Jim Lambers
MAT 460/560
Fall Semester 2009-10
Lecture 4 Notes

These notes correspond to Sections 1.1 and 1.2 in the text.

Review of Calculus, cont'd

Taylor's Theorem, cont'd

We conclude our discussion of Taylor's Theorem with some examples that illustrate how the nth-degree Taylor polynomial $P_n(x)$ and the remainder $R_n(x)$ can be computed for a given function $f(x)$.

Example If we set $n = 1$ in Taylor's Theorem, then we have

$$f(x) = P_1(x) + R_1(x)$$

where

$$P_1(x) = f(x_0) + f'(x_0)(x - x_0).$$

This polynomial is a linear function that describes the tangent line to the graph of f at the point $(x_0, f(x_0))$.

If we set $n = 0$ in the theorem, then we obtain

$$f(x) = P_0(x) + R_0(x),$$

where

$$P_0(x) = f(x_0)$$

and

$$R_0(x) = f'(\xi(x))(x - x_0),$$

where $\xi(x)$ lies between x_0 and x. If we use the integral form of the remainder,

$$R_n(x) = \int_{x_0}^{x} \frac{f^{(n+1)}(s)}{n!}(x - s)^n \, ds,$$

then we have

$$f(x) = f(x_0) + \int_{x_0}^{x} f'(s) \, ds,$$

which is equivalent to the Total Change Theorem and part of the Fundamental Theorem of Calculus. Using the Mean Value Theorem for integrals, we can see how the first form of the remainder can be obtained from the integral form. □

Example Let $f(x) = \sin x$. Then

$$f(x) = P_3(x) + R_3(x),$$

where

$$P_3(x) = x - \frac{x^3}{3!} = x - \frac{x^3}{6},$$

and

$$R_3(x) = \frac{1}{4!}x^4 \sin \xi(x) = \frac{1}{24}x^4 \sin \xi(x),$$

where $\xi(x)$ is between 0 and x. The polynomial $P_3(x)$ is the 3rd Maclaurin polynomial of $\sin x$, or the 3rd Taylor polynomial with center $x_0 = 0$.

If $x \in [-1, 1]$, then

$$|R_n(x)| = \left|\frac{1}{24}x^4 \sin \xi(x)\right| = \left|\frac{1}{24}\right| |x^4||\sin \xi(x)| \leq \frac{1}{24},$$

since $|\sin x| \leq 1$ for all x. This bound on $|R_n(x)|$ serves as an upper bound for the error in the approximation of $\sin x$ by $P_3(x)$ for $x \in [-1, 1]$. □

Example Let $f(x) = e^x$. Then

$$f(x) = P_2(x) + R_2(x),$$

where

$$P_2(x) = 1 + x + \frac{x^2}{2},$$

and

$$R_2(x) = \frac{x^3}{6}e^{\xi(x)},$$

where $\xi(x)$ is between 0 and x. The polynomial $P_2(x)$ is the 2nd Maclaurin polynomial of e^x, or the 2nd Taylor polynomial with center $x_0 = 0$.

If $x > 0$, then $R_2(x)$ can become quite large, whereas its magnitude is much smaller if $x < 0$. Therefore, one method of computing e^x using a Maclaurin polynomial is to use the nth Maclaurin polynomial $P_n(x)$ of e^x when $x < 0$, where n is chosen sufficiently large so that $R_n(x)$ is small for the given value of x. If $x > 0$, then we instead compute e^{-x} using the nth Maclaurin polynomial for e^{-x}, which is given by

$$P_n(x) = 1 - x + \frac{x^2}{2} - \frac{x^3}{6} + \cdots + \frac{(-1)^n x^n}{n!},$$

and then obtaining an approximation to e^x by taking the reciprocal of our computed value of e^{-x}. □

Example Let $f(x) = x^2$. Then, for any real number x_0,

$$f(x) = P_1(x) + R_1(x),$$

where

$$P_1(x) = x_0^2 + 2x_0(x - x_0) = 2x_0x - x_0^2,$$

and

$$R_1(x) = (x - x_0)^2.$$

Note that the remainder does not include a "mystery point" $\xi(x)$ since the 2nd derivative of x^2 is only a constant. The linear function $P_1(x)$ describes the tangent line to the graph of $f(x)$ at the point $(x_0, f(x_0))$. If $x_0 = 1$, then we have

$$P_1(x) = 2x - 1,$$

and

$$R_1(x) = (x - 1)^2.$$

We can see that near $x = 1$, $P_1(x)$ is a reasonable approximation to x^2, since the error in this approximation, given by $R_1(x)$, would be small in this case. □

Roundoff Errors and Computer Arithmetic

In computing the solution to any mathematical problem, there are many sources of error that can impair the accuracy of the computed solution. The study of these sources of error is called *error analysis*, which will be discussed in the next lecture. In this lecture, we will focus on one type of error that occurs in all computation, whether performed by hand or on a computer: *roundoff error*.

This error is due to the fact that in computation, real numbers can only be represented using a finite number of digits. In general, it is not possible to represent real numbers exactly with this limitation, and therefore they must be approximated by real numbers that *can* be represented using a fixed number of digits, which is called the *precision*. Furthermore, as we shall see, arithmetic operations applied to numbers that can be represented exactly using a given precision do not necessarily produce a result that can be represented using the same precision. It follows that if a fixed precision is used, then *every* arithmetic operation introduces error into a computation.

Given that scientific computations can have several sources of error, one would think that it would be foolish to compound the problem by performing arithmetic using fixed precision. As we shall see in this lecture, however, using a fixed precision is far more practical than other options, and, as long as computations are performed carefully, sufficient accuracy can still be achieved.

Floating-Point Numbers

We now describe a typical system for representing real numbers on a computer.

Definition *(Floating-point Number System) Given integers $\beta > 1$, $p \geq 1$, L, and $U \geq L$, a* **floating-point number system** $\mathbb{F}$ *is defined to be the set of all real numbers of the form*

$$x = \pm m\beta^E.$$

The number m is the **mantissa** *of x, and has the form*

$$m = \left(\sum_{j=0}^{p-1} d_j \beta^{-j} \right),$$

where each digit d_j, $j = 0, \ldots, p-1$ is an integer satisfying $0 \leq d_i \leq \beta - 1$. The number E is called the **exponent** *or the* **characteristic** *of x, and it is an integer satisfying $L \leq E \leq U$. The integer p is called the* **precision** *of $\mathbb{F}$, and β is called the* **base** *of $\mathbb{F}$.*

The term "floating-point" comes from the fact that as a number $x \in \mathbb{F}$ is multiplied by or divided by a power of β, the mantissa does not change, only the exponent. As a result, the decimal point shifts, or "floats," to account for the changing exponent.

Nearly all computers use a binary floating-point system, in which $\beta = 2$. In fact, most computers conform to the *IEEE standard* for floating-point arithmetic. The standard specifies, among other things, how floating-point numbers are to be represented in memory. Two representations are given, one for *single-precision* and one for *double-precision*. Under the standard, single-precision floating-point numbers occupy 4 bytes in memory, with 23 bits used for the mantissa, 8 for the exponent, and one for the sign. IEEE double-precision floating-point numbers occupy eight bytes in memory, with 52 bits used for the mantissa, 11 for the exponent, and one for the sign.

Example Let $x = -117$. Then, in a floating-point number system with base $\beta = 10$, x is represented as

$$x = -(1.17)10^2,$$

where 1.17 is the mantissa and 2 is the exponent. If the base $\beta = 2$, then we have

$$x = -(1.110101)2^6,$$

where 1.110101 is the mantissa and 6 is the exponent. The mantissa should be interpreted as a string of binary digits, rather than decimal digits; that is,

$$\begin{aligned} 1.110101 &= 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} + 0 \cdot 2^{-5} + 1 \cdot 2^{-6} \\ &= 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{16} + \frac{1}{64} \end{aligned}$$

$$= \frac{117}{64}$$
$$= \frac{117}{2^6}.$$

□

Properties of Floating-Point Systems

A floating-point system $\mathbb{F}$ can only represent a finite subset of the real numbers. As such, it is important to know how large in magnitude a number can be and still be represented, at least approximately, by a number in $\mathbb{F}$. Similarly, it is important to know how small in magnitude a number can be and still be represented by a *nonzero* number in $\mathbb{F}$; if its magnitude is too small, then it is most accurately represented by zero.

Definition *(Underflow, Overflow) Let $\mathbb{F}$ be a floating-point number system. The smallest positive number in $\mathbb{F}$ is called the* **underflow level***, and it has the value*

$$UFL = m_{\min} \beta^L,$$

where L is the smallest valid exponent and $m_{\min}$ is the smallest mantissa. The largest positive number in $\mathbb{F}$ is called the overflow level*, and it has the value*

$$OFL = \beta^{U+1}(1 - \beta^{-p}).$$

The value of $m_{\min}$ depends on whether floating-point numbers are *normalized* in $\mathbb{F}$; this point will be discussed later. The overflow level is the value obtained by setting each digit in the mantissa to $\beta - 1$ and using the largest possible value, U, for the exponent.

It is important to note that the real numbers that can be represented in $\mathbb{F}$ are not equally spaced along the real line. Numbers having the same exponent are equally spaced, and the spacing between numbers in $\mathbb{F}$ decreases as their magnitude decreases.

Normalization

It is common to *normalize* floating-point numbers by specifying that the leading digit d_0 of the mantissa be nonzero. In a binary system, with $\beta = 2$, this implies that the leading digit is equal to 1, and therefore need not be stored. Therefore, in the IEEE floating-point standard, $p = 24$ for single precision, and $p = 53$ for double precision, even though only 23 and 52 bits, respectively, are used to store mantissas. In addition to the benefit of gaining one additional bit of precision, normalization also ensures that each floating-point number has a unique representation.

One drawback of normalization is that fewer numbers near zero can be represented exactly than if normalization is not used. Therefore, the IEEE standard provides for a practice called *gradual*

underflow, in which the leading digit of the mantissa is allowed to be zero when the exponent is equal to L, thus allowing smaller values of the mantissa. In such a system, the number UFL is equal to β^{L-p+1}, whereas in a normalized system, UFL $= \beta^L$.

Rounding

A number that can be represented exactly in a floating-point system is called a *machine number*. Since only finitely many real numbers are machine numbers, it is necessary to determine how non-machine numbers are to be approximated by machine numbers. The process of choosing a machine number to approximate a non-machine number is called *rounding*, and the error introduced by such an approximation is called *roundoff error*. Given a real number x, the machine number obtained by rounding x is denoted by $\text{fl}(x)$.

In most floating-point systems, rounding is achieved by one of two strategies:

- *chopping*, or *rounding to zero*, is the simplest strategy, in which the base-β expansion of a number is truncated after the first p digits. As a result, $\text{fl}(x)$ is the unique machine number between 0 and x that is nearest to x.

- *rounding to nearest*, or *rounding to even* sets $\text{fl}(x)$ to be the machine number that is closest to x in absolute value; if two numbers satisfy this property, then $\text{fl}(x)$ is set to be the choice whose last digit is even.

Example Suppose we are using a floating-point system with $\beta = 10$ (decimal), with $p = 4$ significant digits. Then, if we use chopping, or rounding to zero, we have $\text{fl}(2/3) = 0.6666$, whereas if we use rounding to nearest, then we have $\text{fl}(2/3) = 0.6667$. If $p = 2$ and we use rounding to nearest, then $\text{fl}(89.5) = 90$, because we choose to round so that the last digit is even when the original number is equally distant from two machine numbers. This is why the strategy of rounding to nearest is sometimes called *rounding to even*. □