Lecture 2: Numerical Analysis Review

Computational Acoustics, © V. Sparrow/PSU, 2000

Reference: Much of this lecture is taken from Section 1.5 of Rubin H. Landau and Paul J. Fink, Jr., A scientists and engineer’s guide to Workstations and Supercomputers (John Wiley, New York, 1993). Some other references are listed at the end of this lecture.

Last time:
- Syllabus (anyone else need one?)
- Computer availability

Access accounts:
- Do you have yours yet?

E-mail list
- Are you on yet?
- Please sign the sheet being passed around.

Reading assignment for next time:
- Chapters 1 and 2 of Todino, Strang, and Peek)
Representing Numbers on Computers

Computers can only represent mathematical quantities using

1. Alphanumeric characters
   → Symbolic representation (talk about later)
2. Integers, finite number of bits
3. Floating point numbers, finite number of bits

Memory requirements for the number 160,000:

\[
2. \quad < \quad 3. \quad < \quad 1. \\
\text{(most compact)} \quad \text{(least compact)}
\]
Bits and bytes:

Bits are 1’s and 0’s: 00011010001110110110001001011010 . . .

Recall the following:

- 1 bit = 1 b = either one 0 or one 1 (ON or OFF)
- 1 byte = 1 B = 8 bits = 1 alphanumeric character (“9”, “c”, “+”, etc.)
- 1 K = 1 KB = $2^{10}$ bytes = 1024 bytes ($\neq$ 1000 bytes)
- 512 K = $2^{19}$ bytes = 524,288 bytes
- 1 MEG = 1 MB = $2^{20}$ bytes = 1,048,576 bytes
- 1 GIG = 1 GB = $2^{30}$ bytes = 1,073,741,824 bytes

1 page of text is about 3 KB

Computations with integers:

N bits: $n_1, n_2, n_3, \ldots, n_N$ where $n_i$ are 0 or 1
- can represent $2^N$ integers
It is standard for $n_1$ to determine whether the integer is positive or negative:

\[ n_1 = 0, \Rightarrow \text{positive integer} \]
\[ n_1 = 1, \Rightarrow \text{negative integer} \]

This lets us represent integers in the range (of only)

\[ -2^{N-1} \text{ to } 2^{N-1} - 1 \]

4 byte integers are a standard on many computers today, and the range is

\[ -2,147,483,648 \text{ to } 2,147,483,647 \]

**Overflow:** when you try to represent an integer larger than the computer’s range allows.

**Strange things can happen when you try to go outside the range.**

Sometimes your computer will give you a warning message, but sometimes it will give you garbage.
Example programs creating overflow for integers (4 bytes or 32 bits)

Start with binary 0 = 000000000000000000000000000000

Add
\[ 2^{30} = 010000000000000000000000000000 \]

Add
\[ 2^{29} = 001000000000000000000000000000 \]

Add
\[ 2^{28} = 000100000000000000000000000000 \]

. . .

Add \[ 2^0 = 000000000000000000000000000001 \]

Get

\[ 011111111111111111111111111111 = 2,147,483,647 \text{ decimal} \]

What happens when we add 1 to this? [See program handouts.]
Pro’s and Con’s for integers:

**Con:** Can only represent integers.
\[ 3/2 = ? \]

**Pro:** Don’t have roundoff error which we have for . . .

**Floating point numbers:**

Represent by a sign bit \( s \), a mantissa, and an exponential field, \( \text{expfld} \), as

\[ x_{\text{float}} = (-1)^s \times \text{mantissa} \times 2^{\text{expfld} - \text{bias}} \]

For 32 bit (4 byte) floating point numbers, called **single precision numbers**:

Have

- 1 sign bit, \( s \)
- 8 bits for the exponent, \([0, 255]\)
- 23 bits for the mantissa

\[ \text{mantissa} = m_1 \times 2^{-1} + m_2 \times 2^{-2} + \ldots + m_{23} \times 2^{-23} \]
The bias (usually 127) is used to keep the expfld stored as all positive numbers. The range is roughly

\[ 2.90 \times 10^{-39} \text{ to } 3.40 \times 10^{38} \]

Example: \( 0.5 = 0 \ 0111 \ 1111 \ 1000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 000 \)

For 64 bit (8 byte) floating point numbers, called double precision numbers:

Have

- 1 sign bit, \( s \)
- 11 bits for the exponent
- 52 bits for the mantissa

The range is roughly

\[ 10^{-322} \text{ to } 10^{308} \]

These are good ranges. \( \frac{\text{size of universe}}{\text{size of proton}} \approx 10^{24} \)
Problems for floating point numbers:

For 32 bits, (similar for 64 bits)

1. If you try to represent a number larger than $3.40 \times 10^{38}$, you have an overflow. This is similar to what we have for integers:

$$3 \times 10^{38} + 3 \times 10^{38} = \text{JUNK}$$

2. If you try to represent a positive number smaller than $1.2 \times 10^{-38}$, you have an underflow. This is new. An underflow occurs when we try to represent a number closer to zero than the computer allows.

3. Can have roundoff error. Roundoff error occurs because computers represent numbers with a finite number of bits. It may be easiest to see this by a simple addition:

If we add $1.0 + 1.0 \times 10^{-7}$, we expect

$$1.0000001$$

However, for 32 bit floating point numbers we instead get

$$1.0$$
This is because single precision numbers only have $6 - 7$ digits of precision, and the small part we added to 1.0 was lost. This is a simple example of roundoff error.

**Precision in scientific/engineering computing:**
Double precision numbers have approximately 16 digits of precision. You can also have overflow, underflow, and roundoff error with double precision numbers, but this is less common.

**If in doubt, use double precision!**
Then if you add $1.0 + 1.0 \times 10^{-7}$ you get 1.0000001.

**Machine precision:**
This is a useful definition we will need later:
Definition: the machine precision of a computer is the largest positive number $\varepsilon$ such that

$$1.0 + \varepsilon = 1.0$$
Please note that the machine precision is NOT the smallest number the computer can represent! Some authors define machine precision differently, but the above definition is becoming a standard. Note that single and double precision numbers will have different machine precisions.

**Accounting for precision** is very useful, since we can then minimize roundoff error. Can you guess what is the best way to add up a list of many numbers whose absolute values are different by many orders of magnitude?

[answer given in class]

This is a good thing for people in acoustics to understand. When finding the total power in a spectrum, the spectral components may differ by many dB.
How poor can a calculation be?
Assume there are $M$ floating point operations.
If you are lucky,

$$\text{error} \propto \varepsilon \times \sqrt{M}$$

But it is often the case that

$$\text{error} \propto \varepsilon \times M$$

This means your final answer can be way off without you knowing it. Some calculations create MUCH LARGER ERRORS!

Example difficulty: The subtraction of two nearly equal numbers
Recall that the quadratic equation

$$ax^2 + bx + c = 0$$

has the solutions

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$
The difficulty you don’t learn in high school is when $4ac \ll b^2$ in the + root:

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

This looks very similar to

$$x = \frac{-b + (b - \text{tiny})}{2a}$$

and you are subtracting nearly equal numbers.

The result is a severe loss of precision.
Some excellent related references:


