evaluate $P_x'(x)$ at $x = x_2$ to obtain the *three-point backward difference approximation* of the first derivative, namely

$$f'(x) \approx \frac{3f(x)-4f(x-h)+f(x-2h)}{2h} \equiv D_{h--}f \tag{210}$$

Finally, recall that we went to a quadratic interpolant in order to have a meaningful second-derivative approximation. Recall also that

$$\begin{aligned} P_2''(x) &= 2f[x_0,x_1,x_2] \\ &= 2\frac{f[x_1,x_2]-f[x_0,x_1]}{x_2-x_0} \\ &= 2\frac{f[x_1,x_2]-f[x_0,x_1]}{2h} \\ &= \frac{\frac{f(x_2)-f(x_1)}{h} - \frac{f(x_1)-f(x_0)}{h}}{h} \\ &= \frac{f(x_2)-2f(x_1)+f(x_0)}{h^2} \end{aligned}$$

If we interpret the expression above as having been evaluated at the center node $x = x_1$, then $x_0 = x_1 - h$ and $x_2 = x_1 + h$. Generalizing, this leads to the *centered-difference approximation to the second derivative*, namely

$$f''(x) \approx \frac{f(x+h)-2f(x)+f(x-h)}{h^2} \equiv D_h^{(2)} f \tag{211}$$

EX: Approximate $f''(\frac{\pi}{3})$ for $f(x) = \sin(x)$ using the centered-difference approximation with $h = 0.01$. (Note that the exact value is $-\sin(\frac{\pi}{3}) = -\frac{\sqrt{3}}{2} = -0.866025404$.)

$$D_h^{(2)} f = \frac{\sin(\frac{\pi}{3}+0.01) - 2\sin(\frac{\pi}{3}) + \sin(\frac{\pi}{3}-0.01)}{(0.01)^2} = -0.866018185 \tag{212}$$

Note that the approximation is correct to between 4 and 5 significant digits!

8.3 Truncation Error Formulas

We'd like to begin this section with a table of errors derived in answer to the following problem: Compare the approximations of $D_{h+}f$, $D_{h-}f$, and $D_h f$ for $f(x) = \ln(x)$ at $x = 2$ for the successive values $h = 1/4$, $h = 1/8$, $h = 1/16$, and $h = 1/32$. Describe how the error diminishes as h diminishes in each case. (Note that, in this pedagogical example, $f'(x) = 1/x$; thus $f'(2) = 1/2 = 0.5$.

h	$D_{h+}f$	$e_{h+}f$	$D_{h-}f$	$e_{h-}f$	$D_h f$	$e_h f$
1/4	0.471132	0.028868	0.534126	-0.034126	0.502629	-0.002629
1/8	0.484997	0.015003	0.516308	-0.016308	0.500653	-0.000652
1/16	0.492346	0.007653	0.507979	-0.007979	0.500163	-0.000163
1/32	0.496134	0.003866	0.053947	-0.003947	0.500040	-0.000041
		$O(h)$		$O(h)$		$O(h^2)$

Table 15: Errors of difference approximations of first derivative.

It is most illuminating to glance down the columns that give the errors of the three approximations. In the first two error columns, each time h is diminished by a factor of two, the error also diminshes by very nearly a factor of two. Notice also that the error columns $e_{h+}f$ and $e_{h-}f$ have approximately the same magnitudes but are of opposite signs. This suggests that an average of $D_{h+}f$ and $D_{h-}f$ would give a nearly correct value because the errors of the one-sided approximations would tend to cancel. This is in fact verified by the third error column, which gives the error of the centered-diffference approximation, which is equivalent to the average of the forward- and backward-difference approximations. Finally, notice how the error in the last column behaves. Each time h is diminished by a factor of two, the error diminishes by a factor of approximately four. That is, $D_{h/2}f \approx 1/4 D_h f$. We can summarize these observations by saying that the errors for the biased (one-sided) difference approximations tends to diminish like h as h decreases. In contrast, the error for the centered approximation tends to diminish like h^2. In shorthand, we say that the error behaves as $O(h)$ ("order h") and $O(h^2)$, respectively.

We are now going to modify Taylor's Remainder Theorem from the previous chapter to actually *derive* these error properties.

ASIDE: Alternative Form of Taylor's Theorem

Let's change the variable names in Eqs. 159 and 163, replacing x by z and a by x, in which case Taylor's Remainder Theorem can be written as follows:

$$\begin{aligned} f(z) &= f(x) + f'(x)(z-x) + \frac{f''(x)}{2}(z-x)^2 + \ldots + \frac{f^{(n)}(x)}{n!}(z-x)^n \\ &+ \frac{f^{(n+1)}(\xi)}{(n+1)!}(z-x)^{n+1} \end{aligned}$$

The last term, the error term, is formally identical to all previous terms except that the derivative is evaluated at ξ, which lies between x and z. By now letting $z = x+h$ ($\Rightarrow z-x = h$), we now get the desired form of Taylor's Theorem.

$$f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2}h^2 + \ldots + \frac{f^{(n)}(x)}{n!}h^n + \underline{\frac{f^{(n+1)}(\xi)}{(n+1)!}h^{n+1}} \quad (x < \xi < x+h) \tag{213}$$

where we presume that $h > 0$.

The expression above can be viewed as an expansion for the quantity $f(x+h)$. It remains exact *regardless of the terminating value of n* provided that the last term $(n+1)$ (underlined above) is expressed as the remainder term (with ξ rather than x).

In order to derive the error formulas for numerical differentiation approximations, we will also need expansions for $f(x-h)$, $f(x+2h)$, and $f(x-2h)$. There is no need to reinvent the wheel here. Simply adapt Eq. 213 by replacing h by $-h$ to obtain the expansion for $f(x-h)$, in which case

$$\begin{aligned} f(x-h) &= f(x) + f'(x)(-h) + \frac{f''(x)}{2}(-h)^2 + \ldots + \frac{f^{(n)}(x)}{n!}(-h)^n \\ &+ \frac{f^{(n+1)}(\xi)}{(n+1)!}(-h)^{n+1} \quad (x-h < \xi < x) \end{aligned}$$

Notice that the signs alternate. Even ordered terms are postive; odd order terms are negative. Notice also that, here, ξ lives between $x-h$ and x.

Similarly, we readily derive the following expansion for $f(x+2h)$ by replacing h with $2h$ in Eq. 213.

$$\begin{aligned} f(x+2h) &= f(x)+f'(x)(2h)+\frac{f''(x)}{2}(2h)^2+...+\frac{f^{(n)}(x)}{n!}(2h)^n \\ &+ \frac{f^{(n+1)}(\xi)}{(n+1)!}(2h)^{n+1} \quad (x<\xi<x+2h) \end{aligned}$$

So what is the point of all the fun and games above? We are now ready to use these expansions to derive error formulas for difference approximations of derivatives. Let's start with the forward-difference approximation of the first derivative. First, we remind the reader of the following:

$$\begin{aligned} f'(x) &= \lim_{h\to 0}\frac{f(x+h)-f(x)}{h} \\ D_{h+}f &= \frac{f(x+h)-f(x)}{h} \end{aligned}$$

Both the exact derivative and the forward-difference approximation to the derivative are defined in terms of *difference quotients*. In the former, however, the limit is taken as h becomes infinitessimally small. In the latter, h remains finite. Recall that *truncation error* is error incurred by the computer's inability to compute infinite sums or to take infinitessimal limits. In the last chapter, we encountered truncation error of the first type. Here, we encounter truncation error of the second type. (Attempts to obtain the exact limit on a computer will ultimately fail from a divide by zero error.) For example, $e_{h+}f = f'(x)-D_{h+}f$ represents the second type of truncation error. We now quantify this error.

$$\begin{aligned} e_{h+}f &\equiv f'(x)-D_{h+}f \text{ (def. of absolute error)} \\ &= f'(x)-\frac{f(x+h)-f(x)}{h} \text{ (def. of } D_{h+}f) \\ &= f'(x)-\frac{[f(x)+hf'(x)+\frac{h^2}{2}f''(\xi)]-f(x)}{h} \text{ (Taylor expansion)} \\ &= f'(x)-\frac{hf'(x)+\frac{h^2}{2}f''(\xi)}{h} \\ &= f'(x)-f'(x)-\frac{h}{2}f''(\xi) \\ &= -\frac{h}{2}f''(\xi) \quad (x<\xi<x+h) \end{aligned} \tag{214}$$

Similarly, the error for the backward-difference approximation is

$$e_{h-}f = +\frac{h}{2}f''(\xi) \quad (x-h<\xi<x) \tag{215}$$

Notice that the errors of the forward- and backward-difference approximations have opposite signs. However, the are not quite equal and opposite, because the values of ξ in the two expressions are different. Nevertheless, if h is small, the two errors are almost additive inverses of one another.

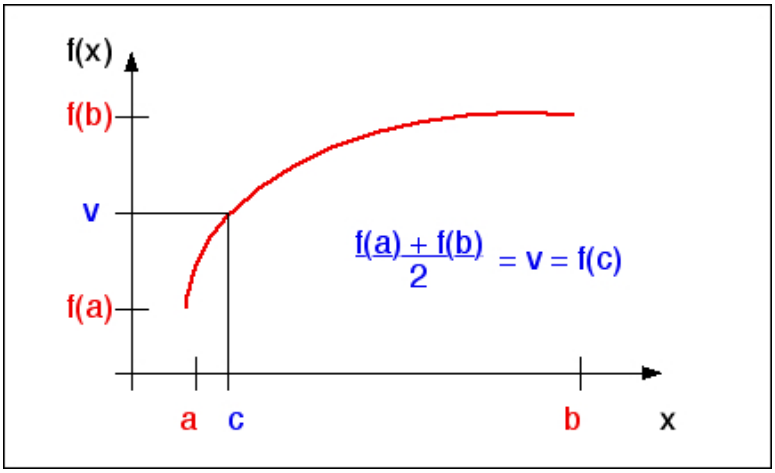


Figure 29: Corollary of the Intermediate Value Theorem.

In order to derive the error formula for the central-difference approximation, we will need another mathematical trick, an adaptation of the Intermediate Value Theorem (IVT), which we encountered previously in Chapter 7.

COROLLARY: If $f(x)$ is continuous on $[a,b]$, then there exists $c \in [a,b]$ such that $f(c) = \frac{f(a)+f(b)}{2}$.

PROOF: Let $v = \frac{f(a)+f(b)}{2}$ and note that v is a value between $f(a)$ and $f(b)$. By the continuity of $f(x)$ and the IVT, $f(x)$ assumes *all* values between $f(a)$ and $f(b)$, and so, in particular, it must assume the value v. (See Fig. 29).

We are now ready to tackle the error of $D_h f$.

$$\begin{aligned}
D_h f &\equiv \frac{f(x+h)-f(x-h)}{2h} \quad \text{(def. of } D_h f\text{)} \\
&= \frac{[f(x)+hf'(x)+\frac{h^2}{2}f''(x)+\frac{h^3}{6}f'''(\xi_1)]-[f(x)-hf'(x)+\frac{h^2}{2}f''(x)-\frac{h^3}{6}f'''(\xi_2)]}{2h} \\
&= \frac{2hf'(x)+\frac{h^3}{6}[f'''(\xi_1)+f'''(\xi_2)]}{2h} \\
&= f'(x)+\frac{h^2}{12}[f'''(\xi_1)+f'''(\xi_2)] \\
&= f'(x)+\frac{h^2}{12}\cdot 2\left\{\frac{1}{2}[f'''(\xi_1)+f'''(\xi_2)]\right\} \\
&= f'(x)+\frac{h^2}{6}f'''(\xi) \quad (x-h<\xi_2<\xi<\xi_1<x+h) \qquad (216)
\end{aligned}$$

where, in the last step, we have used the corollary above to replace the two values of ξ_1 and ξ_2 with a single value of ξ that lies somewhere between them. Finally, using Eq. 216 in the expression for the error $e_h f$, we obtain

$$e_h f \equiv f'(x) - D_h f$$

$$
\begin{aligned}
&= f'(x) - [f'(x) + \frac{h^2}{6} f'''(\xi)] \\
&= -\frac{h^2}{6} f'''(\xi) \quad (x-h < \xi < x+h)
\end{aligned} \tag{217}
$$

We conclude this subsection with a table that summarizes the error formulas for two-point and three-point approximations of the first derivative, including those for $D_{h++}f$ and $D_{h--}f$, which we did not derive. Note that, whereas the two-point difference approximations are each $O(h)$, the three-point difference approximations are each $O(h^2)$. This explains why halving h diminishes the error by a factor of *four* for the higher-order formulas. Although the centered and biased three-point formulas are each $O(h^2)$, the centered formula has the smallest coefficient of error and is to be favored whenever possible.

It is also instructive to ask the question: For what functions is the derivative approximation exact? Certainly, the two-point formulas will be exact if $f(x)$ is linear, in which case $f''(x) = 0$. Thus, two-point formulas, which are based on linear interpolants, are exact for linear functions. Similarly, the three-point approximations, which are based on a quadratic interpolant, are exact for quadratic functions because $f'''(x) = 0$.

HW: Use the methods of this section to derive the error rule for the three-point centered-difference approximation for the second derivative, namely

$$
e_h^{(2)} f \equiv f''(x) - D_h^{(2)} f = -\frac{f^{(4)}(\xi)}{12} h^2 \quad (x-h < \xi < x+h) \tag{218}
$$

Approximation	Error Rule	Interval for ξ
$D_{h+}f$	$-\frac{h}{2}f''(\xi)$	$x < \xi < x+h$
$D_{h-}f$	$+\frac{h}{2}f''(\xi)$	$x-h < \xi < x$
$D_h f$	$-\frac{h^2}{6}f'''(\xi)$	$x-h < \xi < x+h$
$D_{h++}f$	$+\frac{h^2}{3}f'''(\xi)$	$x < \xi < x+2h$
$D_{h--}f$	$-\frac{h^2}{3}f'''(\xi)$	$x-2h < \xi < x$

Table 16: Errors of selected difference approximations of the first derivative.

8.4 Method of Undetermined Coefficients

Undetermined coefficients is another method for deriving the aforementioned two-point and three-point formulas. If we want to approximate a derivative at x_{j-1}, x_j and x_{j+1}, we consider the mathematical expression $Af(x_{j-1}) + Bf(x_j) + Cf(x_{j+1})$. It is more helpful to write the previous expression as $Af(x_j - h) + Bf(x_j) + Cf(x_j + h)$ where we have assumed equally spaced points. Taking a Taylor series of this mathematical expression at $x_j = x$ implies

$$
A\left(f(x) + f'(x)(-h) + \frac{f''(x)}{2}(-h)^2 + \frac{f'''(x)}{6}(-h)^3 + \ldots\right) +
$$

$$Bf(x)+$$
$$C\left(f(x)+f'(x)h+\frac{f''(x)}{2}h^2+\frac{f'''(x)}{6}(-h)^3+...\right)$$

Reordering the terms implies

$$\begin{aligned} f(x)\,(A+B+C)+\\ f'(x)\,(h(C-A))+\\ f''(x)\left(\frac{h^2}{2}(C+A)\right)+\\ f'''(x)\left(\frac{h^3}{6}(C-A)\right)+... \end{aligned}$$

Now we need more information in order to derive the two-point forward difference formula from the earlier section. In order to approximate the first derivative, we want

$$\begin{aligned} f'(x) = f(x)\,(A+B+C)+\\ f'(x)\,(h(C-A))+\\ f''(x)\left(\frac{h^2}{2}(C+A)\right)+\\ f'''(x)\left(\frac{h^3}{6}(C-A)\right)+...+\tau \end{aligned} \tag{219}$$

where τ is the truncation error. Since we are only finding the two-point forward difference approximation, we do not need to consider x_{j-1}, so A=0. Therefore, we need to solve the following system of equations

$$\begin{aligned} B+C&=0\\ hC&=1 \end{aligned}$$

where the first equation amounts to the statement that we *do not* want to approximate the function and the second equation amounts to the statement that we *do* want to approximate the first derivative. The second equation above implies that $C=\frac{1}{h}$ and the first equation implies $B=-\frac{1}{h}$. Thus,

$$f'(x)\approx\frac{f(x+h)-f(x)}{h}. \tag{220}$$

One of the benefits of the method of undetermined coefficients is that finding the truncation error requires very little additional work. Insert the values obtained for B and C into (219) and notice

$$\begin{aligned} f'(x) = f(x)\,(0)\\ f'(x)\,(1)+\\ f''(\xi)\left(\frac{h^2}{2}(\frac{1}{h})\right)+\tau \end{aligned}$$

where the last term is the error term and is thus evaluated for some value $x<\xi<x+h$. Therefore, we can solve for τ and find that $\tau=-\frac{h}{2}f''(\xi)$. Notice this error matches the earlier derivation.

It is analogous to derive the two-point backward difference approximation to the derivative. However, let's calculate the three point centered difference approximation to the first derivative. We have to solve the following system of equations

$$\begin{aligned} A+B+C &= 0 \\ h(C-A) &= 1 \\ \frac{h^2}{2}(C+A) &= 0 \end{aligned}$$

We include the third equation because we have three unknowns. The result of including this third equation is a better truncation error for the centered difference approximation compared with the two-point approximations. Solving this system of equations yields the approximation found in (208).

Another advantage of the method of undetermined coefficients is the ability to approximate the second derivative with very little extra work. For instance, calculating a centered difference approximation to the second derivative implies solving the following system of equations

$$\begin{aligned} A+B+C &= 0 \\ h(C-A) &= 0 \\ \frac{h^2}{2}(C+A) &= 1 \end{aligned}$$

Solving this system of equations, (211) is easily reproduced.

8.5 Higher Order Methods

The methods of the previous subsection can be used to derive difference approximations for higher derivatives and/or with higher order error properties. The game plan is always the same: Choose a subset of $n+1$ points that represent a function, fit a polynomial of degree k through these points, and then approximate f', f'', f''', etc., by derivatives of the polynomial evaluated at selected nodes.

Recall that the nth derivative of a polynomial of degree n is a constant. Thus, for a difference scheme to allow a meaningful approximation of the nth derivative, the scheme must be based upon a polynomial of at least one degree higher.

For brevity, we will adopt the convention that $f_i = f(x_i)$. That is, $f_{i+1} = f(x_i+h)$, $f_{i+2} = f(x_i+2h)$, $f_{i-1} = f(x_i-h)$, etc. The following table provides central-difference approximations of $O(h^2)$ accuracy for derivatives up to the fourth. The first two formulas are already familiar.

$$f'(x_i) \approx \frac{f_{i+1}-f_{i-1}}{2h} \tag{221}$$

$$f''(x_i) \approx \frac{f_{i+1}-2f_i+f_{i-1}}{h^2} \tag{222}$$

$$f'''(x_i) \approx \frac{f_{i+2}-2f_{i+1}+2f_{i-1}-f_{i-2}}{2h^3} \tag{223}$$

$$f^{(4)}(x_i) \approx \frac{f_{i+2}-4f_{i+1}+6f_i-4f_{i-1}+f_{i-2}}{h^4} \tag{224}$$

The following central-difference formulas are of $O(h^4)$.

$$f'(x_i) \approx \frac{-f_{i+2}+8f_{i+1}-8f_{i-1}+f_{i-2}}{12h} \tag{225}$$

$$f''(x_i) \approx \frac{-f_{i+2}+16f_{i+1}-30f_i+16f_{i-1}-f_{i-2}}{12h^2} \tag{226}$$

$$f^{(3)}(x_i) \approx \frac{-f_{i+3}+8f_{i+2}-13f_{i+1}+13f_{i-1}-8f_{i-2}+f_{i-3}}{8h^3} \tag{227}$$

$$f^{(4)}(x_i) \approx \frac{-f_{i+3}+12f_{i+2}-39f_{i+1}+56f_i-39f_{i-1}+12f_{i-2}-f_{i-3}}{6h^4} \tag{228}$$

Our final group is comprised of four biased (backward or forward) difference formulas of $O(h^2)$, the first two of which are familiar.

$$f'(x_i) \approx \frac{-3f_i+4f_{i+1}-f_{i+2}}{2h} \tag{229}$$

$$f'(x_i) \approx \frac{+3f_i-4f_{i-1}+f_{i-2}}{2h} \tag{230}$$

$$f''(x_i) \approx \frac{2f_i-5f_{i+1}+4f_{i+2}-f_{i+3}}{h^2} \tag{231}$$

$$f''(x_i) \approx \frac{2f_i-5f_{i-1}+4f_{i-2}-f_{i-3}}{h^2} \tag{232}$$

HW: How much does the error decrease with a method of $O(h^4)$ if h is a) halved, or b) reduced by a factor of 10?

8.6 Application: Finite-Difference Method (Optional)

Although numerical differentiation is useful in its own right, perhaps one of the most powerful uses of difference approximations is as the basis of the *finite-difference method* (FDM), which is used to solve ordinary and partial differential equations of boundary-value type.

An *ordinary differential equation* (ODE) is one that involves a function and one or more of its derivatives. Let's consider a very simple, second-order ODE as given below:

$$y''(x) = f(x) \quad (0<x<1)$$
$$y(0)=a \quad ; \quad y(1)=b$$

The solution of an ODE involves integrating as many times as the order of the derivative. Thus, a second-order ODE involves integrating twice. Each integration introduces a constant of integration. This, for the example problem there are two constants (a and b) that require specification. If those values are specified at the two ends of the domain, then the ODE is classified as a *boundary-value problem* (BVP). If $f(x)$ is easily integrated twice, then the BVP is easily solved.

So, let's now provide some data for an example problem. Suppose $f(x) = 12x^2$, $a = 0$, and $b = 2$. This is easily solved in closed form. That is,

$$\begin{aligned} y''(x) &= 12x^2 \\ \Rightarrow y'(x) &= 4x^3 + c_1 \\ \Rightarrow y(x) &= x^4 + c_1 x + c_2 \end{aligned}$$

The left *boundary condition* $a = 0 \Rightarrow c_2 = 0$. Imposing the right boundary condition implies that $c_1 = 1$. So the exact solution is $y(x) = x^4 + x$.

For this problem we don't really need the FDM, but we would if there were no closed-form antiderivatives of $f(x)$ or if the orginal problem were slightly more complicated or perhaps nonlinear. So let's apply the FDM to our sample problem to see how it works and to see how closely our approximate solution resembles the true one.

STEP 1: The first step is to perform a regular partition of the domain $[0,1]$ into n equal subintervals, each of width $h = 1/n$. The nodes therefore are given by $x_i = ih,\ i = 0,1,2,...,n$. For specificity, suppose $n = 4$. Then

$$\begin{aligned} x_0 &= 0 \\ x_1 &= 1/4 \\ x_2 &= 1/2 \\ x_3 &= 3/4 \\ x_4 &= 1 \end{aligned}$$

STEP 2: The next step is to write the exact ODE at the interior nodes. That is,

$$y''(x_i) = f(x_i) \ \ i = 1,2,3 \tag{233}$$

STEP 3: We now replace $y''(x_i)$ in Eq. 233 by the centered-difference approximation for the second derivative, corrected by the appropriate error formula (Eq. 218) to obtain

$$\frac{y(x_i + h) - 2y(x_i) + y(x_i - h)}{h^2} - \frac{f^{(4)}(\xi_i)}{12}h^2 = f(x_i) \ \ i = 1,2,3 \tag{234}$$

The equation above remains exact because of the inclusion of the error term, apart from the fact that $x_i - h < \xi_i < x_i + h$, the precise location of which we don't know. If h is small, the error term should be relatively insignificant. So let's just ignore it and see what happens. STEP 4: Ignoring the error term will likely produce an inexact solution. For brevity let $y_i \approx y(x_i)$; that is, y_i is the approximate solution for $y(x_i)$. Thus, y_i is the solution to

$$\frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} = f(x_i) \ \ i = 1,2,3 \tag{235}$$

where, keep in mind, that $y_{i+1} \approx y(x_{i+1}) = y(x_i + h)$ and similarly for y_{i-1}. Let's now unroll (write out explicitly) the three equations represented by Eq. 235 for $i = 1$, $i = 2$, and $i = 3$. To these three equations

we will append two trivial equations for the boundary values. We get

$$\begin{aligned} y_0 &= a \\ y_0 - 2y_1 + y_2 &= h^2 f_1 \\ y_1 - 2y_2 + y_3 &= h^2 f_2 \\ y_2 - 2y_3 + y_4 &= h^2 f_3 \\ y_4 &= b \end{aligned} \tag{236}$$

We have taken a few liberties to obtain the equation system Eq. 236. The first is to abbreviate $f_i = f(x_i)$. The second is to multiply the middle three equations by h^2 to clear denominators. Equations 236 should look familiar. They comprise a linear system for the five unknowns y_i, $\{i = 0,1,2,3,4\}$ that approximate the solution at the nodes. STEP 5: This system can be written in matrix-vector form as

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} a \\ h^2 f_1 \\ h^2 f_2 \\ h^2 f_3 \\ b \end{bmatrix}$$

Yet more succinctly, we can write the linear system as

$$T\vec{y} = \vec{b} \tag{237}$$

where T is a 5×5 tridiagonal matrix, $\vec{y}$ is a 5-vector containing the unknowns, and $\vec{b}$ is a 5-vector containing the right-hand-side values and the boundary conditions.

You may be thinking that the boundary values y_0 and y_4 are given by the boundary conditions, and so it is silly to consider them as unknowns. If so, you are correct. The 5×5 system can be collapsed to the following 3×3 system.

$$\begin{aligned} a - 2y_1 + y_2 &= h^2 f_1 \\ y_1 - 2y_2 + y_3 &= h^2 f_2 \\ y_2 - 2y_3 + b &= h^2 f_3 \end{aligned} \tag{238}$$

(239)

If we migrate the known values a and b from the left to right side of the equation, we obtain the following matrix-vector form:

$$\begin{bmatrix} -2 & 1 & 0 \\ 1 & -2 & 1 \\ 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} h^2 f_1 - a \\ h^2 f_2 \\ h^2 f_3 - b \end{bmatrix}$$

STEP 6: Solving the resulting matrix system by the methods of linear algebra is the last step.

Finally, let's give the FDM a trial run for the data of the example problem: $f(x) = 12x^2$, $a = 0$, $b = 2$, and $n = 4$, in which case the resulting system is

$$\begin{bmatrix} -2 & 1 & 0 \\ 1 & -2 & 1 \\ 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} (1/4)^2(12)(1/4)^2 - 0 \\ (1/4)^2(12)(2/4)^2 \\ (1/4)^2(12)(3/4)^2 - 2 \end{bmatrix}$$

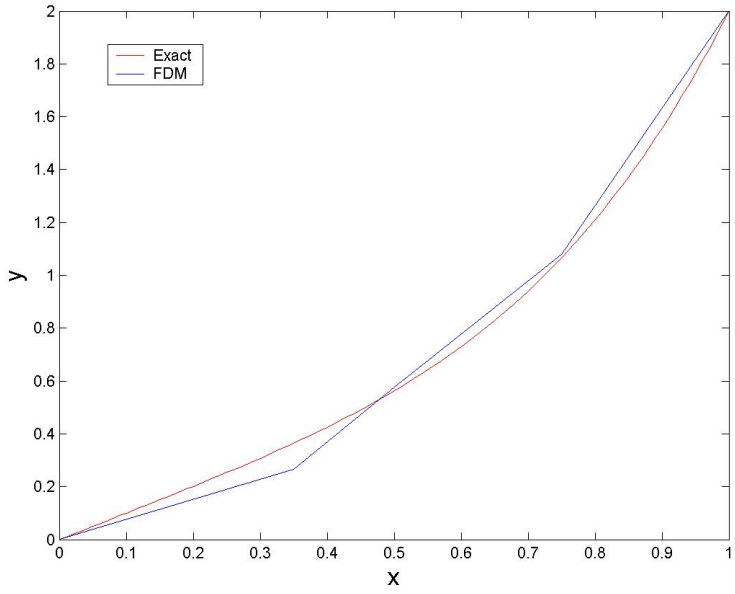


Figure 30: Finite-difference solution vs. exact solution.

Recall that triadiagonal systems can be solved very efficiently in $O(n)$ operations by tridiagonal Gaussian elimination.

Figure 30 compares the discrete solution of the FDM to the exact continuous solution. The FDM solution is inexact at the nodes because $f^{(4)}(x) = 24$, so that the neglected truncation error is significant. However, as h gets smaller (n becomes larger) the error tends toward zero and the FDM solution tends toward the exact solution.

HW: Solve the following exactly and by the FDM for $n = 4$. Compare the results at the nodes. Do the methods agree? Should they?

$$\begin{aligned} y''(x) &= 1 \quad ; \quad 0 < x < 1 \\ y(0) &= -1 \quad ; \quad y(1) = 0 \end{aligned} \tag{240}$$

Finally, a comment of perspective. It may seem that the FDM presented above is a "toy" mathematical approach of little practical significance. Nothing could be further from the truth. The FDM is a practical method that has been used to solve the ODEs and PDEs that original for a wide variety of scientific and engineering problems. In particular, for example, the FDM is used to predict heat flow in engines and airflow over aircraft wings and fuselages in order that processes can be understood and designs improved.

8.7 Exercises

1. Use the most appropriate three-point formula to determine approximations that will complete the following table.

x	$f(x)$	$f'(x)$
0.0	0.0000000	
0.2	0.74140	
0.4	1.3718	

The data above was taken from the function $f(x) = e^x - 2x^2 + 3x - 1$. Find the error bounds using the appropriate error formulas for the above approximations for (a) $x = 0.0$ and (b) $x = 0.2$.

9 Part II: Numerical Integration

Recall the (First) Fundamental Theorem of Calculus from Math 235: If $f(x)$ is an integrable function, and $F(x)$ is *any* antiderivative of $f(x)$, then

$$\int_a^b f(x)dx = F(b) - F(a) \tag{241}$$

This amazing theorem gives us a powerful tool for evaluating definite integrals, or from a geometric point of view, finding the exact area of regions bounded by the graphs of functions. Perhaps a third of the semester of Math 236 was devoted to antidifferentiation techniques: u-substitution, trig. substitution, integration by parts, and so on.

Math 236 may have left you with a false sense of security, the feeling that if you are clever enough and/or you know enough tricks, given $f(x)$, you can always find an antiderivative $F(x)$. Not so.

Consider the following definite integral:

$$\int_0^{\frac{\pi}{2}} \frac{1}{\sqrt{1-k^2\sin^2(x)}}dx \tag{242}$$

If $k = 1$, then the integral is trivial to evaluate. However, if $0 < k < 1$, there is *no* closed-form antiderivative. What does this mean? Well, first of all, the integrand is well-behaved; it is continuous and thus integrable. So the definite integral is well-defined. However, all the usual analytical methods of determining the area under the curve fail.

The example is a type of integral known as an elliptic integral of the second kind (sorta like *Close Encounters of the Third Kind*.) In general, elliptic integrals cannot be evaluated by standard analytical methods. What is to be done then? Numerics to the rescue; hence, the purpose of this Chapter.

Before we proceed, a quick word on nomenclature. A somewhat archaic term for numerical integration is *quadrature*. Thus, we will use the terms *numerical integration* and *quadrature* interchangeably.

9.1 A Review of the Riemann Integral

Most calculus texts develop the notion of the definite integral via the Riemann sum, in which the sum of rectangular areas is used to approximate the area "under the curve." We review this powerful idea.

Consider function $f(x)$ defined on the interval $[a,b]$. The Riemann sum begins with a partition P, not necessarily regular, of the interval such that

$$\begin{aligned} P &= \{x_0, x_1, x_2, ..., x_{n-1}, x_n\} \\ a &= x_0 < x_1 < x_2 < ... < x_{n-1} < x_n = b \end{aligned}$$

Further define the subinterval widths as follows:

$$\begin{aligned}
\Delta x_1 &= x_1 - x_0 \\
\Delta x_2 &= x_2 - x_1 \\
\Delta x_3 &= x_3 - x_2 \\
&\ldots \\
\Delta x_{n-1} &= x_{n-1} - x_{n-2} \\
\Delta x_n &= x_n - x_{n-1}
\end{aligned}$$

or, in general, $\Delta x_i = x_i - x_{i-1}$. Furthermore, within each subinterval i, arbitrarily choose a point to be identified as x_i^*. That is,

$$\begin{aligned}
x_0 \leq \quad & x_1^* \quad \leq x_1 \\
x_1 \leq \quad & x_2^* \quad \leq x_2 \\
x_2 \leq \quad & x_3^* \quad \leq x_3 \\
& \ldots \\
x_{n-2} \leq \quad & x_{n-1}^* \quad \leq x_{n-1} \\
x_{n-1} \leq \quad & x_n^* \quad \leq x_n
\end{aligned}$$

Now let A_i denote the area of a rectangle of height $f(x_i^*)$ and of width Δx_i as shown in Fig. 31; that is,

$$A_i = f(x_i^*)\Delta x_i \tag{243}$$

in which case the area "under the curve" is approximated by the sum of the area of rectangles, as follows:

$$\begin{aligned}
\int_a^b f(x)dx &= \text{area under the curve} \approx \sum_{i=1}^{n} A_i \\
\int_a^b f(x)dx &\approx \sum_{i=1}^{n} f(x_i^*)\Delta x_i
\end{aligned} \tag{244}$$

The sum on the right-hand-side of Eq. 244 is called a *Riemann Sum*, after Bernhard Riemann, who was a student of Gauss at the University of Goettingen, and a formidable mathematician in his own right. Thus far, the Riemann sum only approximates the definite integral. However, if $n \to \infty$ in such a way that $\Delta x_i \to 0$ for all i, then the infinite sum gives the exact value of the integral. We can put it all together succinctly in the following definition of the definite integral:

$$\int_a^b f(x)dx \equiv \lim_{(\max \Delta x_i \to 0)} \sum_{i=1}^{n} f(x_i^*)\Delta x_i \tag{245}$$

The theory of the Riemann integral is quite general. It allows us to partition the interval $[a,b]$ any way we please, and furthermore, it allows us to pick the points x_i^* within the subintervals any way we please. Thus the theory still holds if we consider a regular partition, by which $\Delta x_i = h = \frac{b-a}{n}$ and $x_i = a + ih,\ i = 0,1,2,\ldots,n$. Note that as $n \to \infty$, $h \to 0$. Thus, the exact integral can be expressed as

$$\int_a^b f(x)dx = \lim_{n\to\infty} \sum_{i=1}^{n} f(x_i^*)h = \lim_{n\to\infty} h \sum_{i=1}^{n} f(x_i^*) \tag{246}$$

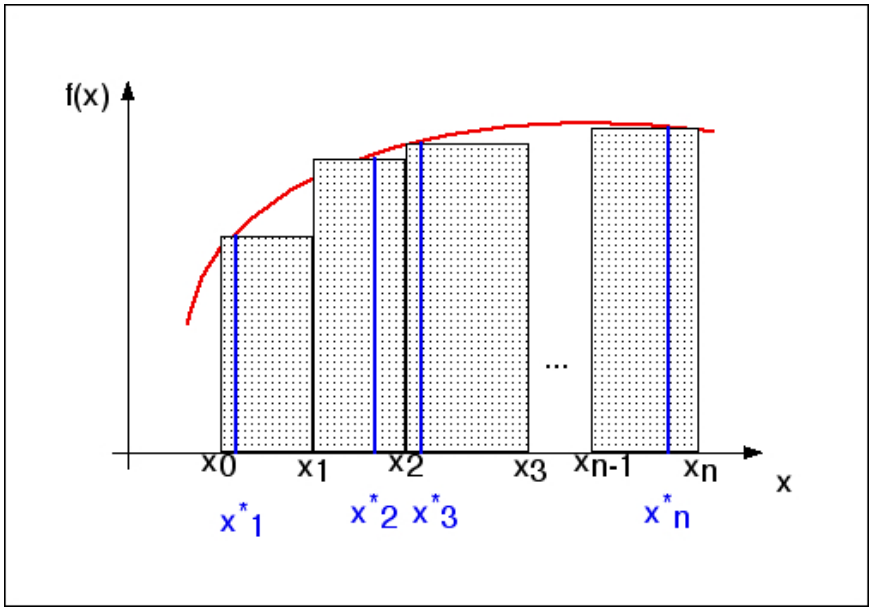


Figure 31: The Riemann integral.

Of course, we cannot evaluate Eq. 246 exactly on a computer because even the fastest computer cannot perform an infinite sum. Once again, we must settle for a finite sum and an approximate result.

9.2 Numerical Integration Rules

Let's keep the idea of a regular partition, but drop the infinite sum. Where does that leave us?

$$\int_a^b f(x)dx \approx h\sum_{i=1}^{n} f(x_i^*) \tag{247}$$

Note that we are still free to choose $x_{i-1} \leq x_i^* \leq x_i$ as we please. Are some choices better than others?

9.2.1 Composite Right and Left Rectangle Rules

Let's be naive and pick x_i^* to lie at one of the endpoints of the subinterval. For example, if we pick x_i^* to coincide with the left endpoint of each subinterval (that is, $x_i^* = x_{i-1}$), then for Eq. 247

$$\int_a^b f(x)dx \approx h\sum_{i=1}^{n} f(x_{i-1}) = h\sum_{i=0}^{n-1} f(x_i) \equiv L(h) \tag{248}$$

where $L(h)$ indicates that the method is commonly called the *left rectangle rule*, whose value depends upon the step-size $h = (b-a)/n$. The left rectangle rule is illustrated in Fig. 32. Note that $L(h)$ underpredicts the true area if the function is increasing. It would, of course, overpredict the true area were the function decreasing.

Alternately, we could have chosen x_i^* to coincide with the right endpoint of the subinterval, in which case $x_i^* = x_i$, and Eq. 247 gives

$$\int_a^b f(x)dx \approx h\sum_{i=1}^{n} f(x_i) \equiv R(h) \tag{249}$$

where $R(h)$ denotes the *right rectangle rule*, whose value also depends upon h. The right rectangle rule is illustrated in Fig. 33. Note that $R(h)$ overpredicts the true area if the function is increasing. Conversely, it would underpredict the true area were the function decreasing.

9.2.2 Composite Midpoint Rule

By now it is clear that some choices for x_i^* are better than others if one is restricted by finite h. One way to balance the tendencies to over- or undershoot the true area is to choose x_i^* to fall at the midpoint of each subinterval. That is,

$$x_i^* = \frac{x_{i-1} + x_i}{2} \equiv m_i \tag{250}$$

in which case

$$\int_a^b f(x)dx \approx h\sum_{i=1}^{n} f(m_i) \equiv M(h) \tag{251}$$

Here, $M(h)$ denotes the *midpoint rule*, whose value depends on h. Midpoint-rule quadrature is illustrated in Fig. 34. Let's compare $L(h)$, $R(h)$, and $M(h)$ on a trial problem.

EX: Recall from Math 235 that $\ln(x) = \int_1^x \frac{1}{t}dt$. Use the numerical integration schemes $L(h)$, $R(h)$, and $M(h)$ with $n = 4$ to approximate $\ln(3)$. That is, use the three quadrature methods to approximate $\int_1^3 \frac{1}{x}dx$, which is shown in Fig 35. With definite integrals, it is always a good practice to estimate the area first by eye, as a ballpark check on your computed answer. It appears that the area is a bit more than 1, say 1.2, square units.

For each method $n = 4 \Rightarrow h = \frac{3-1}{4} = \frac{1}{2}$. NOTE: The number of nodes is always one greater than the number of subintervals n. In this instance, there are five nodes: x_0, x_1, x_2, x_3, and x_4. The data are best organized in tabular form. We should explain the purpose of the fourth and fifth columns of Table 17.

i	x_i	$f(x_i) = \frac{1}{x_i}$	$(wL)_i$	$(wR)_i$
0	1.0	1.0	1	0
1	1.5	$0.6666\overline{6}$	1	1
2	2.0	0.5	1	1
3	2.5	0.4	1	1
4	3.0	$0.3333\overline{3}$	0	1

Table 17: Numerical integration by $L(h)$ and $R(h)$.

These columns contain the *weights* of the left and right rectangle rules, respectively. Note that, whereas $L(h)$ uses functional values at nodes 0 to $n-1$, $R(h)$ makes use of the functional values at notes 1 to n. The

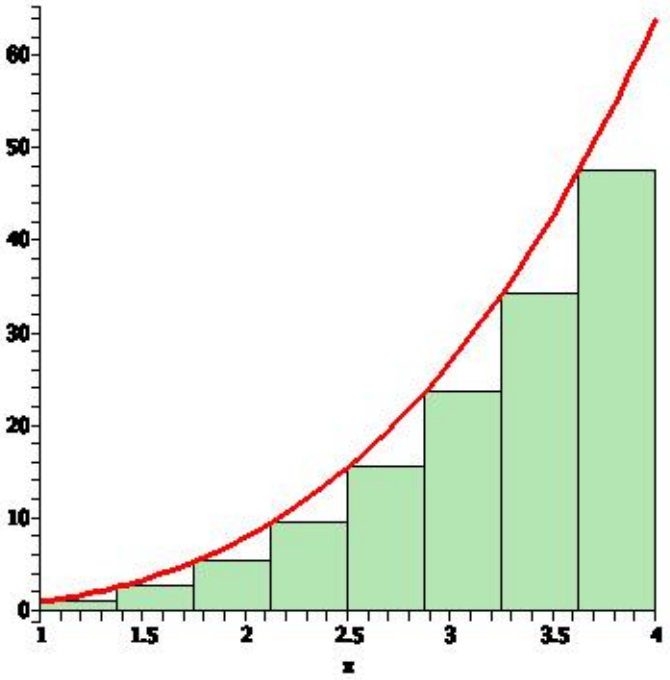


Figure 32: Composite left rectangle rule for $f(x) = x^3$ with $n = 8$.

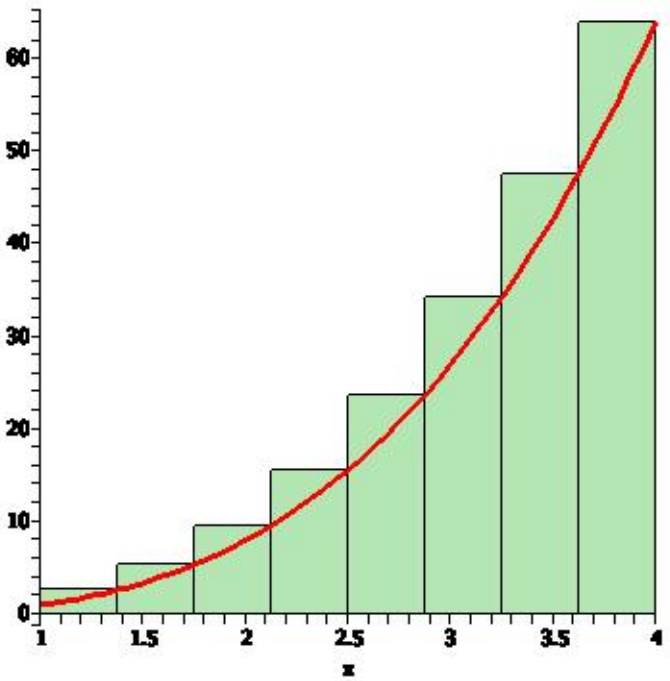


Figure 33: Composite right rectangle rule for $f(x) = x^3$ with $n = 8$.

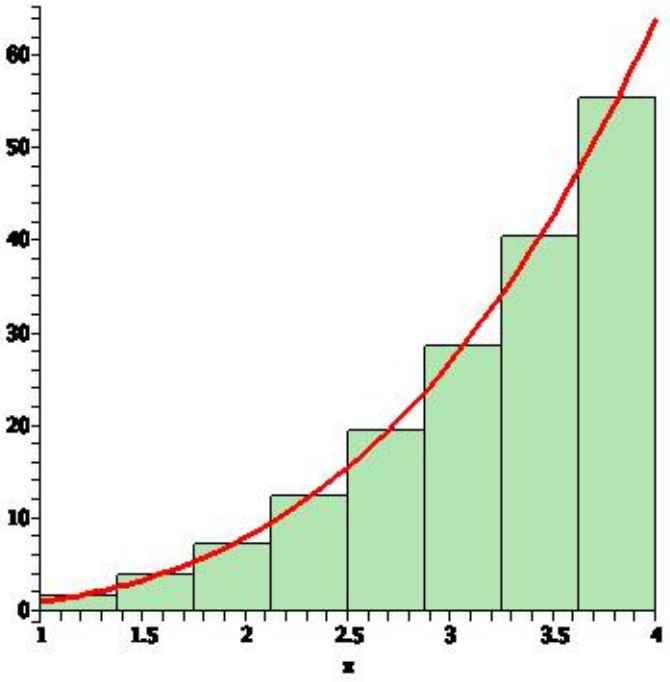


Figure 34: Composite midpoint rule for $f(x) = x^3$ with $n = 8$.

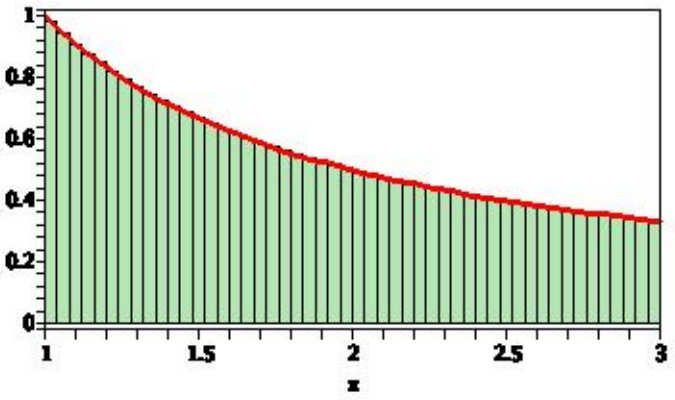


Figure 35: Computing $\ln(x)$ as area under curve.

columns denoted wL_i and wR_i, respectively, indicate which nodes are "turned on" and which are "turned off" for the particular method. For example, in the column headed wR_i, by weights of "0" or "1" for "off" and "on", respectively, we see that the method $R(h)$ uses the functional values at nodes 1-4, but does not make use of $f(x_0)$. By this convention, we can use a single algorithm for both methods $L(h)$ and $R(h)$, namely

$$I \approx h \sum_{i=0}^{n} f_i w_i \tag{252}$$

where $I = \int_a^b f(x)dx$, f_i is shorthand for $f(x_i)$, and w_i is the value of the weight for node i. For example, expanding Eq. 252 for $R(h)$, we get

$$I \approx h \sum_{i=0}^{4} f_i (wR)_i = 0.5[1.0(0) + 0.6666\overline{6}(1) + 0.5(1) + 0.4(1) + 0.3333\overline{3}(1)] = 0.95 = R(0.5) \tag{253}$$

Similarly, the corresponding expansion for $L(h)$ is

$$I \approx h \sum_{i=0}^{4} f_i (wL)_i = 0.5[1.0(1) + 0.6666\overline{6}(1) + 0.5(1) + 0.4(1) + 0.3333\overline{3}(0)] = 1.2833\overline{3} = L(0.5) \tag{254}$$

A disadvantage of midpoint rule is that it is difficult to use the same algorithm we developed for $L(h)$ and $R(h)$. This is because $M(h)$ uses the midpoints of the subintervals, which must be calculated, rather than the original nodal values. Thus, the midpoint rule requires a new table of data. Expanding the rule

i	m_i	$f(m_i) = \frac{1}{m_i}$	$(wM)_i$
1	1.25=5/4	4/5	1
2	1.75=7/4	4/7	1
3	2.25=9/4	4/9	1
4	2.75=11/4	4/11	1

Table 18: Numerical integration by $M(h)$.

$M(h)$ for the data at hand, we get

$$I \approx h \sum_{i=1}^{4} f_i (wM)_i = 0.5[4/5(1) + 4/7(1) + 4/9(1) + 4/11(1)] = 1.08975 = M(0.5) \tag{255}$$

To six significant digits, the true value of $\ln(3) = 1.09861$. As expected $L(0.5)$ considerably overpredicts this value, and $R(0.5)$ considerably underpredicts. In contrast, $M(0.5)$ is correct to three significant digits. We summarize the error in Table 19.

The midpoint rule was the most accurate, but it was also awkward. If you are thinking that perhaps we could have gotten a better estimate more readily by averaging $L(0.5)$ and $R(0.5)$, then pat yourself on the back. Indeed,

$$\begin{aligned} T(0.5) &= \frac{L(0.5) + R(0.5)}{2} = 1.1666\overline{6} \\ E_T(0.5) &= -0.0180543 \end{aligned} \tag{256}$$

$L(0.5)$	$E_L(0.5)$	$R(0.5)$	$E_R(0.5)$	$M(0.5)$	$E_M(0.5)$
$1.28333\overline{3}$	-0.184721	0.950000	0.148612	1.08975	0.008862

Table 19: Quadrature errors compared.

where the rule $T(h)$ will be explained in full shortly. Note that $T(0.5)$ is considerably better than $L(0.5)$ or $R(0.5)$ but not as accurate as $M(0.5)$. The really careful observer, however, may recognized that $E_T(0.5)$ is opposite in sign to $E_M(0.5)$ and almost exactly twice as large in magnitude. This suggests a wild possibility: a weighted combination of $T(0.5)$ and $M(0.5)$ designed to eliminate most of the error. In particular, let's try

$$S(0.5) = \frac{2M(0.5)+T(0.5)}{3} = 1.09872\overline{2} \tag{257}$$

The value above is correct to four significant digits! Is this a fluke, or is something magical happening? Stay tuned.

9.2.3 Composite Trapezoid Rule

Let's take a closer look at what happened when we averaged left and right rectangle rules above. Recall that we defined this average as $T(h)$. That is,

$$\begin{aligned} T(h) &= \frac{1}{2}[L(h)+R(h)] \\ &= \frac{1}{2}[h\sum_{i=0}^{n-1} f(x_i) + h\sum_{i=1}^{n} f(x_i)] \\ &= \frac{h}{2}[f_0+2f_1+2f_2+...+2f_{n-1}+f_n] \\ &= \frac{h}{2}[(wT)_0 f_0+(wT)_1 f_1+(wT)_2 f_2+...+(wT)_{n-1} f_{n-1}+(wT)_n f_n] \end{aligned} \tag{258}$$

where

$$wT_i = \begin{cases} 2 \text{ if } i \neq 0 \text{ and } i \neq n \\ 1 \text{ if } i = 0 \text{ or } i = n \end{cases}$$

The rule $T(h)$ is known as *trapezoid rule* for reasons to become apparent shortly. We summarize by developing an algorithm that can be used for trapezoid rule, left rectangle rule, or right rectangle rule, depending upon the weights chosen.

Key to the common algorithm is the recognition that each of these quadrature rules can be expressed as a scaled dot product of two $(n+1)$-vectors: a vector of functional values and a vector of weights. In particular, let

$$\vec{f} = [f_0, f_1, f_2, ..., f_{n-1}, f_n] \tag{259}$$

and

$$\vec{w} = [w_0, w_1, w_2, ..., w_{n-1}, w_n] \tag{260}$$

Further let $N(h)$ denote the numerical approximation of $I = \int_a^b f(x)dx$ by one of these three methods. Then

$$N(h) = \frac{h}{c}\vec{f} \cdot \vec{w} \tag{261}$$

where

$$\begin{aligned} N &\in \{L, R, T\} \\ \vec{w} &\in \{\vec{w}_L, \vec{w}_R, \vec{w}_T\} \end{aligned}$$

c is a scaling constant, namely

$$c = \begin{cases} 1 \text{ if } N = L \text{ or } N = R \\ 2 \text{ if } N = T \end{cases}$$

and

$$\begin{aligned} \vec{w}_L &= [1,1,1,...,1,1,0] \text{ (for } L(h)) \\ \vec{w}_R &= [0,1,1,...,1,1,1] \text{ (for } R(h)) \qquad (262) \\ \vec{w}_T &= [1,2,2,...,2,2,1] \text{ (for } T(h)) \end{aligned}$$

(263)

ALGORITHM 9.1: Numerical integration by L, R, T, and S methods

given function $f(x)$ (user-defined function subprogram)
input limits of integration a, b for interval $[a,b]$
input number of subintervals n
$h \leftarrow (b-a)/n$

for $i = 0,1,2,...,n$
| $x_i \leftarrow a + ih$
| $f_i \leftarrow f(x_i)$
| ___

define $n+1$ vector of weights w_i (according to method)
define scaling constant c (according to method)
$N \leftarrow 0$ *(initialize accumulator)*

repeat for $i = 0,1,2,...,n$ (performs dot product)
| $N \leftarrow N + w_i f_i$
| ___

$N \leftarrow Nh/c$ *(scale according to method)*
output N (which contains approximation of I)

REMARKS:

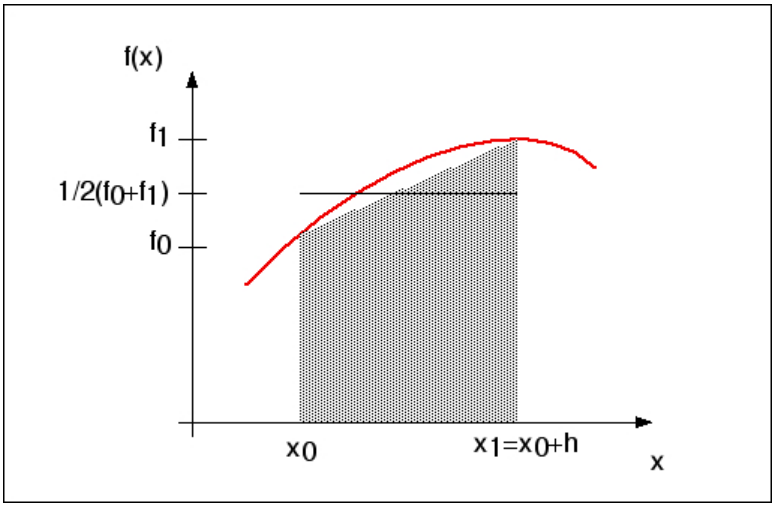


Figure 36: Area of trapezoid.

1. The algorithm above is presented as a stand-alone program, but it should really be implemented as a function. Why?

Let's now return to the trapezoid rule and find out how it got its name. First, note that the area of the trapezoid shown in Fig. 36 is given by the product of the base times the average height. That is,

$$A_T = h\left[\frac{f_0+f_1}{2}\right] \tag{264}$$

With this in mind, let's revisit Eq. 258.

$$\begin{aligned} T(h) &= \frac{h}{2}[f_0+2f_1+2f_2+\ldots+2f_{n-1}+f_n] \\ &= h\left[\frac{f_0+f_1}{2}\right]+h\left[\frac{f_1+f_2}{2}\right]+\ldots+h\left[\frac{f_{n-1}+f_n}{2}\right] \\ &= \sum_{i=1}^{n}(A_T)_i \end{aligned} \tag{265}$$

Geometrically, the sum above results from the approximation of the area under the curve by the use of trapezoids rather than the rectangles. Alternately, the trapezoid rule can be derived from first principles by returning to the notion of the interpolating polynomial from Chapter 7.

Here is the game plan for deriving the trapezoid rule from scratch: 1) approximate the function $f(x)$ on subinterval $[x_{i-1},x_i]$ by linear polynomial $P_1(x)$, 2) approximate $\int_{x_{i-1}}^{x_i} f(x)dx$ by $\int_{x_{i-1}}^{x_i} P_1(x)dx$, and 3) sum the approximated areas on each subinterval.

For starters, consider the linear interpolating polynomial $P_1(x)$ in Lagrange form for the first subinterval $[x_0,x_1]$, namely

$$P_1(x) = f_0\frac{x-x_1}{x_0-x_1}+f_1\frac{x-x_0}{x_1-x_0}$$

$$= -\frac{f_0}{h}(x-x_1)+\frac{f_1}{h}(x-x_0) \tag{266}$$

By integrating the expression above from x_0 to x_1, we obtain

$$\begin{aligned}
\int_{x_0}^{x_1} P_1(x) &= -\frac{f_0}{h}\int_{x_0}^{x_1}(x-x_1)dx+\frac{f_1}{h}\int_{x_0}^{x_1}(x-x_0)dx \\
&= -\frac{f_0}{h}\frac{(x-x_1)^2}{2}|_{x_0}^{x_1}+\frac{f_1}{h}\frac{(x-x_0)^2}{2}|_{x_0}^{x_1} \\
&= \frac{f_0}{h}\frac{(x_0-x_1)^2}{2}+\frac{f_1}{h}\frac{(x_1-x_0)^2}{2} \\
&= \frac{f_0}{h}\frac{h^2}{2}+\frac{f_1}{h}\frac{h^2}{2} \\
&= \frac{h}{2}[f_0+f_1]
\end{aligned} \tag{267}$$

Linear interpolation in each subinterval is equivalent to estimating the area under the curve within each subinterval by the area of a trapezoid, as revealed by the final expression in Eq. 267. Strictly speaking, what we have derived is the *simple* trapezoid rule, which applies to a single subinterval. *Composite* quadrature rules are derived by extending the corresponding simple quadrature rule to all subintervals. Thus, the composite trapezoid rule is obtained by the following logic, presuming there are n subintervals each of width h:

$$\begin{aligned}
\int_a^b f(x)dx &= \int_{x_0}^{x_1} f(x)dx+\int_{x_1}^{x_2} f(x)dx+...+\int_{x_{n-1}}^{x_n} f(x)dx \\
&\approx \int_{x_0}^{x_1} P_1(x)dx+\int_{x_1}^{x_2} P_1(x)dx+...+\int_{x_{n-1}}^{x_n} P_1(x)dx \\
&= \frac{h}{2}[f_0+f_1]+\frac{h}{2}[f_1+f_2]+...+\frac{h}{2}[f_{n-1}+f_n] \\
&= \frac{h}{2}[f_0+2f_1+2f_2+...+2f_{n-1}+f_n] = T(h)
\end{aligned}$$

We have taken some liberties with notation in the derivation above. Note that the linear polynomial interpolant is formally the same but numerically *different* in each subinterval. For example, in subinterval 1, $P_1(x) = f_0\frac{x-x_1}{x_0-x_1}+f_1\frac{x-x_0}{x_1-x_0}$, but in subinterval 2, $P_1(x) = f_1\frac{x-x_2}{x_1-x_2}+f_2\frac{x-x_1}{x_2-x_1}$. It would be notationally cumbersome to distinquish the linear interpolants in each subinterval, and so we have (mis)used the notation $P_1(x)$ for each subinterval.

Now that you have the idea of how quadrature rules can be derived from scratch, we are ready to tackle Simpson's rule, an extraordinarily accurate quadrature rule.

9.2.4 Simpson's Rule

We begin by considering the quadratic interpolant in Langrange form for three equally spaced nodes–x_0, x_1, and x_2–with adjacent nodes separated by subinterval width h.

$$\begin{aligned} P_2(x) &= f_0\frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} + f_1\frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} + f_2\frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} \\ &= f_0L_0(x) + f_1L_1(x) + f_2L_2(x) \end{aligned} \tag{268}$$

Thus,

$$\begin{aligned} \int_{x_0}^{x_2} f(x)dx &\approx \int_{x_0}^{x_2} P_2(x)dx \\ &= f_0\int_{x_0}^{x_2} L_0(x) + f_1\int_{x_0}^{x_2} L_1(x) + f_2\int_{x_0}^{x_2} L_2(x) \end{aligned} \tag{269}$$

You might be wondering why we have used the Lagrange form of the interpolant rather than the Newton form. There is a very good reason. Note that, in the Lagrange form, the weight w_0 for function value f_0, for example, is given by the integral of $L_0(x)$, and similarly for weights w_1 and w_2. For example,

$$\begin{aligned} w_0 &= \int_{x_0}^{x_2} L_0(x)dx \\ &= \int_{x_0}^{x_2} \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}dx \\ &= \int_{x_0}^{x_2} \frac{(x-x_1)(x-x_2)}{(-h)(-2h)}dx \\ &= \frac{1}{2h^2}\int_{x_0}^{x_2} (x-x_1)(x-x_2)dx \\ &= \frac{1}{2h^2}\int_{x_0}^{x_2} (x-x_1)[x-(x_1+h)]dx \\ &= \frac{1}{2h^2}\int_{x_0}^{x_2} (x-x_1)[(x-x_1)-h)]dx \\ &= \frac{1}{2h^2}\int_{x_0}^{x_2} [(x-x_1)^2 - h(x-x_1)]dx \end{aligned}$$

At this point, a change of variables is helpful (but not absolutely essential). Let $t = x - x_1 \Rightarrow x = t + x_1$, and $dt = dx$. Moreover, $x_0 \le x \le x_2 \Rightarrow -h \le t \le h$. Thus,

$$\begin{aligned} w_0 &= \frac{1}{2h^2}\int_{-h}^{+h} [t^2 - ht]dt \\ &= \frac{1}{2h^2}\left[\frac{t^3}{3} - \frac{ht^2}{2}\right]_{-h}^{+h} \\ &= \frac{1}{2h^2}\left[\frac{h^3}{3} - \frac{h^3}{2} - \left(-\frac{h^3}{3} - \frac{h^3}{2}\right)\right] \\ &= \frac{h}{3} \end{aligned} \tag{270}$$

Similarly, $w_1 = \int_{x_0}^{x_2} L_1(x)dx = \frac{4h}{3}$, and $w_2 = \int_{x_0}^{x_2} L_2(x)dx = \frac{h}{3}$, in which case

$$\begin{aligned}\int_{x_0}^{x_2} P_2(x)dx &= f_0\frac{h}{3} + f_1\frac{4h}{3} + f_2\frac{h}{3} \\ &= \frac{h}{3}[f_0 + 4f_1 + f_2] \end{aligned} \tag{271}$$

Equation 271, in fact, is the simple Simpson's quadrature rule. To derive composite Simpson's rule, we apply the simple rule to pairs of subintervals. That is,

$$\begin{aligned}\int_a^b f(x)dx &= \int_{x_0}^{x_2} f(x)dx + \int_{x_2}^{x_4} f(x)dx + ... + \int_{x_{n-2}}^{x_n} f(x)dx \\ &\approx \int_{x_0}^{x_2} P_2(x)dx + \int_{x_2}^{x_4} P_2(x)dx + ... + \int_{x_{n-2}}^{x_n} P_2(x)dx \\ &= \frac{h}{3}[f_0 + 4f_1 + f_2] + \frac{h}{3}[f_2 + 4f_3 + f_4] + ... + \frac{h}{3}[f_{n-2} + 4f_{n-1} + f_n] \\ &= \frac{h}{3}[f_0 + 4f_1 + 2f_2 + 4f_3 + ... + 2f_{n-2} + 4f_{n-1} + f_n] = S(h) \end{aligned} \tag{272}$$

Composite Simpson's rule is readily added to the Pantheon of methods embraced by ALG 9.1 by defining the scaling constant as $c = 3$ and the weight vector as $\vec{w}_S = [1,4,2,4,2,4,...,2,4,1]$.

Now that you are familiar with ALG 9.1 for numerical integration, we can even make it work for the composite midpoint rule, but it requires a slight trick. Consider that the odd-indexed nodes (e.g., x_1, x_3, x_5, etc.) are the midpoints of subintervals of widths $2h$ that span from one even-indexed node to the next. For example, x_1 is the midpoint of the subinterval $[x_0, x_2]$. Provided we have an even number of subintervals (an odd number of nodes), we can adapt ALG 9.1 by defining the weights for the midpoint rule to be $\vec{w}_M M = [0,1,0,1,0,...,1,0]$, in which case

$$M(2h) = 2h\vec{f} \cdot \vec{w}_M \tag{273}$$

whereby the scaling constant $c = 0.5$.

Finally, it can be shown that the following weighted average of the composite trapezoid rule and the composite midpoint rule yields Simpson's rule:

$$S(h) = \frac{2T(h) + M(2h)}{3} \tag{274}$$

Simpson's rule, like the midpoint rule above, and unlike the trapezoid rule, for example, requires an even number of subintervals.

9.3 Quadrature Error

To derive the integration error, we will make use of the formula for the error of polynomial interpolation in Chapter 7. The game plan is roughly as follows:

$$f(x) \approx P_n(x) \;\Rightarrow\; \int_a^b f(x)dx \approx \int_a^b P_n(x)dx \tag{275}$$

Consequently,

$$\begin{aligned} E(h) &= \int_a^b f(x)dx - \int_a^b P_n(x)dx \\ &= \int_a^b [f(x) - P_n(x)]dx \\ &= \int_a^b e_n^P(x)dx \end{aligned} \tag{276}$$

That is, the integration error is the integral of the interpolation error given by Eq. 181. We begin by deriving the error of the simple quadrature rules and later modify the results for composite rules. But to do justice to the derivation, we will first need an integral theorem from Math 235.

Second Integral Mean Value THEOREM (IMVT2): If f and g are continuous on $[a,b]$ and g is non-negative (or non-positive) on the interval, then there exists $c \in (a,b)$ for which

$$\int_a^b f(x)g(x)dx = f(c)\int_a^b g(x)dx \tag{277}$$

The value $f(c)$ is known as the *g-weighted average* of f on $[a,b]$.

9.3.1 Rectangle Rule Error

Consider either left or right rectangle rule on a single subinterval, for which $[x_0,x_1] = [a,b]$. Rectangle rules are crude in that they exploit polynomial interpolation of degree $n = 0$, by which $f(x) \approx P_0(x) =$ constant and there is but a single node. For left rectangle rule the constant is $f(x_0)$, and the node is x_0. For the right rectangle rule the constant is $f(x_1)$, and the node is x_1. For the former, the absolute value of the integration error is given by

$$|E_{SL}(h)| = \left|\int_{x_0}^{x_1} e_0(x)dx\right| = \left|\int_{x_0}^{x_1} f'(\xi)(x-x_0)dx\right| \tag{278}$$

In the equation above, $x_0 < \xi < x \;\Rightarrow \xi = \xi(x)$, and $g(x) = x - x_0$, which is non-negative on the entire interval. Thus, by the IMVT2 above,

$$\begin{aligned} |E_{SL}(h)| &= \left|f'(c)\int_{x_0}^{x_1}(x-x_0)dx\right| \\ &\leq M_1\left|\int_{x_0}^{x_1}(x-x_0)dx\right| \end{aligned}$$

$$
\begin{aligned}
&= M_1 \left| \frac{(x-x_0)^2}{2} \right|_{x_0}^{x_1} \\
&= M_1 \left| \frac{(x_1-x_0)^2}{2} \right| \\
&= \frac{M_1 h^2}{2}
\end{aligned} \tag{279}
$$

where the subscript SL refers to the simple left rectangle rule, and M_1 is an upper bound on the absolute value of the first derivative of f on the interval.

The derivation for $E_{SR}(h)$ is virtually identical to that above. Because the node is x_1 rather than x_2, the factor $-h^2/2$ appears within the absolute value signs. Still, the final result is the same as for left rectangle rule, namely,

$$
|E_{SR}(h)| \leq \frac{M_1 h^2}{2} \tag{280}
$$

9.3.2 Trapezoidal Rule Error

To derive the error formula for simple trapezoid rule (ST), we proceed exactly as before, however, with the error formula appropriate for linear $(n = 1)$ interpolation, in which case

$$
|E_{ST}(h)| = \left| \int_{x_0}^{x_1} e_1(x)dx \right| = \left| \int_{x_0}^{x_1} \frac{f''(\xi)}{2}(x-x_0)(x-x_1)dx \right| \tag{281}
$$

In the equation above, $g(x) = \psi_2(x) = (x-x_0)(x-x_1)$. Here, g is non-positive on the entire interval, which once again satisfies the hypothesis of the IMVT2, in which case

$$
\begin{aligned}
|E_{ST}(h)| &= \left| \frac{f''(c)}{2} \int_{x_0}^{x_1} (x-x_0)(x-x_1)dx \right| \\
&\leq \frac{M_2}{2} \left| \int_{x_0}^{x_1} (x-x_0)(x-x_1)dx \right| \\
&= \frac{M_2}{2} \left| \int_{x_0}^{x_1} (x-x_0)[x-(x_0+h)]dx \right| \\
&= \frac{M_2}{2} \left| \int_{x_0}^{x_1} (x-x_0)[(x-x_0)-h)]dx \right| \\
&= \frac{M_2}{2} \left| \frac{(x-x_0)^3}{3} - h\frac{(x-x_0)^2}{2} \right|_{x_0}^{x_1} \\
&= \frac{M_2}{2} \left| \frac{(x_1-x_0)^3}{3} - h\frac{(x_1-x_0)^2}{2} \right| \\
&= \frac{M_2}{2} \left[\frac{h^3}{3} - \frac{h^3}{2} \right] \\
&= \frac{M_2}{2} \left| \frac{-h^3}{6} \right|
\end{aligned} \tag{282}
$$

$$= \frac{M_2 h^3}{12}$$

Note that, whereas the simple rectangle rules have $O(h^2)$ error properties, the simple trapezoid rule has $O(h^3)$ properties. What do you expect to happen for the simple midpoint rule? Like the rectangle rules, constant interpolation is used. But errors tend to cancel because the height of each rectangle is determined by the value of the function at the midpoint of the subinterval.

9.3.3 Midpoint Rule Error

Just when you have been lulled into a false sense of security that the same approach can be used to derive the error of any quadrature rule, we regret to inform you that it fails for something as simple-minded as midpoint rule. Why is that?

For mipoint rule, which, like the rectangle rules, uses constant interpolation, there is once again, a single node m. The interpolation error formula is therefore

$$e_0(x) = f'(\xi)(x-m) \tag{283}$$

The problem lies in integrating the formula above between x_0 and x_1. Note that $\psi_1(x) = x - m$ changes sign on the interval and thus the IMVT2 is not applicable. We must finesse the error analysis.

To that end, consider a Taylor expansion of $f(x)$ about the midpoint m.

$$f(x) = f(m) + f'(m)(x-m) + \frac{f''(\xi)}{2}(x-m)^2 \tag{284}$$

If we approximate $f(x)$ by $P_0(x) = f(m)$ on $[x_0, x_1]$, then the interpolation error is

$$f(x) - P_0(x) = f(x) - f(m) = f'(m)(x-m) + \frac{f''\xi)}{2}(x-m)^2 \tag{285}$$

and the integration error on the single subinterval is the integral of the interpolation error, that is

$$|E_M(h)| = \left| \int_{x_0}^{x_1} \left[f'(m)(x-m) + \frac{f''(\xi)}{2}(x-m)^2 \right] dx \right| \tag{286}$$

The first term in the integral above vanishes. Why is that? We are left with

$$\begin{aligned} |E_{SM}(h)| &= \left| \int_{x_0}^{x_1} \frac{f''(\xi)}{2}(x-m)^2 dx \right| \\ &= \left| \frac{f''(c)}{2} \int_{x_0}^{x_1} (x-m)^2 dx \right| \\ &\leq \frac{M_2}{2} \left| \int_{x_0}^{x_1} (x-m)^2 dx \right| \end{aligned} \tag{287}$$

$$
\begin{aligned}
&= \frac{M_2}{2}\left|\frac{(x-m)^3}{3}\right|_{x_0}^{x_1} \\
&= \frac{M_2}{6}\left|[(x_1-m)^3-(x_0-m)^3]\right| \\
&= \frac{M_2}{6}\left|\left[\left(\frac{h}{2}\right)^3-\left(-\frac{h}{2}\right)^3\right]\right| \\
&= \frac{M_2 h^3}{24}
\end{aligned}
$$

Finally, by a similar, but more algebraically involved approach, one can derive the following error bound for simple Simpson's quadrature:

$$|E_{SS}(h)| \leq \frac{M_4 h^5}{90} \tag{288}$$

For completeness, we summarize the error rules for simple quadrature below.

$$
\begin{aligned}
|E_{SL}(h)| &\leq \frac{M_1 h^2}{2} \\
|E_{SR}(h)| &\leq \frac{M_1 h^2}{2} \\
|E_{ST}(h)| &\leq \frac{M_2 h^3}{12} \\
|E_{SM}(h)| &\leq \frac{M_2 h^3}{24} \\
|E_{SS}(h)| &\leq \frac{M_4 h^5}{90}
\end{aligned} \tag{289}
$$

9.3.4 Error Rules for Composite Quadrature

No one performs quadrature using a single subinterval (with the notable historical exception that the astronomer/mathematician Johannes Kepler devised the equivalent of simple Simpson's rule to improve greatly on the estimate of the volume of a wine cask). To keep the error within bounds, the interval width h must be small, which implies that many subintervals are generally required. What then is the relationship between the error on a single subinteral and the error when many subintervals are used? We illustrate using composite Simpson's quadrature.

First, recall the *triangle inequality*: For any two real numbers a and b, $|a+b| \leq |a|+|b|$.

For composite Simpson's rule, there are $n/2$ subintervals, each of width $2h$ and spanning two nodes. Let's adopt the nomenclature, for example, that $E_{SS}^{2-4}(h)$ represents the error of simple Simpson quadrature on the subinterval between nodes x_2 and x_4. Thus,

$$E_S(h) = E_{SS}^{0-2}(h) + E_{SS}^{2-4}(h) + \ldots + E_{SS}^{(n-2)-n}(h) \tag{290}$$

in which case, by the triangle inequality

$$|E_S(h)| \leq \frac{M_4^{0-2}h^5}{90} + \frac{M_4^{2-4}h^5}{90} + ... + \frac{M_4^{(n-2)-n}h^5}{90} \tag{291}$$

Where M_4^{2-4}, for example, bounds the absolute value of the fourth derivative of f on the subinterval $[x_2, x_4]$. Noting that the bound on each subinterval is itself bounded by M_4, the bound on $|f^{(4)}(x)|$ on the entire interval $[a,b]$, we have

$$\begin{aligned} |E_S(h)| &\leq \frac{M_4^{0-2}h^5}{90} + \frac{M_4^{2-4}h^5}{90} + ... + \frac{M_4^{(n-2)-n}h^5}{90} \\ &\leq \frac{M_4h^5}{90} + \frac{M_4h^5}{90} + ... + \frac{M_4h^5}{90} \\ &= \frac{M_4h^5}{90}[1+1+...+1] \\ &= \frac{M_4h^5}{90}\left(\frac{n}{2}\right) \\ &= \frac{(b-a)}{180}M_4h^4 \end{aligned} \tag{292}$$

What happened to the power of h in the last step above? Note that $nh = b-a$. Hence, the power of h diminished by one. In general, and for the same reason, the rules for composite quadrature are of one degree less in h than the corresponding rules for simple quadrature. We could derive the error rules for composite rectangle, midpoint, and trapezoid rules, but it turns out that composite Simpson's rule is the only anomolous rule. Care is needed because there are $n/2$ subintervals each of width $2h$ rather than n subintervals each of width h. For every other composite rule, we simply multiply the simple rule by n and replace nh by $b-a$. Without further ado, we summarize the error rules for composite quadrature below.

$$\begin{aligned} |E_{SL}(h)| &\leq \frac{(b-a)}{2}M_1h \\ |E_{SR}(h)| &\leq \frac{(b-a)}{2}M_1h \\ |E_{ST}(h)| &\leq \frac{(b-a)}{12}M_2h^2 \\ |E_{SM}(h)| &\leq \frac{(b-a)}{24}M_2h^2 \\ |E_{SS}(h)| &\leq \frac{(b-a)}{180}M_4h^4 \end{aligned} \tag{293}$$

We end this chapter with some remarks about the significance of the error rules above.

REMARKS:

1. $L(h)$ and $R(h)$ integrate constant functions f exactly, because $f'(x)$ (and hence, M_1) for a constant function is zero.

2. $M(h)$ and $T(h)$ integrate linear functions f exactly (for different reasons), because in each case, $f''(x)$ (and hence, M_2) for a linear function is zero.

3. $S(h)$, on the other hand integrates quadratic and *cubic* functions f exactly.

There is a mathematical surprise in these remarks. Can you spot it? We expect trapezoid rule to integrate linear functions exactly, because it is based on linear interpolation. Simpson's rule is based on quadratic interpolation, and so we would expect it to integrate quadratic functions exactly. But wait! Simpson's rule integrates quadratic and cubic functions exactly, which is unexpected. It happens because of a fortuitous cancellation of error, in the same way that midpoint rule gives better results than one would expect from zero-order interpolation. In sum, Simpson's rule is much better than one would anticipate. The global error diminshes with h as h^4; hence, very good results can be obtained even with moderately small n (moderately large h).

We remind the reader of several things before closing. First, ALG 9.1 covers a multitude of sins. It can be adapted to each and every one of the quadrature rules with appropriate choice of the weight vector $\vec{w}$ and scaling constant c. Second, don't forget that $S(h)$ and $M(2h)$ each require an even number of subintervals. If you try to apply Simpson's rule, for example, to a problem with an odd number of subintervals, all the nice error properties are destroyed.

HW: Make a table of error for $M(2h)$, $T(h)$, and $S(h)$ for the function $f(x) = 1/x$ on $[a,b] = [1,3]$, with $n = 2$, 4, 8, and 16. Verify that the errors diminish as $O(h^2)$, $O(h^2)$, and $O(h^4)$, respectively.

9.4 Exercises

1. Follow the procedure that culminated in $w_1 = h/3$ in Eq. 270 to show that $w_2 = 4h/3$.

2. Approximate $\int_1^3 \frac{1}{x} dx$ by $S(0.25)$ and compare the result to the exact value.

3. Derive Eq. 274 from the composite rules $T(h)$ and $M(2h)$.

4. The following identity holds true for any continuous function $f(x)$ defined on $[0,1000]$:

$$\int_0^1 f(x)\,dx = \int_0^{1000} f(x)\,dx - \int_1^{1000} f(x)\,dx.$$

   One way to approximate the integral on the left-hand side would be to compute approximations via Simpson's Rule to the two integrals on the right-hand side and then subtract. Give two reasons why this is probably a bad algorithm for approximating the integral on the left-hand side.

5. (a) Use Trapezoid Rule with $n = 4$ equal subdivisions to approximate $\int_2^3 \sin(x)dx$. Round all values to four decimal places.

   (b) What is the absolute error in your Trapezoid approximation in part (a).

   (c) What is the best error bound that can be achieved using the error bound associated with the Trapezoid Rule for $\int_2^3 \sin(x)dx$ with $n = 4$ subdivisions?

(d) Using midpoint rule's error bound, how many subdivisions would you need to guarantee that midpoint rule would approximate this integral with an absolute error of less than 10^{-6}?

6. (a) Use Simpson's Rule with $n = 2$ equal subdivisions to approximate $\int_2^3 ln(x)dx$. Round all values to five decimal places.

(b) What is the absolute error in your Simpson's rule approximation in part (a). (Hint: use integration by parts)

(c) What is the best error bound that can be achieved using the error bound associated with the Simpson's Rule for $\int_2^3 ln(x)dx$ with $n = 2$ subdivisions?

(d) Using midpoint rule's error bound, how many subdivisions would you need to guarantee that midpoint rule would approximate this integral with an absolute error of less than 10^{-6}?