Indian Statistical Institute, Kolkata
Numerical Analysis (BStat–I)

Instructor: Sourav Sen Gupta
Scribe: Spandan Bhattacharya, Kaustav Chatterjee
Date of Lecture: 14 January 2016

LECTURE
1

Numerics and Error Analysis

In the first lecture of Numerical Analysis (Fall 2016), the different possible representations of numbers in an algorithm are considered. The merits and demerits of each method are considered. By studying examples, we approach the central idea of numerical analysis, that is, mathematically correct is NOT equivalent to numerically correct. We formally define abstract terminology like error, backward error, forward error etc and look into the different representations of the error of an element, with the help of examples. To make regular addition numerically correct, we introduce the Kahan summation algorithm. Finally, to wind up the lecture, we took some practical examples like vector norms, summation etc.

1.1 Storing Numbers

In numerical analysis, we traditionally work with floats and doubles instead of ints and longs. This might introduce a runtime error in our code.

Example 1.1. *double x= 1.0*

double y= x/3.0

*if (x==y*3.0) cout << "They are equal!";*

else cout << "They are not equal!";

Coming as a shock to intuition, the code will print "They are not equal!". This is because the value of y will be something like 0.33333... This value can be represented in binary $2^{-2} + 2^{-4} + 2^{-6}$ and so on... (we will understand binary representation later in this text)

The number of individual elements in y, as we can see, approaches infinity, but we have finite storage space available in our memory, thus, it takes only the first few elements, and the rest of the number is discarded, introducing an error in the value of y, as it becomes numerically less than 1/3. This gives rise to the error in our algorithm.

Thus, in our example, although x and $y*3.0$ are close enough to be considered as identical for all practical purposes, they certainly are not numerically equal.

We essentially approach the central idea of the subject, which is

The operator == is rarely used to compare two variables, instead, we must allow some 'tolerance'

Now that we have seen the problems with representation of numbers, let us answer the deeper question- how are numbers represented inside a machine?

1.1.1 Fixed Point Representation

Like the name suggests, in this system, we store numbers by using a fixed decimal point. For instance, let us consider the number 271.375 in this system. For the representation of any arbitrary number n, we will take a series of values from 2^k to 2^{-l} where $2^k <= n < 2^{k+1}$ and $2^l <= frac(n) < 2^{l-1}$ (where $frac$ denotes fractional part) and for every p in (l, k), a value of either 0 or 1 will be assigned.

Given below is the representation of 271.375

1	0	0	0	0	1	1	1	1	0	1	1
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}

Thus, our number can be represented in $k + l + 1$ binary digits, out of which l lie after the decimal point.

The chief advantage of this type of method is that several arithmetic operations can be carried out in the same way as we do for regular integers.

For instance, consider the addition of two numbers x and y represented in fixed point representation.

Example 1.2. $x + y = (x * 2^k + y * 2^k)2^{-k}$

However, a big drawback of fixed-point representation is precision, mainly because, after performing some operations, the end result might require more bits to represent than the operands

For instance, let us specify two elements in fixed point representation.

Example 1.3. $0.03_3 * 0.05_3 = 0.0015_3$ *=0.00*

Representation of numbers seems strangely akin to Heisenberg's uncertainity principle, since higher speed results in compromising on accuracy and vice versa. This system of representation is common in graphics processing unit (GPU) since less precision is sufficient for many graphical representations.

1.1.2 Floating Point Representation

Floating-point systems are dened using three parameters:

1)The base or radix b belongs to the set of natural numbers. For scientic notation (explained below), the base is b = 10; for binary systems the base is b = 2.

2)The precision p, where p belongs to n, represents the number of digits used to store the signicand.

3)The range of exponents [L,U] representing the allowable values for e.

Like mentioned above, floating point numbers are especially useful when we look upon things from a scientific point of view.

For instance, the value of Rydberg constant is $1.0974 * 10^7 m^{-1}$ and the permittivity of vacuum is $8.854 * 10^{-12} Fm^{-1}$

Now, it's obviously easier to write a scientific notation as an 'important' element times some power of 10, rather than writing it in its normal form, hence, floating point representation is preferred here.

However, for scientific constants, the base is taken equal to 10, for simplifying our calculations while the base is taken to be 2 in the regular representation of the floating point numbers.

Looking back at the description of floating point number given in the previous page, we realize that its representation will look like

$(d_0+d_1 b^{-1}+d_2 b^{-2} ... d_{p-1} b^{1-p}) * b^e$

The first part is called the mantissa and the second part is called the exponent. Also, we can fix any sign in front of the number, + for positive and - for negative.

We intuitively observe that the value for each d_k should belong to $[0, b-1]$, thus, to keep our binary system intact, the value of b must be 2.

Though this system of representing numbers gives a more accurate picture than fixed point, the biggest drawback of this system is that their spacing is uneven. In a fixed point representation system, the space between is arithmetically proportional while in this case, the spacing is geometrically proportional. For instance, the spacing between b^x and b^{x+p} is same as the spacing between b^y and b^{y+p}

1.1.3 Other Exotic Representations

Let us consider the following addition of rational numbers.

Example 1.4. $a/b + c/d = (ad + bc)/bd$

What's special about this is that we are back to a perfect equality without any requirement whatsoever for tolerance values.

While this might seem extremely cool in the first glance, the flipside is that we have to decide our operations beforehand and that we are stuck in the set of rational numbers.

1.2 Introducing Error

With the exception of the representation we saw in subsection 1.1.3, almost every computerized representation of real numbers with fractional parts is forced to have some error.

The most fundamental feature of numerical computing is the inevitable presence of error. The result of any interesting computation (and of many unin- teresting ones) will be only approximate, and our general quest is to ensure that the resulting error be tolerably small.

1.2.1 Types Of Error

Let us look at the different types of error we encounter

1) Truncation error- Like we saw in section 1.1.1, sometimes, the tolerance of the ALU of the machine is simply not enough to store the entire value. For instance, let us consider the binary value of 1/3

$1/3 = 0.010101....$

Obviously, it will be truncated after taking a finite number of values.

2) Discretization error- Error comes from our computerized adaptations of calculus, physics, and other aspects of continuous mathematics. For instance, a numerical system might attempt to approximate the derivative of a function $f(t)$ using divided dierences

$f'(t) \approx (f(t+\epsilon) - f(t))/\epsilon$

lim $\epsilon - 0$. This approximation is a legitimate and useful one, , but since we must use a finite $e > 0$ rather than taking a limit, the resulting value for $f'(t)$ is only accurate to some number of digits.

3) Modelling error- Modelling error arises from having incomplete or inaccurate descriptions of the problems we wish to solve. Constants such as the speed of light or acceleration due to gravity might be provided to the system with a limited degree of accuracy.

4) Input error- This can come from user-generated approximations of parameters of a given system (and from typos!). In this case, a highly accurate simulation might be a waste of computational time, since the inputs to the simulation were so rough.

1.2.2 Classification Of Error

Let us consider two numbers

$1 + -0.01$

$10^3 + -0.01$

Although it may appear that both have the same error, still, we can intuitively realize that the second measurement is more 'accurate' than the first one simply because 10^3 is greater than 1.

We can distinguish between these two ways of identifying error as absolute error and relative error.

1) Absolute error- The absolute error of a measurement is the difference between the approximate value and its underlying true value.

2) Relative error- The relative error of a measurement is the absolute error divided by the true value.

1.2.3 Measuring Error

One way to measure error is by directly computing the difference between the approximated and actual solution. However, in most cases, the true value is unknown. Thus, it is difficult to compute relative error in closed form. This is called as forward error.

The trouble with this obvious definitions of error is that the true value y may be unknown or unknowable. For example, suppose x=2 and we have the estimate of its square root as y=1.5. We have only approximations to the true value of y, such as 1.41 or 1.414 or 1.4142 or 1.41421 etc. James Wilkinson found an ingenious way around this problem. Instead of comparing the computed value y to the true value y Wilkinson asked what was the value of x that had y as a true value. He then determined the difference between this x and the true value of x and called it the backward error.

1.3 Conditioning and sensitivity

We introduce a new term called 'condition number', defining it as the measure of a problem's sensitivity.

> A problem is insensitive or well-conditioned when small amounts of backward error imply small amounts of forward error. In other words, a small change in the backward error results in only a minuscule change in the forward error.
>
> A problem is said to be sensitive, poorly conditioned, or sti when this is not well-conditioned.

Let us define condition number.

> The condition number of a problem is the ratio of how much its solution changes to the amount its statement changes under small perturbations. Alternatively, it is the ratio of forward to backward error for small changes in the problem statement.

References

[1] Samuel Conte and Carl de Boor, *Elementary Numerical Analysis – An Algorithmic Approach*, McGraw-Hill Education.

[2] Justin Solomon, *Numerical Algorithms.* Available online on the course website.

[3] Thayer Watkins, *http://applet-magic.com/numcomp.htm*