

2. 780.20 Session 2

a. Follow-ups to Session 1

First, a couple of general comments on the 1094 sessions.

- Please try to not skip items if you have time; everything is there for a purpose, even though it might not be obvious to you (and might seem pointless or too easy). If you are already quite familiar with a topic, you can make the task more challenging or ask (and try to answer) a more complicated question (e.g., the “To do” list in the `area.cpp` program).
- Take the verification of codes very seriously. *It's important!* The modified code to calculate the volume is a good example of how you have to be careful. A typical modification of the calculation line is:

```
double volume = 4/3 *pi * radius * radius * radius; // volume formula
```

Verifying with a value of `radius` equal to 1.0 would yield an answer of π , which is clearly wrong. Two things are happening. First, C++ interprets the 4 and the 3 as integers, because they do not have decimal points. Second, C++ evaluates `4/3` first. Since both are integers, the answer is 1, which is *then* converted to a floating-point number when multiplied by `radius`.

Summary: If a constant is supposed to be a floating-point number, write it as a floating-point number: not 3, but 3.0.

Here's a question to check your understanding: What would be the value of `x` after the statement `float x = 1/2;`

- Another case of verification is checking against a theoretical answer. Sometimes a disagreement is because the theory is wrong (see the next section) but often the program has a mistake or is not doing what it is designed to you. If you determined the machine precision to be roughly 10^{-12} in Session 1, you need to go back and figure out why you didn't get the expected answer, which is of order 10^{-16} . (Hint: There is a design flaw in the program because `setprecision` limits the precision of the printed number, even if internally it has greater precision. So `setprecision(12)` means *at most* 12 digits will be printed.)

Some other comments on Session 1 items:

- Integer powers of numbers. In fortran, you would use `radius**3`, while in Mathematica or MATLAB you would use `radius^3` to calculate the cube of a number. In C or C++ there is a library function called `pow`. To find the cube of `radius`, you could use `pow(radius,3)`. However, this is not advisable, because `pow` treats the power as a real number, which means that it will be very inefficient (e.g., it might use logarithms to do the calculation). We'll learn later on how to define *inline functions* to do simple operations like squaring or cubing a number.
- The “manipulator” `endl` used with `cout` (which is short for “console output”) indicates “endline”, which means a new line. (You could achieve the same effect with the C-style `\n`.)

- If you change the compiler or linker options in Dev-C++, you should use the “Rebuild All” button to make sure the change is incorporated in the project makefile. The makefile is essentially a list of instructions on how to compile and link the code.
- When determining a quantity like the machine precision, if you only find it to within (say) 10%, you shouldn’t give 10 digits in your answer, but only a couple.

b. Follow-up On Underflows

The theoretical discussion of minimum numbers in the notes for Session 1 was based on assumptions not completely consistent with what is called the IEEE floating point standard (which is what is used in C++ on almost all computers). That is why the theory didn’t match the Session 1 computer experiment! For example, you found single-precision underflow at about 10^{-45} instead of 10^{-38} . The following discussion is based on the the section on “IEEE floating-point arithmetic” in the online GSL manual.

The way our notes for Session 1 define m_1 , it is the coefficient of 2^{-1} in the mantissa. That is, it is the beginning of the fraction, and the fraction is between 0 and 1. The IEEE 754 standard is that there is always a 1 in front of the mantissa (i.e., 2^0) for what is called a *normalized* number. Thus, the mantissa is always between 1 and 2, rather than between 0 and 1 (i.e., `1.ffffff...`, where the `f`’s can be either 0’s or 1’s). The exponent for the normalized numbers are not allowed to be all 0’s, as allowed in the Session 1 notes, so the minimum exponent for normalized single precision numbers is 0000001, meaning $2^{1-127} = 2^{-126}$.

But there are also *denormalized* numbers, which are signaled by exponents with all 0’s but a nonzero fraction. For these, the mantissa is assumed to be of the form `0.fffff...`, with the `f`’s again either 0 or 1. This means that the smallest possible mantissa is 2^{-23} (0’s everywhere but the last place on the right), so that the smallest positive number is 2^{-149} , as found in Session 1. An analogous discussion applies for double-precision numbers. Zero is represented with 0’s everywhere (exponent and mantissa) and can have either sign.

To summarize, single-precision floating-point numbers can be normalized:

$$(-1)^{\text{sign}} \times (1 + m_1 \times 2^{-1} + m_2 \times 2^{-2} + \dots + m_{23} \times 2^{-23}) \times 2^{[(\text{exponent field}) - \text{bias}]}, \quad (2.1)$$

with the m_i ’s either 0 or 1 and the “exponent field” from 1 to 255, or denormalized:

$$(-1)^{\text{sign}} \times (m_1 \times 2^{-1} + m_2 \times 2^{-2} + \dots + m_{23} \times 2^{-23}) \times 2^{-126}, \quad (2.2)$$

or zero. For double precision, $23 \rightarrow 52$, $255 \rightarrow 2047$, and $126 \rightarrow 1022$.

c. Round-Off Errors

We can envision various classes of errors that might occur in a computation physics project. For example [1]:

1. Blunders — typographical errors, programming errors, using the wrong program or input file. If you invoke all the compiler warning options (see the handout on “Recommended C++ Options”) and don’t proceed further until you have eliminated all warnings, you will catch most of the first two types of errors. (We’ll try this in Session 2.)
2. Compiler errors — that is, bugs in the compiler. These are really insidious but do happen (I could tell you stories . . .). Fixes include trying more than one compiler (we’ll use both g++ and the Intel C++ compiler, called icpc) or check another way, such as using Mathematica. Another type of compiler error sometimes comes from “optimization”. We’ll discuss this later, but the rule to deal with this is always to compile and test your code first without optimization (which means to use the compiler option `-O0`).
3. Random errors — e.g., cosmic rays or power surges changing bits in your machine. This is not likely to be a problem, but you can detect them (and detect other problems) by reproducing at least part of your output with repeated runs and/or on a different computer.
4. Approximation (or truncation) errors. Here’s an example, calculating a function using a truncated Taylor series expansion:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} \approx \sum_{n=0}^N \frac{x^n}{n!} = e^x + \mathcal{E}(x, N) , \quad (2.3)$$

where \mathcal{E} is the total absolute error. We’ll look at these type of errors in more detail starting in Session 3. In this case, we can say that for small x the error goes like the first omitted term, which we can approximate:

$$\mathcal{E}(x, N) \approx \frac{x^{N+1}}{(N+1)!} \approx \left(\frac{x}{N+1} \right)^{N+1} \quad \text{for } x \ll 1 . \quad (2.4)$$

For fixed x , this means that $\mathcal{E} \propto N^{-N}$. We will typically find that truncation errors follow a power law N^α for constant α (e.g., for numerical differentiation or for numerical integration using the trapezoid or Simpson’s rules). We can discover whether this is true and identify α by plotting $\log \mathcal{E}$ against $\log N$.

5. Round-off errors. These come about because the computer representation of floating-point numbers is approximate (from the Session 1 notes: z_c is the *computer representation* of z):

$$z_c = z(1 + \epsilon) \quad \text{with} \quad |\epsilon| \lesssim \epsilon_m , \quad (2.5)$$

where ϵ_m is the machine precision (the largest number such that $1 + \epsilon_m = 1$). Operations *accumulate* errors, depending on how numbers are combined (e.g., subtracting or multiplying). This is the type of error we’ll focus on in this session.

Round-off errors show up most dramatically when there is a *subtractive cancellation*: subtracting two numbers close in magnitude. So, for example, if we have 5 decimal digits of accuracy and we add $z_1 = 1.234567$ and $z_2 = 1.234569$, then the answer is still accurate to five digits, but if we subtract them we get garbage rather than 0.000002 because the computer representations z_{1c} and z_{2c} don’t have enough digits.

Let's see how it works formally. Let

$$z_3 = z_1 - z_2 , \quad (2.6)$$

which on the computer becomes:

$$z_{3_c} = z_{1_c} - z_{2_c} = z_1(1 + \epsilon_1) - z_2(1 + \epsilon_2) = z_3 + z_1\epsilon_1 - z_2\epsilon_2 . \quad (2.7)$$

Then the *relative* error in z_3 , which is ϵ_3 from

$$z_{3_c} = z_3(1 + \epsilon_3) , \quad (2.8)$$

is

$$|\epsilon_3| = \left| \frac{z_{3_c} - z_3}{z_3} \right| = \left| \frac{z_{3_c}}{z_3} - 1 \right| = \left| \frac{z_1}{z_3}\epsilon_1 - \frac{z_2}{z_3}\epsilon_2 \right| . \quad (2.9)$$

What does this imply? Consider cases. If z_1 and/or z_2 is comparable in magnitude to z_3 , then the error in z_3 is the same magnitude as ϵ_1 or ϵ_2 , e.g.,

$$z_1 \ll z_2 \quad \text{or} \quad z_2 \ll z_1 \quad \text{or} \quad z_1 \approx -z_2 \quad \implies \quad \epsilon_3 \sim \epsilon_1, \epsilon_2 . \quad (2.10)$$

(Note that $|\epsilon_1 - \epsilon_2|$ is the same order of magnitude as ϵ_1 since they are assumed to be randomly distributed.) So the error stays about the same size. But if z_1 and z_2 are about the same magnitude and sign, the error is *magnified*, since in this case

$$z_1 \approx z_2 \quad \implies \quad z_3 \ll z_1, z_2 \quad (2.11)$$

and so

$$|\epsilon_3| \approx \left| \frac{z_1}{z_3}(\epsilon_1 - \epsilon_2) \right| \gg \epsilon_m \quad (\text{in general}). \quad (2.12)$$

The ordinary quadratic equation provides an instructive example. There are two ways we can solve it, by writing it in two ways:

$$ax^2 + bx + c = 0 \quad \text{or} \quad y = \frac{1}{x} \implies cy^2 + by + a = 0 , \quad (2.13)$$

and then applying the usual quadratic formula to obtain two formulas for each of the roots:

$$x_1 \equiv \frac{-b + \sqrt{b^2 - 4ac}}{2a} = x'_1 \equiv \frac{-2c}{b + \sqrt{b^2 - 4ac}} , \quad (2.14)$$

$$x_2 \equiv \frac{-b - \sqrt{b^2 - 4ac}}{2a} = x'_2 \equiv \frac{-2c}{b - \sqrt{b^2 - 4ac}} . \quad (2.15)$$

Which is the best to use for numerical calculations, x_1 or x'_1 ?

Analysis for $b > 0$. Let

$$z_1 = b \quad \text{and} \quad z_2 = \sqrt{b^2 - 4ac} \approx b - 2ac/b , \quad (2.16)$$

where we used $(1+x)^n \approx 1+nx$ for small x . Then Eq. (2.12) tells us that

$$|\epsilon_3| \approx \left| \frac{b}{2ac/b}(\epsilon_1 - \epsilon_2) \right| \approx \left| \frac{b^2}{2ac} \right| \epsilon_m . \quad (2.17)$$

Looking at Eqs. (2.14) and (2.15), we see that there will be two “bad” expressions, for x_1 and x'_2 , with relative errors

$$\left| \frac{x_1 - x'_1}{x'_1} \right| \approx |\epsilon_3| \approx \left| \frac{b^2}{2ac} \right| \epsilon_m \gg \epsilon_m \quad (2.18)$$

and

$$\left| \frac{x'_2 - x_2}{x_2} \right| \approx |\epsilon_3| \approx \left| \frac{b^2}{2ac} \right| \epsilon_m \gg \epsilon_m . \quad (2.19)$$

So we should use the formula for x'_1 for the first root and the formula for x_2 for the second root! How do we see if the predicted behavior is correct? Generate roots for a wide range of small c values and plot

$$\text{Log}_{10} \left| \frac{x_1 - x'_1}{x_1} \right| \quad \text{vs.} \quad \text{Log}_{10} \frac{1}{c} . \quad (2.20)$$

If Eq. (2.18) holds, what should the slope be and what should be the approximate extrapolation to $c = 1$?

We'll investigate this example in Session 3 using a test case:

$$a = 1, \quad b = 2, \quad c = 10^{-n}, \quad \text{for } n = 1, 2, 3, \dots \quad (2.21)$$

so that $c \ll a \approx b$. We can devise a pseudo-code to investigate errors:

```

input a,b,c
calculate discriminant:  disc =  $\sqrt{b^2 - 4ac}$ 
find roots:
   $x_1 = (-b + disc)/2a$ 
   $x'_1 = -2c/(b + disc)$ 
   $x_2 = (-b - disc)/2a$ 
   $x'_2 = -2c/(b - disc)$ 
output  $x_1, x'_1, x_2, x'_2$  (with many digits)

```

This pseudo-code is implemented in `quadratic_equation_1a.cpp` (see printout). This version has several bugs and is not indented. You are to fix both problems in Session 2, ending up with the corrected version `quadratic_equation_1.cpp`. Then the extended version `quadratic_equation_2.cpp` generates an output file that you will plot using `gnuplot` to test the analysis above.

This example illustrates an important maxim in numerical analysis (according to me!), which we will cite repeatedly:

“It matters how you do it.”

Different algorithms for doing a calculation, although equivalent mathematically, can be very different in their accuracy (and efficiency). We must always remember that

“Computer math \neq regular math.”

Another good example of these maxims is given in section 2.2 of the Hjorth-Jensen notes [2] (linked on the web page under “Supplementary Reading”). He considers the function

$$f(x) = \frac{1 - \cos(x)}{\sin(x)}, \quad (2.22)$$

for small values of x . An equivalent representation (mathematically!) is obtained by multiplying top and bottom by $1 + \cos(x)$ and simplifying:

$$f(x) = \frac{\sin(x)}{1 + \cos(x)}. \quad (2.23)$$

Next he supposes that a floating-point number is represented on the computer with only five digits to the right of the decimal point. Then if we take $x = 0.007$ radians, we will have

$$\sin(0.007) \approx 0.69999 \times 10^{-2} \quad \text{and} \quad \cos(0.007) \approx 0.99998 \times 10^0. \quad (2.24)$$

Using the first expression to evaluate $f(x)$, we get

$$f(x) = \frac{1 - 0.99998}{0.69999 \times 10^{-2}} = \frac{0.2 \times 10^{-4}}{0.69999 \times 10^{-2}} = 0.28572 \times 10^{-2}, \quad (2.25)$$

while using the second yields

$$f(x) = \frac{0.69999 \times 10^{-2}}{1 + 0.99998} = \frac{0.69999 \times 10^{-2}}{1.99998} = 0.35000 \times 10^{-2}. \quad (2.26)$$

The second result is the exact result but the first one is way off! In fact, with our choice of precision there is only one relevant digit in the numerator after the subtraction of two nearly equal numbers. Note that this result doesn't mean the second expression is always better; for $x \approx \pi$ the second expression loses precision.

Hjorth-Jensen gives several more examples in section 2.2 [2], which I recommend you read through. This includes an illuminating discussion of three possible algorithms for computing e^{-x} .

d. How do you tell if two numbers are equal?

Here is a problem for you to think about. We'll come back to it in a future session.

Suppose in a computational physics program we want to compare two floating-point numbers. The pseudo-code might be:

```

if x is equal to y
    print "they are equal"
otherwise print "they are different"

```

How should we implement this? Why does the computer representation of floating-point numbers make this a non-trivial problem?

e. Spherical Bessel Functions

Here are some brief comments on spherical Bessel functions, which arise frequently in physics problems (see the chapter 3 handout from Ref. [1] for more details). For example, in the decomposition of a plane wave,

$$e^{i\mathbf{k}\cdot\mathbf{r}} = \sum_{l=0}^{\infty} i^l (2l+1) j_l(kr) P_l(\cos\theta), \quad (2.27)$$

we need to know the $j_l(kr)$ for various $l = 0, 1, 2, \dots$ for given values of kr . The first two are easy:

$$j_0(x) = \frac{\sin x}{x} \quad \text{and} \quad j_1(x) = \frac{\sin x - x \cos x}{x^2}, \quad (2.28)$$

but what about higher l ?

We could solve the differential equation that they satisfy:

$$x^2 f''(x) + 2x f'(x) + [x^2 - l(l+1)]f(x) = 0 \quad (2.29)$$

or we could use recursion relations:

$$j_{l+1}(x) = \frac{2l+1}{x} j_l(x) - j_{l-1}(x) \quad [\text{up}] \quad (2.30)$$

$$j_{l-1}(x) = \frac{2l+1}{x} j_l(x) - j_{l+1}(x) \quad [\text{down}] \quad (2.31)$$

and the boundary values from (2.28). To go “up” for a given value of x , start with $j_0(x)$ and $j_1(x)$ for (2.28), then find $j_2(x)$ from (2.30). Given $j_2(x)$ and $j_1(x)$, we can find $j_3(x)$ from (2.30), and so on. To recurse downward, we use the fact that for fixed x and large l , $j_l(x)$ decreases rapidly with l . So we start with *arbitrary* values for $j_{l_{\max}}(x)$ and $j_{l_{\max}-1}(x)$. Then we use (2.31) to find $j_{l_{\max}-2}(x), \dots, j_1(x)$, and $j_0(x)$. Then we *rescale* all of the values to the known value of $j_0(x)$ from (2.28) and we have them all. (You will probably have to think this through carefully!)

So why is one way better or worse, depending on the values of l and x ? [Hint: How many solutions are there to the differential equation?]

f. References

- [1] R.H. Landau and M.J. Paez, *Computational Physics: Problem Solving with Computers* (Wiley-Interscience, 1997).
- [2] M. Hjorth-Jensen, *Computational Physics*. These are notes from a course offered at the University of Oslo. See the 780.20 webpage for links to excerpts and the full text.
- [3] W. Press *et al.*, *Numerical Recipes in C* (Cambridge, 1992). Individual chapters are available online from <http://lib-www.lanl.gov/numerical/bookcpdf.html>. There are also versions for Fortran and C++.