3. 6810 Session 3

a. Follow-ups to Session 2

Here are some brief comments on various things encountered in Session 2:

- Problem 2 in Assignment #1 is to compare summing $1/n$ from 1 to N to summing it from N to 1. When trying to understand what happens, keep in mind the simple example from Session 2 of $1 + a + a + \cdots$ vs. $a + a + \cdots + 1$. The same principles apply.
- A quick word on using . and .. in Cygwin or Linux to refer to relative directories. A single period refers to the current directory while a double period refers to one directory up (the "parent" directory). Suppose that the 6810 directory has two sub-directories, session_01 and session_02. In session_01 there is the file make_area while you are currently working in the session_02 directory. The "." refers to session_02 while ".." refers to 6810. Here are some commands and outcomes (`ls` will **ls**t the files in the current directory):

`ls ..`	$\Longrightarrow$	`session_01 session_02`
`cp ../session_01/make_area .`	$\Longrightarrow$	copy to session_01, same name
`cp ../session_01/make_area make_bessel`	$\Longrightarrow$	copy with a new name

  Ok, here's a test. Where is `../session_01/../session_02`?
- When making a plot with gnuplot, it is a good idea to always `set timestamp`, which adds the current date and time to your plot. This lets you keep track when a plot was made and which of two plots is the later one.
- Recap: "power laws," which take the form of

$$z = Cw^{\alpha} \ , \tag{3.1}$$

  with C and α constants (and α is often *not* an integer) are common in physics. We'll see them frequently in looking at the errors of numerical calculations. Our simplest means of analysis is the log-log plot. Taking the logarithm of both sides of Eq. (3.1),

$$\log z = \log(Cw^{\alpha}) = \log C + \alpha \log w \ , \tag{3.2}$$

  so if we plot $y = \log z$ versus $x = \log w$, with $C' = \log C$, then we are plotting

$$y = C' + \alpha x \ , \tag{3.3}$$

  which is a straight line with slope α and intercept C'.

  > *Moral: If a quantity (such as an error) obeys a pure power law in some region, it will appear as a straight line in that region and the slope gives the power. (However, it is also easy to fool yourself if the region is not large enough.)*

  Note that we could either output $\log w$ and $\log z$ from our code and plot these with gnuplot on a regular linear plot, *or* output w and z and `set logscale` to have gnuplot take the logarithms.

- To call C routines from C++, we have to use `extern "C"` when specifying function prototypes for the C routines in the C++ code:

  ```
  extern "C" {
      <header stuff>
  }
  ```

  where `<header stuff>` for a single C function is just the usual prototype. (A function prototype is given at the top of the program or read in from a header file [.h file], and gives the name of the function, the number and types [e.g., int or double] of its variables and the type of its return value.) GSL routines automatically have this built into their header files, so we can call GSL routines from C++ simply by including the appropriate GSL header files.

- **Gotcha #1.** When adding the spherical Bessel function call from GSL, you might have gotten an error message that C++ didn't recognize the function. But there are two possible reasons for an error. If the error message was something like:

  ```
  error: ‘gsl_sf_bessel_jl’ undeclared
  ```

  then the problem is you didn't include the appropriate *header file*. That is, you need the line:

  ```
  #include <gsl/gsl_sf_bessel.h>
  ```

  at the top of your program. The header file contains a prototype of the desired function; the GSL documentation lists the appropriate file name at the beginning of each major section. However, if the error message was something like:

  ```
  undefined reference to ‘gsl_sf_bessel_jl’
  ```

  then the problem was not during *compiling* but during *linking*, when the GSL library code is combined with yours. You need to specify where this GSL code is, which is what the `-lgsl -lgslcblas` specification in the makefile (under `LDFLAGS`) does. (If you got this error, you probably didn't follow the instructions to copy a previous makefile!)

- We'll see many examples of makefiles as we proceed, such as the one that maintains multiple program files in Session 3. A few explanations to help make it easier to read them:

  - The `SHELL=/bin/sh` line at the very top signals that the makefile is a form of "shell script", which means it is a type of program.
  - Comments are indicated by `#`, so that anything after a `#` is ignored by make. The usual comments in our makefile describe how to use the makefile.
  - A variable assignment takes the form:

    ```
    VARIABLE= value
    ```

    where "value" is everything up to the end of the line. The line can be (and frequently is) continued with a \. (*Warning!* There must not be a space after the \.) So

    ```
    SRCS= \
    file1.cpp \
    file2.cpp
    ```

    is the same as

    ```
    SRCS= file1.cpp file2.cpp
    ```

    It is traditional but not required to put variables in uppercase (but all variable names are case sensitive; that means that `SRCS` and `srcs` are different variables).

– To substitute the value of a variable, use a dollar sign and parentheses: `$(VARIABLE)`

That's enough for now. In the future we'll describe more about how the makefile carries out its mission.

b. C++ Input/Output Formatting: Take 1

We've already encountered input and output to the screen in C++ using `cin` and `cout` in our first programs as well as output to a file. Here we'll elaborate a bit on their use. A good online introduction is Ref. [1].

The C++ system of input and output offers (at least) three advantages over that of C:

1. C++ has an unformatted mode (by default) for input and output. That is, we can use `cin` and `cout` without any specified formatting, as one has to do in C. This is useful for new users, for simple input/output, and for incremental code development (worry about the formatting later). This avoids a lot of bugs encountered using C's input/output functions, such as `scanf` and `printf`.
2. The C++ input/output operators can be "overloaded" to work with new data types defined in a program (e.g., a class). There is no way to do this in C with `printf` or `scanf`.
3. There is type checking in C++ output operations, unlike in C. In C, you can get bizarre output if the type of your variable and the format do not match [1], e.g.,
   ```
   printf("%d%s", "Hello", 27);
   ```

As we develop our computational physics codes, we'd like to control the format of the output, such as whether or not floating-point numbers are printed in scientific notation and how many digits are used. We can use "manipulators" to do this.

- The handout "Formatting with Manipulators" gives a summary with examples of the use of manipulators in C++.
- We can stick manipulators anywhere in an output "stream" between `<<`'s, but the placement can matter. They will only affect output following their appearance and in some cases only the next item in the stream.
- The most common manipulator is `endl`, which generates a carriage return. If you want to skip three lines:
  ```
  cout << endl << endl << endl;
  ```
  will do it. You get use of `endl` when you include iostream with `#include <iostream>`
- To have access to most of the other manipulators, include the iomanip header file with the statement `#include <iomanip>` along with the other include files. If you don't do this, the use of manipulators such as `setprecision` will give a warning that it is undeclared.
- Here are some basic examples. See the online handout for more options.
  ```
  cout << rel_error << endl;
  ```

$\Longrightarrow$ prints the value with fixed decimal point and 5 or 6 digits.

```
cout << scientific << rel_error << endl;
```

$\Longrightarrow$ now in scientific notation.

```
cout << scientific << setprecision(16) << rel_error << endl;
```

$\Longrightarrow$ scientific with 16 digits.

```
cout << fixed << setprecision(6) << setw(8) << rel_error << endl;
```

$\Longrightarrow$ fixed point with precision 6 and the width set to 8 to get output to line up.

We'll see more examples as we go. See the online handout on manipulators and Ref. [1] for more information.

Sending output to a file is the same as sending it to `cout`, except that we have to:

- Put `#include <fstream>` with the other include files (the "f" in "fstream" stands for "file").
- Associate the output file (let's call it `my_file.out`) with the name of a "stream" that substitutes for `cout` (let's call it `my_out`) using `ofstream` (the "of" is for "output file"):
  ```
  ofstream my_out ("my_file.out");
  ```
- We can use the same conventions and manipulators as when using `cout`, e.g.,
  ```
  my_out << "The relative error is " << scientific << rel_error << endl;
  ```

c. Numerical Differentiation [4, 2]

The problem that round-off error causes for numerical derivatives is easy to see. In any numerical approximation to a derivative, one is simulating something like

$$\lim_{h\to 0} \frac{f(x+h)-f(x)}{h} = f'(x) \ . \tag{3.4}$$

But on a computer, the difference $f(x+h)-f(x)$ *does not* keep decreasing with h as h decreases. Instead, it reaches a minimum approximately equal to the machine precision ϵ_m (or equals zero). Thus,

$$f'_c(x) \overset{h\to 0}{\longrightarrow} \frac{\epsilon_m}{h} \quad \Longrightarrow \quad \log_{10} f'_c(x) \approx -\log_{10} h + \log_{10} \epsilon_m \ . \tag{3.5}$$

What does this imply for a graph on a log-log plot?

c.1 Forward Difference

The easiest approximation to a numerical derivative is based on a simple Taylor expansion:

$$f(x_0+h) = f(x_0) + hf'(x_0) + \frac{h^2}{2} f^{(2)}(x_0) + \cdots \ , \tag{3.6}$$

and solving for $f'(x_0)$ (we'll use a subscript c for a computed expression):

$$\begin{aligned} f'_c(x_0) &\approx \frac{f(x_0+h)-f(x_0)}{h} \\ &\approx f'(x_0) + \frac{h}{2} f^{(2)}(x_0) + \cdots \end{aligned} \tag{3.7}$$

This is called the "forward-difference" derivative algorithm. Unless the second derivative $f^{(2)}(x_0)$ happens to vanish, the error is proportional to h. [Note: The assumption is that the function is smooth. This means we must be very careful when taking derivatives numerically of measured data that has noise because this assumption is likely violated.]

We can improve the approximation by reducing h, but only to the point where the round-off error from the subtractive cancellation gets to be the same size. Thus, the optimal choice for h is when the round-off error is about equal to the approximation error. Since

$$\epsilon_{\rm round-off} \approx \frac{\epsilon_m}{h} \quad \text{and} \quad \epsilon_{\rm approx}^{\rm forward} \approx \frac{f^{(2)}h}{2} \ , \tag{3.8}$$

the optimal h is about

$$h \approx \left(\frac{2\epsilon_m}{f^{(2)}}\right)^{1/2} \ . \tag{3.9}$$

How big is that? (Plug in some test cases.)

c.2 Central Difference

The forward-difference algorithm approximates the slope at x_0 by the slope of the line from x_0 to $x_0 + h$. We could imagine that a better line to use has x_0 in the middle rather than one of the endpoints. This is indeed the case. The "central-difference" derivative algorithm takes a half-step back and a half-step forward from x_0:

$$f_c'(x_0) \approx \frac{f(x_0 + h/2) - f(x_0 - h/2)}{h} \equiv D_c f(x, h) \ , \tag{3.10}$$

where D_c stands for central difference.

If we use the Taylor series for $f(x \pm h/2)$ in this formula, we find

$$f_c'(x_0) \approx f'(x_0) + \frac{h^2}{24} f^{(3)}(x_0) + \cdots \ , \tag{3.11}$$

where all the odd powers of h cancel. So we get an extra power of h for no extra computational cost (we still only evaluate the function twice)! This is the same type of computational gain we get from Simpson's rule (see below), which we'll see has an error that goes like $1/N^4$ rather than the naive expectation of $1/N^3$.

We can apply the same error analysis as with the forward-difference algorithm to find the optimal h. Since

$$\epsilon_{\rm round-off} \approx \frac{\epsilon_m}{h} \quad \text{and} \quad \epsilon_{\rm approx}^{\rm central} \approx \frac{f^{(3)}h^2}{24} \ , \tag{3.12}$$

the optimal h is about

$$h \approx \left(\frac{24\epsilon_m}{f^{(3)}}\right)^{1/3} \ . \tag{3.13}$$

Unless the derivatives are ill-behaved (so that $f^{(3)}$ is unexpectedly large compared to $f^{(2)}$), the central-difference algorithm will be far superior to the forward-difference algorithm. *Also, the optimal h is much larger for the central-difference algorithm!* We can do even better using Richardson extrapolation, which is discussed in the Session 4 notes.

d. Numerical Integration

Numerical integration (also called "numerical quadrature") is both a topic worth knowing about, because it shows up frequently in computational physics problems, and a good example of the interplay of approximation and round-off errors. Chapter 5 of the Hjorth-Jensen notes [2] (available from the 6810 webpage under "Supplementary Readings") and *Numerical Recipes* [3] have good introductions. Here we'll give a partial discussion based on Chapter 4 in the Landau-Paez text [4].

The basic idea [4] is that we approximate the integral of $f(x)$ from a to b as a weighted sum of N values of the integral at selected points x_i in the interval $[a, b]$:

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{N} f(x_i)\,w_i \ , \tag{3.14}$$

where the w_i are a set of "weights", which are particular to the integration rule. That is, different choices of the x_i and the w_i for a given N are what distinguish different rules. As $N \to \infty$, the sum formally converges to the value of the integral (but not on the computer!!). In general, the precision of the approximation increases with N until round-off errors set in (you'll explore this in Session 3). The best approximation to the integral will depend on details of the function $f(x)$; automated integration routines will often try different methods to find one that works well.

The simplest integration rules, which we start with, have evenly spaced intervals (these are called "Newton-Cotes methods"). If each interval is small and the function is smooth, the function should be well approximated by a polynomial, which we know how to integrate term-by-term. Thus, within each interval, the function is approximated by the first terms in a Taylor series expansion of f. If we use more terms, we should get a better approximation, unless the Taylor series expansion is not valid. The latter happens when we have singularities in the interval like $1/x$ or non-integral powers like $x^{1/2}$ for $x = 0$. (There is no problem for these outside of intervals containing $x = 0$.) We have to deal with these cases carefully, as we'll do later in Session 4.

The *trapezoid* rule approximates the function by a straight line in the interval while *Simpson's* rule approximates the function by a parabola (which should be more accurate). Check the references for the standard derivations of the rule and the approximations. Below we'll consider an alternative way to get these rules. The trapezoid rule is for $N - 1$ intervals (i.e., N points) of width

$$h = \frac{b-a}{N-1} \qquad \text{with} \qquad x_i = a + (i-1)h\,, \quad i = 1, \ldots, N \tag{3.15}$$

is

$$\int_a^b f(x)\,dx \approx \frac{h}{2} f_1 + h f_2 + h f_3 + \cdots + h f_{N-1} + \frac{h}{2} f_N \ , \tag{3.16}$$

so the weights are

$$w_i = \left\{ \frac{h}{2}, h, \cdots, h, \frac{h}{2} \right\} \ . \tag{3.17}$$

Integration with Simpson's rule is

$$\int_a^b f(x)\, dx \approx \frac{h}{3} f_1 + \frac{4h}{3} f_2 + \frac{2h}{3} f_3 + \frac{4h}{3} f_4 + \cdots + \frac{4h}{3} f_{N-1} + \frac{h}{3} f_N \ , \tag{3.18}$$

where N must be odd, so the weights are

$$w_i = \left\{ \frac{h}{3}, \frac{4h}{3}, \frac{2h}{3}, \frac{4h}{3}, \cdots, \frac{4h}{3}, \frac{h}{3} \right\} \ . \tag{3.19}$$

These weights can be specified by the "Elementary Weights" for each subinterval, as in this table:

Name	Degree	Elementary Weights	# of points ($i = 1, 2, \cdots$)
Trapezoid	1	$(h/2, h/2)$	$i + 1$
Simpson's	2	$(h/3, 4h/3, h/3)$	$2i + 1$
$\frac{3}{8}$	3	$(3h/8, 9h/8, 9h/8, 3h/8)$	$3i + 1$
Milne	4	$(14h/45, 64h/45, 24h/45, 64h/45, 14h/45)$	$4i + 1$

The full interval is formed by combining elementary intervals, overlapping the last and first points. (E.g., combine two Simpson's rule intervals to get $\{h/3, 4h/3, h/3 + h/3, 4h/3, h/3\}$, which requires the function to be evaluated at five separate points.) Note that the number of points that can be used varies with the rule (last column).

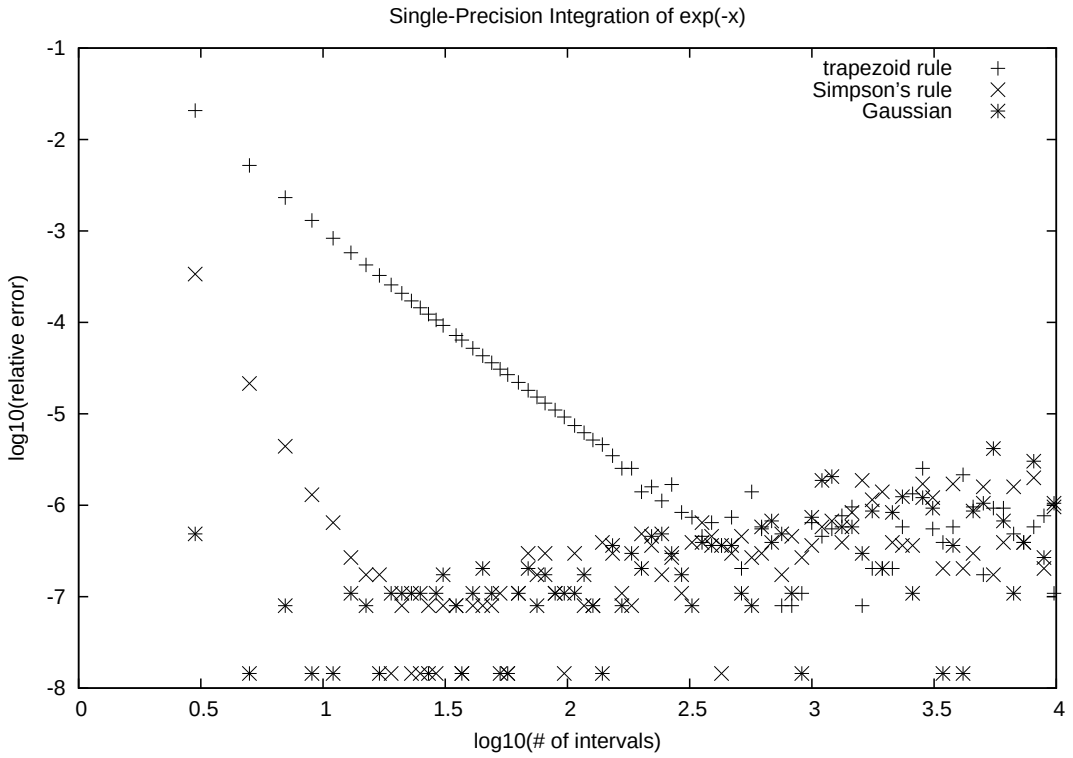
The approximation errors for each rule are estimated by expanding $f(x)$ in a Taylor series about the midpoint of each interval and then multiplying by the number of intervals to get the global errors. The error depends on N, for fixed a and b, as

$$\text{trapezoid} \implies \epsilon_{\text{approx}} \propto \frac{1}{N^2} \tag{3.20}$$

$$\text{Simpson's} \implies \epsilon_{\text{approx}} \propto \frac{1}{N^4} \ . \tag{3.21}$$

(The second result is puzzling at first; shouldn't it be $1/N^3$? How do we gain an extra power of $1/N$?) We'll test these results in Session 3 (see the figure below for sample results of the relative error vs. the number of points). As we did for derivatives, we find the total error by adding this approximation error to the round-off error. In the next section we discuss the round-off errors that apply for integrals, so that the total error is given by Eq. (3.35).

A useful way to think about the integration rules is in terms of what functions they can integrate exactly. The degree n rule in the table integrates exactly a polynomial in x with maximum power n (that is, the leading term is proportional to x^n). If it will integrate any polynomial exactly, then in particular it will integrate the $n + 1$ monomials $\{1, x, x^2, \cdots, x^{n-1}, x^n\}$ from -1 to $+1$. You should convince yourself that the converse is also true: if each of these is integrated exactly in this interval, then a general polynomial of degree n will be integrated exactly in an arbitrary interval.



[Hint: split up your general polynomial into separate monomials and make an appropriate change of variables.] This set of polynomials are convenient choices to generate the elementary weights for subintervals in the table, although any other linearly independent set and any other interval would suffice as well.

Let's try the first two cases. For the Trapezoid rule there are two points in the subinterval and therefore two weights to determine, w_{-1} and w_{+1}. The spacing is $h = 2$, which we can put back at the end. Evaluate the integrals of 1 and x exactly and with the rule, and set them equal:

$$\int_{-1}^{+1} dx\, 1 = 2 = (w_{-1} \times 1) + (w_{+1} \times 1) = w_{-1} + w_{+1} \ , \tag{3.22}$$

$$\int_{-1}^{+1} dx\, x = 0 = (w_{-1} \times -1) + (w_{+1} \times 1) = -w_{-1} + w_{+1} \ . \tag{3.23}$$

Solving these linear equations yields $w_{-1} = w_{+1} = 1$. Recalling that $h = 2$ here, the general result comes from multiplying each weight by $h/2$, so we get $(h/2, h/2)$, as in the table. For Simpson's rule, we have three points in the subinterval, so $h = 1$ and there are three weights. The integrals

are now:

$$\int_{-1}^{+1} dx\, 1 = 2 = (w_{-1} \times 1) + (w_0 \times 1) + (w_{+1} \times 1) = w_{-1} + w_0 + w_{+1} \ , \tag{3.24}$$

$$\int_{-1}^{+1} dx\, x = 0 = (w_{-1} \times -1) + (w_0 \times 0) + (w_{+1} \times 1) = -w_{-1} + w_{+1} \ , \tag{3.25}$$

$$\int_{-1}^{+1} dx\, x^2 = 2/3 = (w_{-1} \times 1) + (w_0 \times 0) + (w_{+1} \times 1) = w_{-1} + w_{+1} \ . \tag{3.26}$$

The last two equations yield $w_{-1} = w_{+1} = 1/3$ by inspection, which then implies $w_0 = 4$. With $h = 1$, we just multiply by h to get the result in the table. Try the last two yourself. [Hint: for the 3/8 rule, the four equal-spaced x points are -1, $-1/3$, $+1/3$, and $+1$. This means $h = 2/3$ so we'll multiply by $3h/2$ at the end. You might find it convenient to use the Mathematica `Solve` command to find the unique solution to the set of four linear equations.]

The other integration rule we'll use is Gaussian quadrature, which does not use equally spaced intervals. Rather, the N points x_i and weights w_i are chosen so that a polynomial of degree $(2N-1)$ times a specified function is integrated *exactly* over a corresponding interval. The specified function could be 1 with the interval $[-1, 1]$ (Gauss-Legendre quadrature), or e^{-x} (Gauss-Laguerre) with the interval $[0, \infty]$, or many other choices. By appropriate scaling, the Gaussian quadrature intervals can be transformed to $[a, b]$. These integration rules are amazingly effective for a wide range of integrands, as we'll see by example in Session 3 (and in the figure). If we compare Gauss-Legendre quadrature to the equal-space rules discussed above, we realize that it is the freedom to choose where the x_i points are, rather than requiring them to be equally spaced, that allows us to integrate a degree-$(2N-1)$ polynomial rather than merely a degree-$(N-1)$ polynomial.

Please see the Hjorth-Jensen notes (chapter 5) for more detail on the theory and practice of Gaussian quadrature. Being able to use it is an indispensable tool in your computational physics toolkit. Question: If you want to apply Gaussian quadrature with 40 points total, is it better to use a 40-point rule over the entire interval, or to subdivide the interval and apply smaller rules to each subinterval (e.g., two 20-point rules or four 10-point rules)? Keep in mind that smooth functions like exponentials look like polynomials over small intervals, but not generally over large intervals.

e. Accumulation of Multiplicative Errors and Random Walks

We saw in Session 2 how errors combine when floating-point numbers are added or subtracted. What about when they are multiplied? Suppose z_1 and z_2 are multiplied to give z_3. How does the round-off error in z_3 relate to the errors in z_1 and z_2? Write it out using $z_c = z(1+\epsilon)$ with $|\epsilon| \le \epsilon_m$ (recall that ϵ_m is the machine precision and z_c means the computer representation of z):

$$\begin{aligned} z_{3_c} &= z_3(1+\epsilon_3) \\ &= z_1(1+\epsilon_1) \times z_2(1+\epsilon_2) \\ &\doteq z_1 z_2(1+\epsilon_1+\epsilon_2) \\ &\doteq z_3(1+\epsilon_1+\epsilon_2) \ , \end{aligned} \tag{3.27}$$

where we've dropped $\epsilon_1\epsilon_2$ because it is much smaller than either ϵ alone. Thus we find that:

$$\epsilon_3 \approx \epsilon_1 + \epsilon_2 \ , \tag{3.28}$$

so when we multiply (or divide, you check!), we add the round-off errors.

What is the implication of adding many round-off errors (for example, in doing a numerical integration)? It is often (usually?) the case that the errors are *uncorrelated*; that is, ϵ_1 might be anywhere in the interval $-\epsilon_m \le \epsilon_1 \le \epsilon_m$ and ϵ_2 is in the same interval but is "random" compared to ϵ_1. Let's assume that the distribution is random (the BONUS problem in problem set #1 explores the actual distribution for a test case). What do we expect to find as a total error after N multiplications?

Let s_i be a random number between -1 and 1, that is,

$$-1 \le s \le 1 \ . \tag{3.29}$$

Then we can say that

$$\epsilon_i = s_i \epsilon_m \ . \tag{3.30}$$

Now s_i should be symmetrically distributed and the width of the distribution should be about 1 (up to factors of order 2), so the mean of s_i and s_i^2 are (here $\langle O \rangle$ means the average of O):

$$\langle s_i \rangle = 0 \qquad \text{and} \qquad \langle s_i^2 \rangle \approx 1 \ . \tag{3.31}$$

On the other hand, s_i and s_j are uncorrelated, so

$$\langle s_i s_j \rangle = 0 \ . \tag{3.32}$$

This means that N multiplications is like a random walk with N steps. We can find the magnitude of the total error ϵ_{total} by considering

$$\begin{aligned} \epsilon_{\text{total}}^2 &= \epsilon_m^2 (s_1 + s_2 + \cdots + s_N)^2 \\ &= \epsilon_m^2 (s_1^2 + s_2^2 + \cdots + s_N^2 + 2s_1 s_2 + 2s_1 s_3 + \cdots) \\ &\doteq \epsilon_m^2 (N\langle s^2 \rangle + 2N\langle s_i s_j \rangle) \\ &\doteq N\epsilon_m^2 \ . \end{aligned} \tag{3.33}$$

(Do you see how the third line follows from the second, since we have effectively chosen N random numbers?) Thus, the total error goes like the square root of the number of operations:

$$\epsilon_{\text{total}} \approx \sqrt{N}\epsilon_m \ , \tag{3.34}$$

which is characteristic of a random walk.

Bottom line: If we're in a regime where the error is dominated by round-off error rather than the approximation error, we should see the error increase as the square root of the number of operations. This tells us, for example, that we can't indefinitely improve the result of a numerical integration by making the sub-intervals smaller and smaller (the number of operations with scale

as the number of sub-intervals). If we have an approximation error that goes like a power of the number of sub-intervals N, $\epsilon_{\rm approx} \approx \alpha/N^\beta$, with $\beta > 0$, then the total approximation plus round-off error will go like

$$\epsilon_{\rm total} \approx \frac{\alpha}{N^\beta} + \sqrt{N}\epsilon_m \ . \tag{3.35}$$

What is the implication for the best N to choose? Answer: This occurs roughly when the two errors are equal. For $\beta = 2$ and $\alpha \approx 1$, this implies $N_{\rm best} \approx 1/(\epsilon_m)^{2/5}$.

f. Pointers to Functions

Here we'll make our first (but not last) discussion of "pointers" in C++. Pointers are often a mystery even to those who use C or C++ frequently, and are usually completely baffling (at first) to those who grew up using Fortran. We'll need to revisit the use of pointers soon to take advantage of the GSL routines. In Session 3, we introduce pointers to functions, which solve the problem of how to pass a function to a subroutine. In our particular case, we want to tell a routine that does a numerical integration what integrand it should integrate.

You should note that the original "solution" to this problem in the integration program adapted from integ.c in the Landau–Paez book [4] was to give a name to the function, e.g., "f", in the integration subroutines and then use the *same* global name "f" to define the function in the calling program. This is not a good solution in general. For example, what if we needed to integrate more than one function? More generally, we would like a wall between the subroutine function and the calling function (a form of "encapsulation"), and all we do is pass things through the wall, without knowing precisely what is happening on the other side.

As an analogy to pointers that is now familiar to most of us, think about web pages and URL web addresses. The URL `http://physics.osu.edu/` is clearly different from the content of the OSU Physics homepage. If we wanted to ask someone by email to look at something (e.g., a picture) on that page, we could do it two ways:

i) send a copy of the content of the homepage, or

ii) send them the URL.

These two ways of passing information correspond in C++ to "passing by value" and "passing by reference". The analog of the URL is the "address" of a variable or a function. The special characters `*` and `&` are used in connections with pointers. In future sessions we'll see detailed examples of their use.

For now, let's just see the how pointers to functions are used in practice. We'll take the `trapezoid_rule` function in `integ_routines.cpp` as an example. In the `integ_test.cpp` program, a function called `my_integrand` is defined with an example integrand:

```
float my_integrand (float x)
{
  return (exp (-x));
```

```
}
```

To call `trapezoid_rule` with `my_integrand` as an argument:

```
result = trapezoid_rule (i, lower, upper, &my_integrand);
```

Note the ampersand & in front of `my_integrand`. That sends to trapezoid the *address* of the `my_integrand` function (i.e., it sends the location in the computer memory).

The function `trapezoid_rule` is defined as:

```
float trapezoid_rule ( int num_pts, float x_min, float x_max,
    float (*integrand) (float x) )
```

The `*` indicates a *pointer*, which holds the address that is passed in this case. For now, just note that the function

`float integrand (float x)`

becomes

`float (*integrand) (float x)`

in the function definition. (For the experts: We don't need to specify "x" in the last argument; it is a dummy variable and can be omitted entirely.)

g. References

[1] "Basic Input/Output" at http://www.cplusplus.com/doc/tutorial/basic_io/ gives a brief introduction, "C++ Notes: I/O Manipulators" gives a summary and examples at http://www.fredosaurus.com/notes-cpp/io/omanipulators.html and "C++ Input/Output: Streams" at http://courses.cs.vt.edu/~cs1044/Notes/C04.IO.pdf will step you through all the background and details.

[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.

[4] 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.]