Department of Mathematics, University of Utah

Analysis of Numerical Methods I

MTH6610 – Section 001 – Fall 2017

Lecture notes: Floating-point representation

Monday September 25, 2017

These notes are not a substitute for class attendance. Their main purpose is to provide a lecture overview summarizing the topics covered.

Reading: Trefethen & Bau III, Lecture 13

The modern representation of numbers stored and manipulated internally on computers is *floating-point* format. First note that we usually represent numbers in the decimal ("base-10") system, e.g.,

$$34.1503$$

This representation has six digits, each taking a value between 0 and 9. The above representation actually means the following:

$$34.1503 = 3 \times 10^1 + 4 \times 10^0 + 1 \times 10^{-1} + 5 \times 10^{-2} + 0 \times 10^{-3} + 3 \times 10^{-4},$$

but the latter is obviously quite unwieldy. However, this decomposition reveals that we can represent numbers using any base $b \in \mathbb{N}$, $b \geq 2$ we like. Essentially, need only replace the number 10 above with a different base b to achieve a similar breakdown, of course corresponding to a different number:

$$b = 6, \qquad 34.1503 = 3 \times 6^1 + 4 \times 6^0 + 1 \times 6^{-1} + 5 \times 6^{-2} + 0 \times 6^{-3} + 3 \times 6^{-4},$$

where now the digits take values between 0 and $b - 1 = 5$. (Sometimes we write something like 34.1503_6 to emphasize that this number is written in base 6.) This procedure works for any valid base. Humans like base-10 things because our hands, and our feet, have 10 *digits*.

Computers internally represent and manipulate numbers with the presence or absence of an electrical impulse (1 = on or 0 = off), and so computers "prefer" base-2, or binary, representations. Floating-point representation is a way of writing decimal numbers in a type of base-2 way. We'll describe the high-level idea of modern floating-point representations, the vast majority of which are based on the IEEE 754 standard. A binary digit (a 0 or 1) is called a bit, and 8 sequential bits make up a byte.

Roughly speaking, a floating-point number consists of two integers, a "significand", and an "exponent". The popular *single precision* and *double precision* standards use 32 and 64 bits total, respectively, to represent a number. These bits are split between the significand and the exponent. A simplistic example with 10 bits total may devote 7 bits to the significand and 3 to the exponent:

$$1111001011 \to 1111001, \;\; 011 \to 1111001_2, \;\; 011_2 \to$$
$$1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + 0 + 0 + \frac{1}{64}, \quad 0 \times 2^2 + 2^1 + 2^0 \to 1.890625 \;\; 3 \to 1.890625 \times 2^3 = 15.125$$

Things like negative signs, Infs, Nans, and some detailed particulars make the actual machine-level map between bits and significand/exponent a little more complicated.

The double precision standard (8 bytes or 64 bits) allocates bits between the exponent and significant to allow exponents (in decimal) to vary between -1022 and 1024, and has approximately 16 decimal digits in the significand. Thus, the largest possible positive value that can be represented in double precision is approximately 2^{1024}, and the closest floating-point-representable number to 0 is approximately 2^{-1022}.

This representation also implies that the significand can only express numbers with a finite (16-digit) precision. This implies, for example, that the distance between the floating-point number 1 and the next largest floating point number is the minimum number that the significand can represent, which in decimal is around 10^{-16}. The maximum error one can then make when rounding exact numbers to their floating point representation is half of this distance; such errors due to floating-point rounding are called *roundoff error*.

This maximum rounding error (relative to 1) is called *machine epsilon*, often abbreviated ϵ_{mach} and is therefore a bound on relative floating-point errors due to rounding. Relative rounding errors, on the order of 10^{-16} in double precision floating-point, may seem insignificant, but these casue serious problems when performing some numerical computations. Here is a simplistic example: Consider the exact arithmetic computation

$$\frac{1}{\delta}\left[1 + \delta - 1\right] = 1$$

For δ smaller than machine precision, a straightforward floating-point arithmetic computation of the left-hand side will evaluate to 0 because $(1 + \delta)$ rounds to floating-point 1.

The phenomenon of order-1 errors (or larger) in computations due to machine precision limitations is called *loss of significance*. Loss of significance can sometimes be avoided by rearranging the order of computations. Here are some examples where loss of significance plays a role.

- Evaluation of $\sqrt{1+x^4} - 1$ for small, positive x. (Instead, compute as $x^4/(1+\sqrt{1+x^4})$.)
- Evalution of e^x for $x < 0$ using its (absolutely convergent!) Taylor series. (Instead, compute as $1/e^{-x}$.)
- Evaluation of $\frac{f(x+h)-f(x)}{h}$ for small h.

A relevant example for this class is what we saw earlier: classical Gram-Schmidt can sometimes provide very incorrect answers, and this happens when loss of significance occurs. Consider the matrix

$$A = \begin{pmatrix} 1 & 1 & 1 \\ \delta & 0 & 0 \\ 0 & \delta & 0 \\ 0 & 0 & \delta \end{pmatrix}$$

For any $\delta > 0$, this matrix is full rank and so there is no problem in directly applying classical Gram-Schmidt. We seek to orthogonalize these vectors, essentially computing the decomposition $A = QR$ for a 4×3 matrix Q with orthonormal columns q_j, and a 3×3 matrix R.

A computation in exact arithmetic shows that

$$q_1 = \frac{1}{\sqrt{1+\delta^2}} \begin{pmatrix} 1 \\ \delta \\ 0 \\ 0 \end{pmatrix}, \qquad r_{2,2} q_2 = a_2 - q_1 q_1^* a_2 = \frac{1}{1+\delta^2} \begin{pmatrix} (1+\delta^2) - 1 \\ -\delta \\ \delta(1+\delta^2) \\ 0 \end{pmatrix}$$

The problem here occurs when the first entry of $r_{2,2} q_2$ is truncated to 0 instead of the exact δ^2 when $\delta \ll 1$. This truncation does not adversely affect the actual vector q_2, since q_2 is still nearly orthogonal to q_1. To see this, consider the next step of generating q_3 from a_3:

$$q_2 = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 \\ -1 \\ 1 \\ 0 \end{pmatrix} \implies r_{3,3} q_3 = a_3 - q_1 q_1^* a_3 - q_2 q_2^* a_3 = \begin{pmatrix} 0 \\ -\frac{\delta}{1+\delta^2} \\ 0 \\ \delta \end{pmatrix} \implies q_3 \approx \frac{1}{\sqrt{2}} \begin{pmatrix} 0 \\ -1 \\ 0 \\ 1 \end{pmatrix}$$

Our computed q_3 is clearly not orthogonal to q_2.

The modified Gram-Schmidt procedure fixes this problem because it first computes the intermediate vector:

$$v_3 = a_3 - q_1^* a_3 \approx \begin{pmatrix} 0 \\ -\delta \\ 0 \\ \delta \end{pmatrix}$$

When one takes this vector and orthogonalizes against q_2, the (approximate) correct answer is obtained:

$$r_{3,3} q_3 = v_3 - q_2 q_2^* v_3 = \frac{\delta}{2} \begin{pmatrix} 0 \\ -1 \\ -1 \\ 2 \end{pmatrix}$$