2. 6810 Session 2

a. Follow-ups to Session 1

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

- 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 an 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. Because both are integers, the answer is `1`, which is *then* converted to a floating-point number when multiplied by `radius`.

  > *Moral: If a constant is supposed to be a floating-point number, write it as a floating-point number: not 3, but 3. or 3.0. Always, always, always!*

  Here's a question to check your understanding: What would be the value of `x` after the statement `float x = 1/2;` ?
  Note: This definition of volume:
  ```
  double volume = 4 * pi * radius * radius * radius/3; // volume formula
  ```
  would work, because the first 4 and the last 3 would be converted to doubles. But (4./3.) is much safer (and therefore much better).
- Checking against a known answer is one of many ways to verify a code is working correctly. The known answer is typically a special case (like a `radius` of 1) or a limiting case (like a `radius` of 0). In many cases you might know there should be a *scaling law*, e.g., that the volume should scale as `radius` cubed, but you may not know the coefficient. So you compare results from two different radii, say 1 and 10, and check whether the answer increases by a factor of 1000 (this would catch the mistake of forgetting to change from radius squared to radius cubed, even if you correctly included the factor of 4./3.). More generally, you can plot the answer vs. input on a log-log plot and look for straight lines with definite slopes—we'll be doing this a lot!
- 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 do. 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.)

- There may be a conflict between writing code that is *optimized* (i.e., runs fastest) and that is clear (e.g., can be easily compared to written equations). *Always* code for clarity first, verify that it is correct, and then optimize!

Some other comments on Session 1 items:

- Integer powers of numbers. In Fortran or Python, 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 *may* not be not advisable, because `pow` treats the power as a real number, which means that it might 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. [Note: it is possible a compiler could replace `pow` automatically or that the performance penalty is negligible in any case. We'll see later how to check if it is a problem.]
- 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`. More on manipulators such as `setprecision` later.)
- 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. In Session 1, what you really determined were upper and lower bounds for such quantities.
- The "intentional bug" in precision.cpp was that the `#include <fstream>` line at the top was left out. This statement makes available commands related to file output, including `ofstream`, which is used in the program to open the file we print to. If it is not there, the error is:

  ```
  error: variable 'std::ofstream my_out' has initializer but incomplete type
  ```

  It is common to forget these `include` statements, so I wanted you to be familiar with the type of error message you might get. Keep this in mind for Session 2!
- If you are baffled by an error message, it is often helpful to cut-and-paste the message into Google (deleting any specific parts like the name of your program). Within the top few results you will frequently find the explanation of (and possibly the solution to) your problem. Also use Google to learn about C++ or Python (or ...) commands, e.g., use "C++ printing" or "python spherical bessel function" as search terms.
- Floating-point underflows in an intermediate step of a numerical calculation are often harmless in practice as long as it is ok that they are treated as zero (e.g., if you are adding numbers). But an overflow can occur in an intermediate step even if the final answer is in the valid range of floating point numbers, and this is usually a disaster. However, it can be avoided by rearranging or reformulating your calculation. See the example in Section 2.4 of the Hjorth-Jensen notes of calculating e^{-x} using a series expansion:

$$e^{-x} = \sum_{n=0}^{\infty} (-1)^n \frac{x^n}{n!} \ , \tag{2.1}$$

keeping enough terms until the next term is less than a desired accuracy. With this method, computing factorials leads to overflows. How is the problem averted in this case?

- One of the most important lessons to take away from Session 1 is that there are only a finite number of floating point numbers that can be represented. The consequences are minimum and maximum numbers (leading to underflow and overflow), and that the computer representation of only certain numbers are exact (called "machine numbers"). Which numbers are machine numbers can be surprising if we forget that they are stored as binary numbers. This means that $5 \rightarrow 101.0$ and $1/4 \rightarrow 0.01$ are machine numbers but $1/5$ is an infinite sequence after the period (should I call it a "binary point"?), like $1/3$ in base ten. The worst *relative* error you can make is the machine precision ϵ_m (more on this and round-off errors below).

b. Further Comments on Writing Programs

Let's consider the `area.cpp` code from Session 1 again and think of the one-line calculation as a placeholder for a more complex calculation. For example, solving for the energies of atoms using matrix methods. Here are some common features with more complicated codes:

- there is more than one way (or algorithm) to solve the problem (e.g., calculating r^2 as `r*r` or `pow(r,2)` in `area.cpp`);
- there is more than one way to input data, e.g., interactive input (as in `area.cpp`) or from a file, or we might want to calculate a range of values;
- there is more than one thing we might want to do with the output, e.g., print to the screen, save in a file or make plots;
- there may be related or more general quantities we want to calculate (e.g., the circumfrence of a circle or the volume of a sphere).

How should we structure our programs so that our code can be extended in these ways *while still being reliable*? The latter means we need to be able to test our calculations and then have confidence that they are still correct when we change the code.

As we advance through the course, we will consider how to make our codes more *modular* and *encapsulated.* The original codes we use in class will often lack these features for one reason or another (often on purpose). You should critically examine them for how they could be improved.

c. 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 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 $2^{-23} \times 2^{-126} = 2^{-149}$, 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:

$$(-1)^{\text{sign}} \times (1 + m_1 \times 2^{-1} + m_2 \times 2^{-2} + \cdots + m_{23} \times 2^{-23}) \times 2^{[(\text{exponent field})-\text{bias}]} , \tag{2.2}$$

with the m_i's either 0 or 1 and the "exponent field" from 1 to 255, or denormalized:

$$(-1)^{\text{sign}} \times (m_1 \times 2^{-1} + m_2 \times 2^{-2} + \cdots + m_{23} \times 2^{-23}) \times 2^{-126} , \tag{2.3}$$

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

This discussion can explain what seemed to be a peculiar result from Session 1. When determining the machine precision for single precision numbers (i.e., `float`'s), the output file ended:

```
1.00000011921  7.59377911663e-08
1.00000011921  6.90343568976e-08
1.00000011921  6.27585095003e-08
```

Clearly the number on the left is not equal to one plus the number on the right. What is happening? Suppose we are not yet down to machine precision, but just above. Then from Eq. (2.2) that means that all of the m_i's are zero except for $m_{23} = 1$. Sure enough, $2^{-23} \approx 1.19209 \times 10^{-7}$. The previous different numbers on the right were 1.00000023842 and 1.00000035763. Can you explain them?

d. 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 typos and many programming 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 for us, but becomes more and more relevant as the number of parallel processors ("cores") increases. 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:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} \approx \sum_{n=0}^{N} \frac{x^n}{n!} = e^x + \mathcal{E}(x,N) \ , \tag{2.4}$$

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:

$$\mathcal{E}(x,N) \approx \frac{x^{N+1}}{(N+1)!} \approx \left(\frac{x}{N+1}\right)^{N+1} \quad \text{for} \quad x \ll 1 \ . \tag{2.5}$$

For fixed x, this means that $\mathcal{E} \propto N^{-N}$. In many cases we will 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):

$$z_c = z(1+\epsilon) \quad \text{with} \quad |\epsilon| \lesssim \epsilon_m \ , \tag{2.6}$$

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_{1_c} and z_{2_c} don't have enough digits. More generally, subtractive cancellations will cause a reduction in precision, which can accumulate with successive operations.

Let's see how it works formally. Let

$$z_3 = z_1 - z_2 \ , \tag{2.7}$$

which on the computer becomes:

$$z_{3_c} = z_{1_c} - z_{2_c} = z_1(1+\epsilon_1) - z_2(1+\epsilon_2) = z_3 + z_1\epsilon_1 - z_2\epsilon_2 \ . \tag{2.8}$$

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

$$z_{3_c} = z_3(1+\epsilon_3) \ , \tag{2.9}$$

is

$$|\epsilon_3| \overset{(2.9)}{=} \left|\frac{z_{3_c} - z_3}{z_3}\right| = \left|\frac{z_{3_c}}{z_3} - 1\right| \overset{(2.8)}{=} \left|\frac{z_1}{z_3}\epsilon_1 - \frac{z_2}{z_3}\epsilon_2\right| \ . \tag{2.10}$$

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.,

$$z_1 \ll z_2 \quad \text{or} \quad z_2 \ll z_1 \quad \text{or} \quad z_1 \approx -z_2 \quad \Longrightarrow \quad \epsilon_3 \sim \epsilon_1, \epsilon_2 \ . \tag{2.11}$$

(Note that $|\epsilon_1 - \epsilon_2|$ is the same order of magnitude as ϵ_1 because 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*, because in this case

$$z_1 \approx z_2 \quad \Longrightarrow \quad z_3 \ll z_1, z_2 \tag{2.12}$$

and so

$$|\epsilon_3| \approx \left|\frac{z_1}{z_3}(\epsilon_1 - \epsilon_2)\right| \gg \epsilon_m \quad \text{(in general).} \tag{2.13}$$

The ordinary quadratic equation provides an instructive example of how round-off errors can make two mathematically equivalent ways to evaluate an expression give very different results computationally. Consider two ways to solve the quadratic equation, by writing it in two ways:

$$ax^2 + bx + c = 0 \quad \text{or} \quad y = \frac{1}{x} \Longrightarrow cy^2 + by + a = 0 \ , \tag{2.14}$$

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

$$x_1 \equiv \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad = \quad x_1' \equiv \frac{-2c}{b + \sqrt{b^2 - 4ac}} \ , \tag{2.15}$$

$$x_2 \equiv \frac{-b - \sqrt{b^2 - 4ac}}{2a} \quad = \quad x_2' \equiv \frac{-2c}{b - \sqrt{b^2 - 4ac}} \ . \tag{2.16}$$

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

Analysis for $b > 0$ (and then you do $b < 0$). Let

$$z_1 = b \quad \text{and} \quad z_2 = \sqrt{b^2 - 4ac} \approx b - 2ac/b \ , \tag{2.17}$$

where we used $(1+x)^n \approx 1 + nx$ for small x. Then Eq. (2.13) tells us that

$$|\epsilon_3| \approx \left|\frac{b}{2ac/b}(\epsilon_1 - \epsilon_2)\right| \approx \left|\frac{b^2}{2ac}\right| \epsilon_m \ . \tag{2.18}$$

Looking at Eqs. (2.15) and (2.16), we see that there will be two "bad" expressions, for x_1 and x_2', with relative errors

$$\left|\frac{x_1 - x_1'}{x_1'}\right| \approx |\epsilon_3| \approx \left|\frac{b^2}{2ac}\right| \epsilon_m \gg \epsilon_m \tag{2.19}$$

and

$$\left|\frac{x_2' - x_2}{x_2}\right| \approx |\epsilon_3| \approx \left|\frac{b^2}{2ac}\right| \epsilon_m \gg \epsilon_m \ . \tag{2.20}$$

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

$$\mathrm{Log}_{10}\left|\frac{x_1 - x_1'}{x_1}\right| \quad \text{vs.} \quad \mathrm{Log}_{10}\,\frac{1}{c} \ . \tag{2.21}$$

[Note: You can graph $\log_{10} y$ vs. $\log_{10} x$ on a linear plot or y vs. x on a log–log plot.] *If Eq. (2.19) holds, what should the slope be and what should be the approximate extrapolation to $c = 1$?*

We'll investigate this example in Session 2 using a test case:

$$a = 1,\ b = 2,\ c = 10^{-n}, \quad \text{for } n = 1, 2, 3, \cdots \tag{2.22}$$

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`. 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!):

"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

"Computational math $\neq$ regular math."

Another good example of these maxims is given in section 2.3 of the Hjorth-Jensen notes [2] (linked on the web page under "Supplementary Reading"). He considers the function

$$f(x) = \frac{1-\cos(x)}{\sin(x)} \;, \tag{2.23}$$

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

$$f(x) = \frac{\sin(x)}{1+\cos(x)} \;. \tag{2.24}$$

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

$$\sin(0.007) \approx 0.69999 \times 10^{-2} \quad \text{and} \quad \cos(0.007) \approx 0.99998 \times 10^{0} \;. \tag{2.25}$$

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

$$f(x) = \frac{1-0.99998}{0.69999\times 10^{-2}} = \frac{0.2\times 10^{-4}}{0.69999\times 10^{-2}} = 0.28572\times 10^{-2} \;, \tag{2.26}$$

while using the second yields

$$f(x) = \frac{0.69999\times 10^{-2}}{1+0.99998} = \frac{0.69999\times 10^{-2}}{1.99998} = 0.35000\times 10^{-2} \;. \tag{2.27}$$

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 $x \approx \pi$ it is the second expression that loses precision.

e. Further examples of subtractive cancellations

We will explore in Session 2 a simplified version of problem 3 from section 3.4 of the Landau-Paez text [1], where subtractive cancellations enter to haunt apparently simple summations. Consider the two series for integer N:

$$S^{(\mathrm{up})} = \sum_{n=1}^{N} \frac{1}{n} \qquad S^{(\mathrm{down})} = \sum_{n=N}^{1} \frac{1}{n} \tag{2.28}$$

Mathematically they are finite and equivalent, because it doesn't matter in what order you do the sum. However, when you sum numerically, $S^{(\mathrm{up})} \neq S^{(\mathrm{down})}$ because of round-off error. You will analyze this phenomena in the first homework assignment.

Hjorth-Jensen gives several more examples in section 2.4 of Ref. [2], which I recommend that you read through (it is linked on the web page). This includes an illuminating discussion of three possible algorithms for computing e^{-x}. The problem in this case arises from a sum with alternating signs, for which a naive treatment is almost guaranteed to be plagued by subtractive cancellations. So don't be naive! For example, which of the following expressions for the same finite sum do you think will be affected least by subtractive cancellations [1]:

$$S_N^{(1)} = \sum_{n=1}^{2N} (-1)^n \frac{n}{n+1} , \quad S_N^{(2)} = -\sum_{n=1}^{N} \frac{2n-1}{2n} + \sum_{n=1}^{N} \frac{2n}{2n+1} , \quad S_N^{(3)} = \sum_{n=1}^{N} \frac{1}{2n(2n+1)} . \tag{2.29}$$

(Hint: How many subtractive cancellations are there in each expression?)

f. 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?

g. Spherical Bessel Functions

Here is some background on spherical Bessel functions, which arise frequently in physics problems (the discussion is based on chapter 3 in Ref. [1]). For example, when we apply the decomposition of a plane wave (θ is the angle between $\mathbf{k}$ and $\mathbf{r}$),

$$e^{i\mathbf{k}\cdot\mathbf{r}} = \sum_{l=0}^{\infty} i^l \,(2l+1) j_l(kr) P_l(\cos\theta) , \tag{2.30}$$

we need to know the $j_l(kr)$ for various $l = 0, 1, 2, \ldots$ for given values of kr. The first two are well known (see Fig. 1 for plots):

$$j_0(x) = \frac{\sin x}{x} \qquad \text{and} \qquad j_1(x) = \frac{\sin x - x\,\cos x}{x^2} , \tag{2.31}$$

but what about higher l?

We could solve the differential equation that they satisfy:

$$x^2 f''(x) + 2x f'(x) + [x^2 - l(l+1)] f(x) = 0 , \tag{2.32}$$

where $j_l(x)$ is the solution that is regular (nonsingular) at the origin. A second, independent solution, irregular at the origin, is the spherical Neumann function $n_l(x)$. These functions have the

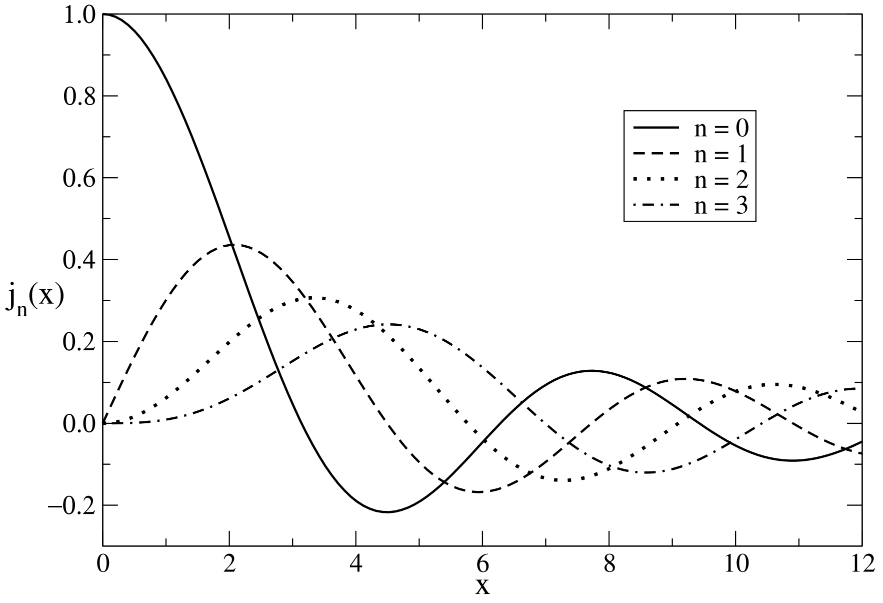

Figure 1: Spherical Bessel functions, $j_n(x)$, as functions of x for $n = 0, 1, 2, 3$.

limiting behavior:

$$\begin{aligned} j_l(x) &\longrightarrow x^l/(2l+1)!! \quad \text{for } x \ll l\,, \\ n_l(x) &\longrightarrow -(2l-1)!!/x^{l+1} \quad \text{for } x \ll l\,, \\ j_l(x) &\sim \sin(x - l\pi/2)/x \quad \text{for } x \gg l\,, \\ n_l(x) &\sim -\cos(x - l\pi/2)/x \quad \text{for } x \gg l\,, \end{aligned} \tag{2.33}$$

where $(2l+1)!! \equiv 1 \cdot 3 \cdot 5 \cdots (2l+1)$.

An alternative to solving the differential equation is to use recursion relations:

$$j_{l+1}(x) = \frac{2l+1}{x} j_l(x) - j_{l-1}(x) \quad \text{[up]} \tag{2.34}$$

$$j_{l-1}(x) = \frac{2l+1}{x} j_l(x) - j_{l+1}(x) \quad \text{[down]} \tag{2.35}$$

and the boundary values from Eq. (2.31). To go "up" for a given value of x, start with $j_0(x)$ and $j_1(x)$ in Eq. (2.31), then find $j_2(x)$ from Eq. (2.34). Given $j_2(x)$ and $j_1(x)$, we can find $j_3(x)$ from Eq. (2.34), 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 Eq. (2.35) to find $j_{l_{\max}-2}(x)$, $j_{l_{\max}-3}(x)$,..., $j_1(x)$, and $j_0(x)$. Then we *rescale* all of the values to the known value of $j_0(x)$ from Eq. (2.31) 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: Because of finite numerical precision, the computed value $j_l^{(c)}$ will become a mixture j_l and some n_l, e.g., $j_l^{(c)}(x) = j_l(x) + \epsilon n_l(x)$. What are the implications for each of the recursion directions?]

h. References

[1] R.H. Landau and M.J. Paez, *Computational Physics: Problem Solving with Computers* (Wiley-Interscience, 1997). [See the 6810 info webpage for details on the updated eTextbook version.]

[2] M. Hjorth-Jensen, *Computational Physics* (2015). These are notes from a course offered at the University of Oslo. See the 6810 webpage for links to excerpts.

[3] W. Press *et al.*, *Numerical Recipes in C++*, 3rd ed. (Cambridge, 2007). Chapters from the 2nd edition are available online from `http://www.nrbook.com/a/`. There are also Fortran versions.