

## Class Business

- Assignments due Thursday.
- Time, date, and videotaping of class.

## Materials used:

- Forsythe, et al., Chapter 2
- Gerald and Wheatley, 6th Edition, Chapter 0.1–0.6

## Objectives:

- Computers most often represent the infinite set of real numbers using a finite set discrete numbers.
- Approximation and errors are part of any numerical calculation.
- The choice of algorithm can determine whether a calculation will be reliable or not. Numerical analysis can help to decide between algorithms.

## Representing Numbers on the Computer

Most mathematical analysis are completed on the set of real numbers. Reals have several interesting properties; they are an infinite set- there is no maximum, no minimum, and between any two reals there exists an infinite number of reals.

Computers, however, represent numbers using a finite set of bits. This means that there are both upper and lower limits to the magnitude of the numbers that can be represented, and that the set of numbers that can be represented exactly is also limited.

Most computers use a dual-state representation - each bit can have one of two values: 0 or 1. Bits are grouped into groups of 8, so that 8 bits = one byte. Bytes are often grouped into “words”, where one word is used to represent a number. The size of the word depends on the specifics of the computer being used, a word is often 32-bits.

**Integers:** Computers generally store integers in base-2. The first bit in the word is usually used to indicate whether the integer is positive or negative. A “0” in the most significant bit indicates that the integer is larger than 0, and a leading “1” indicates that the integer is less than 0. For positive integers, the computer stores them as the binary representation of the integer. We are used

to thinking in base-10: for example the number  $37 = 3 \times 10^1 + 7 \times 10^0$ . This same would be represented in base-2 by one byte;  $00100101 = 2^5 + 2^2 + 2^0$ . Integers that are less than zero can be represented a few different ways. The most common scheme for representing signed integers is *two's complement*. To convert a negative integer to its two's complement representation, the absolute value is first converted to binary, then each bit is "flipped", and 1 is added to the result. An 8-bit representation of -37 using two's complement would therefore be 11011011. Twos complement is more often used than other methods for representing integers, because it makes low-level operations, such as addition, more efficient. If a word contains N bits the computer can represent integers between  $\sim \pm 2^{N-1}$ . Values outside of this range can not be represented.

**Floating point notation** Most scientific calculations rely on real numbers. Often the numbers that go into a single calculation may range over several orders of magnitude. Computer scientists developed floating point notation as a way to represent a wide range real numbers with a given number of bits. The general notation for base  $\beta$  machines is

$$x = (d_1/\beta + d_2/\beta^2 + \dots + d_t/\beta^t) \times \beta^e; = f \times \beta^e \quad (1)$$

where  $t$  is the *precision* of the representation,  $e$  is the *exponent*, and the part in parenthesis ( $f$ ) is called the *mantissa* or *fraction*. Each digit,  $d_i$ , in the mantissa satisfies  $0 \leq d_i \leq \beta - 1$ . To represent the real number,  $x$ , within one word of memory that is N-bits long, the computer reserves one bit, uses  $t$ -bits of the word to store  $f = d_1 d_2 d_3 \dots d_t$ , and the remaining bits to store the value of the exponent,  $e$ .

Suppose we want to store 37 in floating point notation, in base-10. That would simply be  $0.37 \times 10^2$ , so we would store 37 in the mantissa, or fraction, of the word, and 2 in the exponent. A lot of floating point representations assume that the fraction,  $f$ , is preceded by a leading "0.". Converting to binary, the value  $37_{10}$  would become  $100101_2$ . This could be represented as a float using  $f=0.100101$  and  $e = 6_{10} = 110_2$ .

Different computers (and different compiler options on individual computers) will use different conventions for the length a word, and for allocating the number of bits for the mantissa and exponents. The IEEE standard for floating point numbers specifies that 32 bits are used for each floating point number; 23 bits are used for the fractional part of the number, 8 bits are used for the exponent, and 1 bit is used as a "hidden" bit (for special cases). By this standard, the biggest number than can be represented is  $1.111\dots \times 2^{127} = 2^{128} - 1 \approx 10^{38}$ . The smallest positive number that can be represented approximately equals the reciprocal of this,  $10^{-38}$ .

**Example of floating point set:** As an example, consider a small set of floating point numbers that use base-2. Suppose we have 4 bits for the fraction or mantissa, and 2 bits for the exponent, and we use two's complement represent the numbers. We can store fractions equal to 0.100, 0.101, 0.110, and 0.111, and

special case 0.000. The representation of the positive fractions within four bits would be 0100, 0101, 0110, 0111; the negative fractions would be 1100, 1011, 1010, and 1001. The exponents can be 00, 01, 11, or 10 (decimal equivalents 0, 1, -1, -2). The total range of our set of numbers is therefore  $\pm 1.75$ , and within that range we can represent 33 numbers. Numbers whose magnitudes are larger than 1.75 will overflow; numbers whose magnitudes are less than  $1/8$  will underflow. Real numbers whose magnitudes fall between two of our set of finite numbers would have to be rounded or truncated in order to be represented. Clearly, we need more than 33 unique numbers in scientific calculations. Values therefore include those shown in table 1. Notice all of the leading digits are a “1” after the decimal point; this is because in our computer the values are *normalized*, that is, the mantissa is chosen so that a “1” is in the leading digit after the decimal point for the positive floats. The negative floats are chosen to be the twos complements of their absolute values. These values can be mapped on the interval  $\pm 1.75$ . The value for 0 (zero) is represented as  $0.00 \times 2^0$ , and is a special case because it is not normalized.

Within this example, we can see that any non-zero real number,  $x$ , where  $|x| < 0.125$  can not be represented. Such a number will fall into the *underflow* region-it will be indistinguishable from zero. Any real number whose magnitude is greater than 1.75 will likewise fall into the *overflow* region. This indicates an important part of floating point notation- there are both maximum and minimum values for the size of floats. Another important thing to notice is that there are gaps between the valid numbers. For example, the number 1.05 can not be represented within this set, but would have to be *rounded* or *truncated* to a valid number. It is also interesting that there are fewer valid numbers between 0 and  $1/8$  than there are between  $1/8$  and  $1/4$ . This is because we used a normalized set of mantissas and thereby lost resolution near zero; but we maintained precision within our set of numbers because they all have 3 digits of precision.

The value of *machine epsilon* ( $\epsilon$ ) is often cited to provide a measure of the precision with which floating point numbers are represented. A few different values of  $\epsilon$  have been defined; most often it is the smallest positive number, which when added to 1 will give a floating point number greater than 1; i.e. find  $\epsilon$  such that  $1 + \epsilon > 1$ . In our example,  $\epsilon = 1/4$ . Sometimes, another value of  $\epsilon$  is defined to be the value that when subtracted from 1 gives a floating point number less than 1; this value would be  $1/8$  in our example. Try estimating the machine  $\epsilon$  present for your computer.

## Accuracy in floating point computations

The above example hints at the roundoff errors present in any floating point computation. These and other errors are itemized below.

- Roundoff: many real numbers will not have an exact representation in floating point notation, and will have to be rounded or chopped or truncated to a value that is within the set represented by the computer. The

error that this produces is called *roundoff* error. In the example above, if we wished to represent 3.25 we could round it to either 3 or 3.5, or truncate it to 3.0. Either way, in this example we would incur a roundoff error of 0.25.

Roundoff error exists no matter how many bits can be used to represent a number. Many real numbers simply can not be represented by a finite set of digits. Imagine that you want to represent a number like  $7/10$ . In base-10, this is simply  $7.0 \times 10^{-1}$ . In binary, however, you would need an infinite number of bits to represent  $(7/10)_{10}$  exactly; because  $7/10_{10} = 0.101100110011001100\dots_2$ . If you only have 11 bits in binary,  $0.7_{10}$  will be represented as  $0.10110011001_2$ , which has the decimal value of 0.69921875. The representation of  $0.7_{10}$  therefore introduced a *rounding error* equal to  $|0.7 - 0.69921875| \approx 7.8 \times 10^{-4}$ , with a *relative error* equal to  $1.1 \times 10^{-3}$ . Try writing the number  $1/10$  in binary floating point representation.

Increasing the digits, or precision, that are available with which to represent floating point numbers will reduce rounding error.

- Underflow and overflow: Sometimes the result of a calculation is a number that lies outside of the range included in the floating point representation. These are termed *underflow* if the real solution would be closer to zero than the smallest valid floating point number, and *overflow* if the real solution would have a magnitude larger than the range represented by the floating point set. Different compilers (and compiler options) will handle these in different ways- some will note “overflow” or “underflow” upon completion, some will simply map the real solution onto  $\pm MAXFLOAT$  for an overflow, and 0 for an underflow; and some will bomb.
- Truncation: Many numerical algorithms rely on an iterative approach that seeks to converge on the solution, or on a truncated version of an infinite sum (Taylor expansion, for example). The difference between the exact solution and the computed solution is referred to as *truncation error*, because some part of the exact solution was truncated to get the computed solution. Truncation error can be reduced by increasing the number of iterations or terms included in the solution. The *order* of a solution usually indicates the magnitude of the truncation error, and is represented in the form  $O(h(x))$  (more on this later in the examples).
- Error in original data: This is especially relevant to numerical methods as applied to marine sciences. Most of our computations are aimed at predicting or explaining observable phenomena in the coastal realm. Our computations therefore either rely on data measured in the coastal ocean, or they will be compared against such data. Most measurement techniques have errors in themselves- either with respect to the absolute values which were measured, interpretations of data which are proxies for the value needed, or uncertainties with respect to the timing or location of the

measurement. It is important to remember the limitations of your data sets when developing a numerical method. You might wish to test the sensitivity of your calculation to the uncertainties present in the data.

- **Blunders:** computer programmers make mistakes. Debugging and quality-control are an important (and often under-represented) part of the process of numerical computations. Always look at your results and make sure that they make sense. When possible, run the numerical calculation against a problem that has a known or analytical solution for comparison.
- **Propagated errors:** Whether or not *local* errors are propagated through an algorithm, and whether they grow or shrink with each step determines the stability of the method. Relatively large local errors may be acceptable if they do not corrupt the final solution. Solutions where the error continues to grow with each step or iteration are termed *unstable*, whereas solutions where the error term shrinks are called *stable*. Stable methods are obviously preferred. Numerical analysis is one method for determining whether an algorithm is stable or unstable.

## Examples

1. **Comparative methods for storing floating-point numbers:** Table that compares IEEE, VAX, and IBM representations of floating point numbers.
2. **Example of roundoff error:** estimate the function  $e^x = \exp(x)$  using the sum

$$\exp(x) = e^x = 1 + x + x^2/2! + x^3/3! \dots \quad (2)$$

Use a base-10 floating point representation, with  $t=5$  digits of precision. Estimate  $e^{-5.5}$ .

This example illustrates *catastrophic cancellation*; a loss in precision that occurs when the intermediate steps in an algorithm relies on subtracting numbers that are much larger than the final solution.

3. **Try again to estimate  $\exp(x)$ :** The problems of the above solution can be avoided if we realize that the error originated from the oscillating sign of large intermediate terms. These can be avoided by noting that  $e^{-x} = 1/e^x$ , and using equation 2 for  $x > 0$ ; and for  $x < 0$  using

$$e^{-x} = \exp(-x) = 1/e^x = 1/[1 + x + x^2/2! + x^3/3! \dots] \dots \quad (3)$$

4. **Adding numbers of different magnitudes:** Suppose you have 4 digits of precision, and you need to add  $0.2501 + 0.1000 \times 10^{-4}$ . The answer, in floating point computations, will be 0.2501; the smaller number was too small to impact the solution, and was effectively lost.

5. **Dividing by a small number:** Suppose you have a 4-digit decimal representation of a floating point number; and intermediate results are truncated to 4 digits. Compute  $A - B/C$ , where  $A = 0.1120 \times 10^9$ ,  $B = 0.1000 \times 10^6$ , and  $C = 0.9000 \times 10^{-3}$ . Using 4 digits:  $B/C$  is approximated to be  $B/C = 0.1111 \times 10^9$ , and  $A - B/C = 0.9000 \times 10^6$ .

Suppose  $C$  were just a little different, say  $C = 0.9001 \times 10^{-3}$ ; now  $A - B/C = A - (0.1110 \times 10^9)$ , or  $A - B/C = 0.1000^7$ . So that a 1/9000 change in  $C$  led to a 10% change in the final solution.

This calculation could be done in a more stable manner using  $\frac{AC-B}{C}$ , which yields the exact answer to four decimal places, and is insensitive to small changes in  $C$ .

6. **Error propagation:** Example 2.4 from Forsythe, et al. shows an example where the original roundoff error is magnified until the solution is unusable. Suppose you want to solve

$$E_n = \int_0^1 x^n e^{x-1} dx; n = 1, 2, \dots \quad (4)$$

Forsythe shows that this can be represented as

$$E_n = 1 - nE_{n-1}, n = 2, 3, \dots; \quad (5)$$

where  $E_1 = 1/e$ . The first step in solving  $E_n$  is to estimate  $1/e$ . This involves some rounding error - no matter how many digits you have at your disposal. You could write that  $E_1 = 1/e + \delta$ , and use  $E_1$  to represent the approximation to  $1/e$ , and  $\delta$  to represent the rounding error. Subsequent steps multiply that rounding error. For example,  $E_2 = 1 - 2(E_1) = 1 - 2(1/e + \delta)$ ;  $E_3 = 1 - 3E_2 = 1 - 3 \times 2(1/e + \delta)$ . As  $n$  increases, the original rounding error is multiplied by  $n!$ , the error grows unbounded, and the method is unstable.

Forsythe, et al. suggest an alternative that will provide a stable method.

$$E_{n-1} = [1 - E_n] / n; n = \dots, 3, 2. \quad (6)$$

Using this method to solve for  $E_{n-1}$ , you would choose a starting point  $E_m$ , where  $m \gg n - 1$ . This value will also have some rounding error, such that the exact value is represented as the approximation plus  $\delta_m$ . Now,  $E_{m-1} = (1 - (E_m - \delta)) / (m - 1)$ ; and so on. Now, the original truncation error  $\delta$  is divided, so that the error is decreased with each successive step. This leads to a stable method.

7. **Solving roots using the quadratic equation:** The quadratic  $ax^2 + bx + c = 0$  is solved using

$$x = \left[ -b \pm \sqrt{(b^2 - 4ac)} \right] / 2a. \quad (7)$$

Suppose you want to solve the following cases using base-10 floating point numbers.

- Case 1: suppose  $b$  is large compared to  $a$  and  $c$ . The value in the square-root will be poorly resolved so that the numerator is  $\approx b \pm (\sim b)$ . Two sources of the error: one is adding a small number to a large number; and the second is catastrophic cancellation. The method can be improved by using the best estimate of the first root;  $x_1 = 2b/ac$ ; and then solving  $x_2 = c/(ax_1)$ .
- Case 3: Floating point overflow. Like Case 1, except  $a$ ,  $b$ , and  $c$  are all multiplied by a large number (Forsythe uses  $10^{30}$ ). The roots are the same as in Case 1; but using the second method described above will give floating-point overflows. The method would work, and a correct solution found, if all numbers ( $a$ ,  $b$ , and  $c$ ) were scaled so that the smallest had an exponent of 0.

## Error analysis

- Inverse error approach: consider how little change in the data would be necessary to cause the computation to be the exact solution for the changed problem. Possible to evaluate complex and multi- dimensional problems.
- Direct error approach: evaluate the difference between the computed and the exact solution. Classical approach.
- *Absolute error* is defined to be the magnitude of the difference between the exact solution and the computed solution. *Relative error* is defined to be the absolute error divided by the exact solution.
- The *order* of an algorithm indicates the size of the error term. For example, numerical analysis of one of the preceding examples indicated that the absolute error of the method increased with  $n!$ . That method therefore has an error term of order  $n!$ , or in mathematical shorthand, the error term  $\sim O(n!)$ .

Table 1: Set of numbers represented using 4 bits for the mantissa, 2 bits for the exponent, base-2, and two's complement.

$-0.111 \times 2^1$	$= -1.11 \times 2^0$	$= -1 - 1/2 - 1/4$	$= -1 \ 3/4$
$-0.110 \times 2^1$	$= -1.10 \times 2^0$	$= -1 - 1/2$	$= -1 \ 1/2$
$-0.101 \times 2^1$	$= -1.01 \times 2^0$	$= -1 - 1/4$	$= -1 \ 1/4$
$-0.100 \times 2^1$	$= -1.00 \times 2^0$	$= -1$	$= -1$
$-0.111 \times 2^0$	$= -1.11 \times 2^{-1}$	$= -1/2 - 1/4 - 1/8$	$= -7/8$
$-0.110 \times 2^0$	$= -1.10 \times 2^{-1}$	$= -1/2 - 1/4$	$= -3/4$
$-0.101 \times 2^0$	$= -1.01 \times 2^{-1}$	$= -1/2 - 1/8$	$= -5/8$
$-0.100 \times 2^0$	$= -1.00 \times 2^{-1}$	$= -1/2$	$= -1/2$
$-0.111 \times 2^{-1}$	$= -1.11 \times 2^{-2}$	$= -1/4 - 1/8 - 1/16$	$= -7/16$
$-0.110 \times 2^{-1}$	$= -1.10 \times 2^{-2}$	$= -1/4 - 1/8$	$= -3/8$
$-0.101 \times 2^{-1}$	$= -1.01 \times 2^{-2}$	$= -1/4 - 1/16$	$= -5/16$
$-0.100 \times 2^{-1}$	$= -1.00 \times 2^{-2}$	$= -1/4$	$= -1/4$
$-0.111 \times 2^{-2}$	$= -1.11 \times 2^{-3}$	$= -1/8 - 1/16 - 1/32$	$= -7/32$
$-0.110 \times 2^{-2}$	$= -1.10 \times 2^{-3}$	$= -1/8 - 1/16$	$= -3/16$
$-0.101 \times 2^{-2}$	$= -1.01 \times 2^{-3}$	$= -1/8 - 1/32$	$= -5/32$
$-0.100 \times 2^{-2}$	$= -1.00 \times 2^{-3}$	$= -1/8$	$= -1/8$
$0.000 \times 2^0$	$= 0 \times 2^0$	$= 0$	$= 0$
$0.100 \times 2^{-2}$	$= 1.00 \times 2^{-3}$	$= 1/8$	$= 1/8$
$0.101 \times 2^{-2}$	$= 1.01 \times 2^{-3}$	$= 1/8 + 1/32$	$= 5/32$
$0.110 \times 2^{-2}$	$= 1.10 \times 2^{-3}$	$= 1/8 + 1/16$	$= 3/16$
$0.111 \times 2^{-2}$	$= 1.11 \times 2^{-3}$	$= 1/8 + 1/16 + 1/32$	$= 7/32$
$0.100 \times 2^{-1}$	$= 1.00 \times 2^{-2}$	$= 1/4$	$= 1/4$
$0.101 \times 2^{-1}$	$= 1.01 \times 2^{-2}$	$= 1/4 + 1/16$	$= 5/16$
$0.110 \times 2^{-1}$	$= 1.10 \times 2^{-2}$	$= 1/4 + 1/8$	$= 3/8$
$0.111 \times 2^{-1}$	$= 1.11 \times 2^{-2}$	$= 1/4 + 1/8 + 1/16$	$= 7/16$
$0.100 \times 2^0$	$= 1.00 \times 2^{-1}$	$= 1/2$	$= 1/2$
$0.101 \times 2^0$	$= 1.01 \times 2^{-1}$	$= 1/2 + 1/8$	$= 5/8$
$0.110 \times 2^0$	$= 1.10 \times 2^{-1}$	$= 1/2 + 1/4$	$= 3/4$
$0.111 \times 2^0$	$= 1.11 \times 2^{-1}$	$= 1/2 + 1/4 + 1/8$	$= 7/8$
$0.100 \times 2^1$	$= 1.00 \times 2^0$	$= 1$	$= 1$
$0.101 \times 2^1$	$= 1.01 \times 2^0$	$= 1 + 1/4$	$= 1 \ 1/4$
$0.110 \times 2^1$	$= 1.10 \times 2^0$	$= 1 + 1/2$	$= 1 \ 1/2$
$0.111 \times 2^1$	$= 1.11 \times 2^0$	$= 1 + 1/2 + 1/4$	$= 1 \ 3/4$