

4. 780.20 Session 4

a. Brief Follow-ups to Session 3 (and earlier)

- **Comparing floating point numbers.** We saw in Session 3 the perils of comparing floats or doubles using `if (x1 == x2) { <something> }`. What we really want to do is check whether they are close enough to each other, which might mean within a factor of 2 of the machine precision. So we define `epsilon` to be $2\epsilon_m$, and make the comparison: `if (fabs(x1-x2) < epsilon) { <something> }`. Note the use of `fabs`, since we don't know the signs of `x1` and `x2` or which is larger.
- **Number-one cause of “syntax errors”.** If the compiler reports a “syntax error” at a particular line number, check first that you haven't omitted a semicolon at the end of the previous line.
- **Output to a file in C++.** 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`:


```
ofstream my_out ("my_file.out");
```

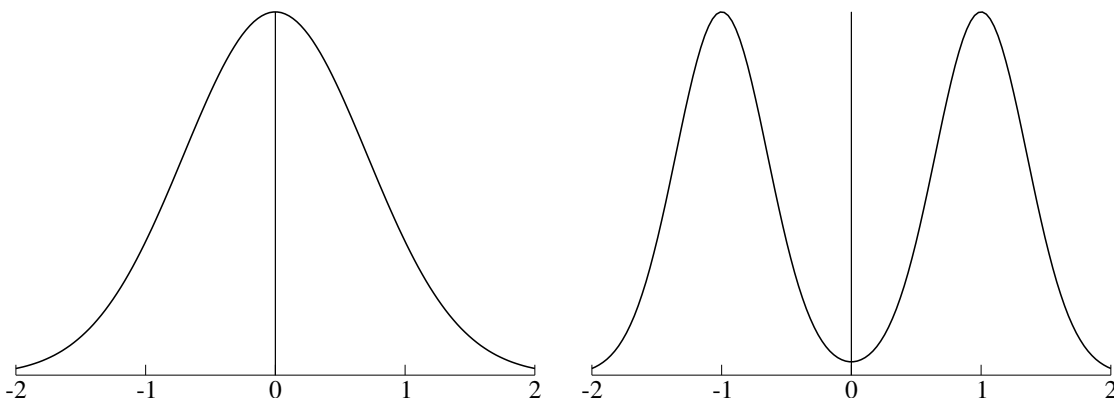
Then we can use the same conventions and manipulators as when using `cout`, e.g.,

```
my_out << "The relative error is " << scientific << rel_error << endl;
```

- **Distribution of round-off errors.** In order to discuss the accumulation of round-off errors, we've assumed that they are distributed randomly. More precisely, we've assumed that if we have a series of N operations with numbers z having round-off errors ϵ given by

$$z_c = z(1 + \epsilon) \quad \text{where} \quad |\epsilon| \leq \epsilon_m, \quad (4.1)$$

then the ϵ 's are symmetrically distributed about $\epsilon = 0$ and are *uncorrelated*. But we could imagine different distributions for ϵ/ϵ_m , for example, either of the two pictured here:



Which, if either, is the actual distribution like? This is one of the goals of the bonus problem for Assignment 1. Results of one investigation will be given in Session 5.

- **Type casting in C++.** Here are four ways to can convert between types (e.g., changing an `int` to a `double`) in C++:

```
int my_int = 7;
double my_double1 = my_int;           // implicitly convert by coercion
double my_double2 = (double)my_int;   // C-like prefix notation
double my_double3 = double(my_int);   // C++ functional notation
double my_double4 = static_cast<double>(my_int); // C++ static_cast
```

Avoid using “coercion”, which can get you in trouble. I prefer the C++ functional approach for most ordinary situations (like the one here). Also, *always* use decimal points for floating-point constants: $1./3$. rather than $1/3$ or $1./x$ rather than $1/x$.

b. Richardson Extrapolation and Romberg Integration

If we know the *form* of the error as a function of the mesh spacing h for an algorithm, we can systematically eliminate the error up to some power of h by using the rules for different h 's. Consider numerical differentiation, with the symmetric rule evaluated at h and then $h/2$:

$$\begin{aligned} D_c f(x_0, h) &\equiv \frac{f(x_0 + h/2) - f(x_0 - h/2)}{h} \\ &= f'(x_0) + \frac{h^2}{24} f^{(3)}(x_0) + \mathcal{O}(h^4) \end{aligned} \quad (4.2)$$

$$\begin{aligned} D_c f(x_0, h/2) &\equiv \frac{f(x_0 + h/4) - f(x_0 - h/4)}{h/2} \\ &= f'(x_0) + \frac{h^2}{4 \cdot 24} f^{(3)}(x_0) + \mathcal{O}(h^4), \end{aligned} \quad (4.3)$$

where we've used the Taylor expansion of $f(x)$ about x_0 ,

$$f(x_0 + \Delta x) = f(x_0) + \Delta x f'(x_0) + \frac{1}{2}(\Delta x)^2 f''(x_0) + \frac{1}{6}(\Delta x)^3 f^{(3)}(x_0) \cdots \quad (4.4)$$

repeatedly with $\Delta x = \pm h/2$ or $\pm h/4$. The derivative we are looking for, $f'(x_0)$, appears in both equations, but note that the leading correction term in the first is precisely 4 times the correction term in the second. Thus, we can eliminate the h^2 error to get an improved formula:

$$f'(x_0) \approx \frac{4D_c f(x_0, h/2) - D_c f(x_0, h)}{3} + \mathcal{O}(h^4) \quad (4.5)$$

with an error proportional to h^4 .

This approach can be repeated (and you'll have a chance to do so!). It is called “Richardson extrapolation” in general. It can be applied to any numerical calculation where the (leading)

approximation error is known (note that we need to know not only that the error scales with a definite power of h or whatever, but that the coefficient is the same for different h).

When applied to integration, Richardson extrapolation is called Romberg integration. Consider the trapezoid rule applied to

$$I \equiv \int_{x_0}^{x_0+2h} f(x) dx \quad (4.6)$$

in two ways (here f_i denotes $f(x_0 + i \times h)$):

i) one step of width $2h$:

$$I_T(2h) = \frac{2h}{2}(f_0 + f_2) = I + \frac{8h^3}{12}f''(x_0 + h) + \mathcal{O}(h^5), \quad (4.7)$$

ii) two steps of width h :

$$\begin{aligned} I_T(h) &= \left\{ \int_{x_0}^{x_0+h} + \int_{x_0+h}^{x_0+2h} \right\} f(x) dx = \frac{h}{2}(f_0 + f_1 + f_1 + f_2) \\ &= I + \frac{h^3}{12} \left[f''(x_0 + \frac{h}{2}) + f''(x_0 + \frac{3h}{2}) \right] = I + \frac{2h^3}{12}f''(x_0 + h) + \mathcal{O}(h^5). \end{aligned} \quad (4.8)$$

Combining these, we find that we can eliminate the $\mathcal{O}(h^3)$ error:

$$4I_T(h) - I_T(2h) = 3I + \mathcal{O}(h^5), \quad (4.9)$$

or

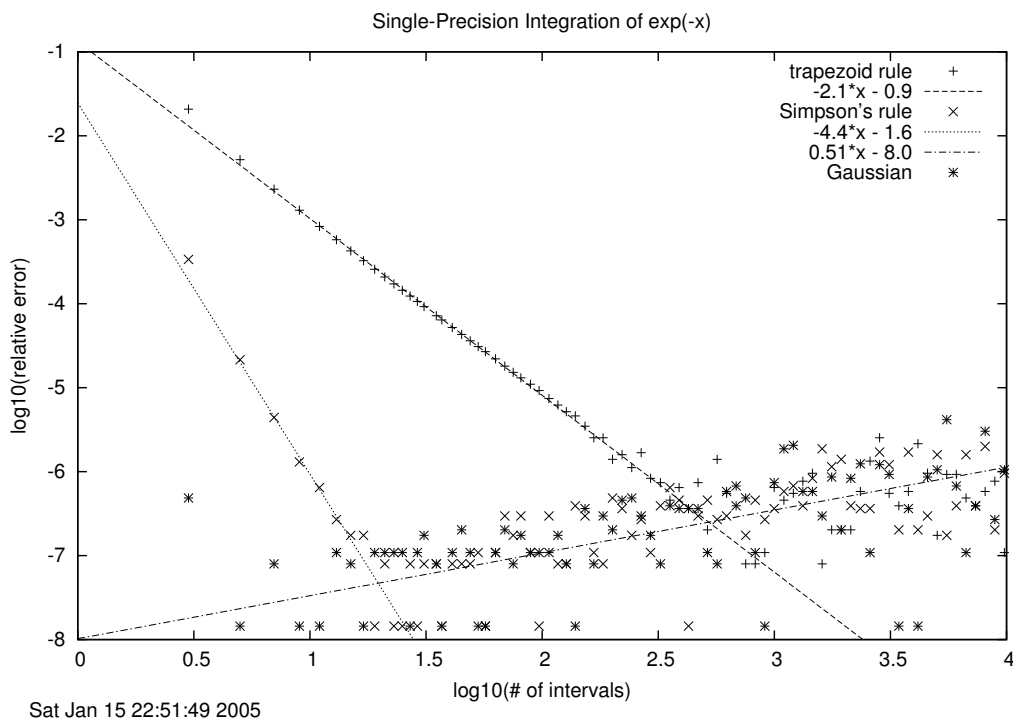
$$I \approx \frac{4I_T(h) - I_T(2h)}{3} = \frac{h}{3}(2f_0 + 4f_1 + 2f_2 - f_0 - f_2) = \frac{h}{3}(f_0 + 4f_1 + f_2) + \mathcal{O}(h^5), \quad (4.10)$$

which is Simpson's rule!

c. Local and Global Approximation Errors for Integration [1]

The underlying idea of the Newton-Cotes integration rules is that a smooth function looks locally like a polynomial. More precisely, Taylor's theorem says that any function without singularities in a small interval is well approximated in that interval by an n^{th} -degree polynomial, and the deviation from the true function is $\Delta f \sim h^{n+1}f^{(n+1)}(\xi)$, where $f^{(n+1)}$ is the $(n+1)^{\text{th}}$ derivative of f and ξ is in the interval. So the error is locally (i.e., for an interval of width h) expected to go like $h\Delta f \sim h^{n+2}f^{(n+1)}(\xi)$ and globally (i.e., for the entire integral) by $N \cdot h\Delta f \sim Nh^{n+2}\langle f^{(n+1)} \rangle$ (where the $\langle \rangle$'s mean some kind of average, whose details we'll not specify). The *relative* error is then $\sim Nh^{n+2}\langle f^{(n+1)} \rangle / f$ and since $h = (b-a)/N$, it goes like $N^{-(n+1)}\langle f^{(n+1)} \rangle / f$. If this analysis is correct, the error for the trapezoid rule should go like $1/N^2$, Simpson's rule like $1/N^3$, and $3/8$ rule like $1/N^4$.

Here's a log-log plot of trapezoid and Simpson's (and Gauss), extended somewhat from the range in Session 3, and with fits included (note that the titles give the equations of the fit lines).



You should have found that the 3/8 rule had about the same slope as Simpson's. [Note: You'll find slopes more accurately in double precision.] So trapezoid and 3/8 scale with N as expected, but Simpson's rule was *better* than expected. The reason is very similar to how the "central difference" approximation to numerical derivatives is better than "forward difference".

Note that a line fit to the round-off error region of the Simpson's rule points yields a slope close to $1/2$. Is this consistent with the discussion of the accumulation of round-off errors from the Session 3 notes?

d. Empirical Error Analysis

This discussion is based on Section 3.13 of the Landau-Paez text [2].

In making error plots so far, we have relied on the fact that we knew the exact answer, so calculating relative errors was easy. This is fine for studying the approximation and round-off errors for a given algorithm, but not very useful if we want to know about the error from a calculation for which we don't have the exact answer. After all, if we had the answer, we wouldn't need to do the calculation! Here's what we can do.

Consider a calculation with N steps. From the discussion of round-off errors in the Session 3 notes, we expect a total round-off error $\epsilon_{\text{round-off}}$ to be proportional to the square-root of N and the machine precision:

$$\epsilon_{\text{round-off}} \approx \sqrt{N} \epsilon_m . \quad (4.11)$$

[Note: The actual number of calculational steps to be used in calculating the round-off error random walk may be a constant times N . This just shifts by a small constant the region in the log-log error plot where the round-off error takes over.] The accumulated round-off will therefore dominate the error when N is sufficiently large (since approximation errors go like inverse powers of N).

Conversely, for smaller N , the error should be dominated by the approximation error, which we will assume is a power law. [Note: The approximation error may be a sum of powers; we assume that the coefficients are such that one term dominates.] Suppose that with N points or steps your method predicts $A_{\text{approx}}(N)$ and the exact answer is A_{exact} . If N is small enough that round-off errors are small, then

$$A_{\text{approx}}(N) \approx A_{\text{exact}} + \frac{\alpha}{N^\beta} \quad (4.12)$$

for α, β independent of N . Now calculate the answer again for a large number of points, say $2N$:

$$A_{\text{approx}}(2N) \approx A_{\text{exact}} + \frac{\alpha}{(2N)^\beta} . \quad (4.13)$$

Combining these results,

$$\frac{A_{\text{approx}}(N) - A_{\text{approx}}(2N)}{A_{\text{approx}}(2N)} \approx \frac{\alpha}{A_{\text{exact}}} \frac{1}{N^\beta} , \quad (4.14)$$

where we've neglected $1/(2N)^\beta$ compared to $1/N^\beta$. This is the secret: *The error at $2N$ is small compared to the error at N , so we can treat the $2N$ result as the exact answer.* A log-log plot of Eq. (4.14) reveals the power law exponent β . Note that you don't have to use $2N$ for the comparison; comparing N and $2N + 3$ or N and $3N$, for example, works just as well.

e. Spherical Bessel Function Recap (Session 2 and Assignment 1)

The two gnuplot graphs on the next page were made from an output file generated from the revised program `bessel_final.cpp`, which is included with the Session 3 programs (in `session03.tarz`). What do we learn from the plot of $j_{l=10}(x)$ vs. x on a log-log plot?

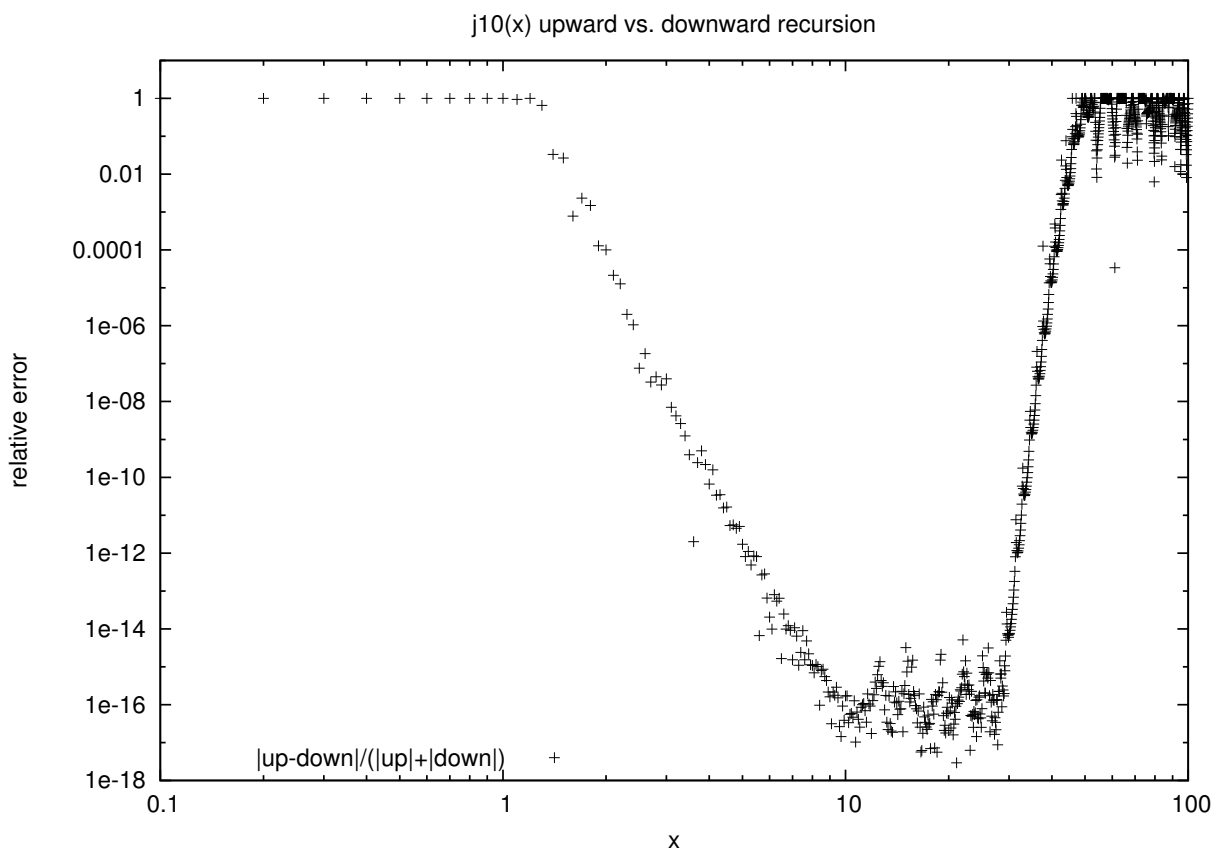
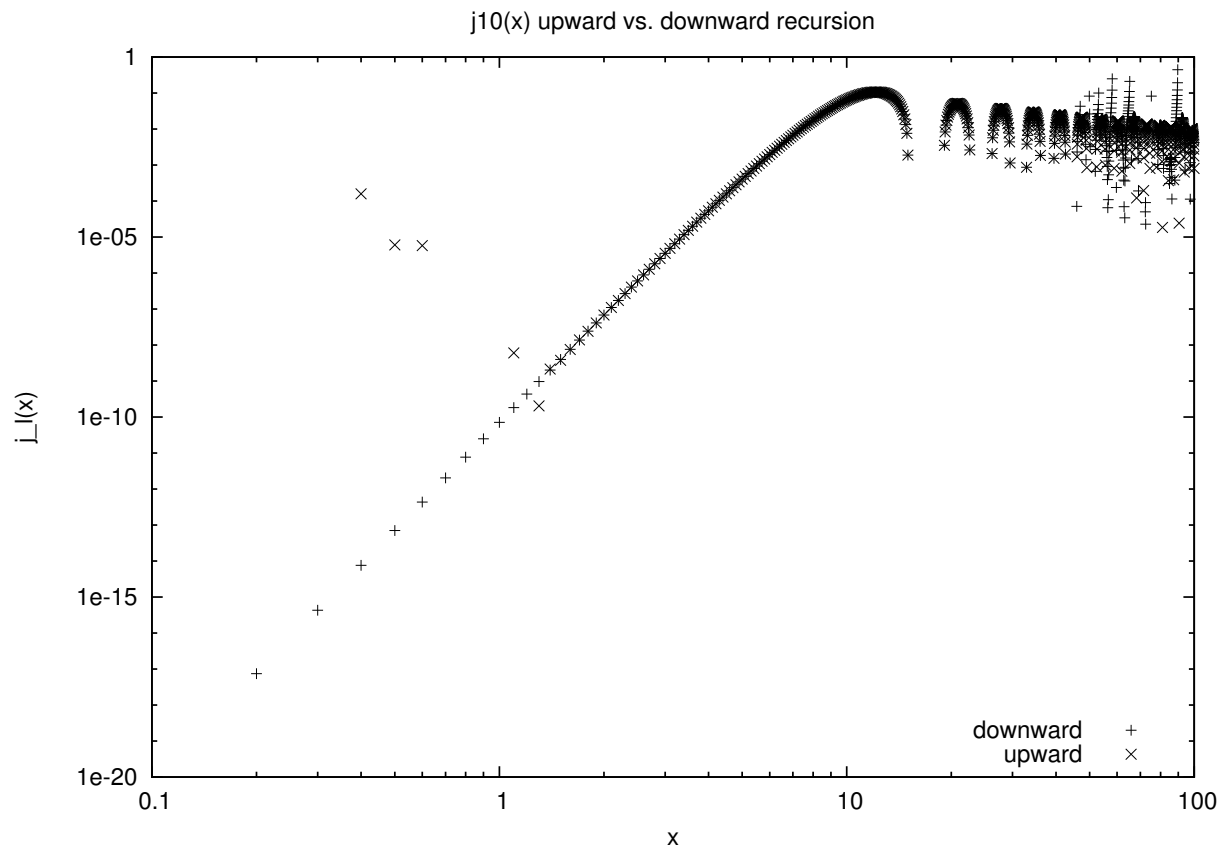
- The results for up and down recursion are clearly different for $x \leq 1$ and the downward ones are more plausible because they vary smoothly.
- We can be more precise by looking up the formula for spherical Bessel functions for small values of x :

$$j_l(x) \longrightarrow \frac{x^l}{(2l+1)!!} \quad \text{for } x \ll l \quad \text{where} \quad (2l+1)!! \equiv 1 \cdot 3 \cdot 5 \cdots (2l+1) , \quad (4.15)$$

so if we take the \log_{10} of both sides:

$$\log_{10}[j_l(x)] \approx \log_{10} \left[\frac{x^l}{(2l+1)!!} \right] = l \log_{10} x - \log_{10}[(2l+1)!!] , \quad (4.16)$$

the region $x \ll 10$ on a log-log plot should be a straight line with slope $l = 10$. That agrees with the downward recursion result.



- The up and down results differ again for $x \geq 40$ or so, but in between they look quite similar.
- The asymptotic form of $j_l(x)$ for large l is

$$j_l(x) \propto \frac{\sin(x - l\pi/2)}{x} \quad \text{for } x \gg l. \quad (4.17)$$

Can you think of how to use this result to check the numerical calculations?

What do we learn from the relative-error plot?

- The relative error plotted here is bounded, that is,

$$\frac{|\text{up} - \text{down}|}{|\text{up}| + |\text{down}|} \leq 1 \quad (4.18)$$

always, with it close to one if $|\text{up}| \gg |\text{down}|$ or $|\text{up}| \ll |\text{down}|$. Either case means *at least one* of the values is very wrong. In the small x region, we see that the upward recursion must be totally off (since we decided that downward was better).

- When $x \approx 10$ (which is $x \approx l$), the relative error is roughly machine precision (for doubles) with random coefficients. This implies that both work.
- When $x > 50$, *at least one* is bad again. How can we determine which? One way is to compare a test case to an accurate result from some other source. (Answer: “up” is good in this region.)
- What about the transition regions? We have roughly straight line behavior on a log-log plot. Recall that a straight line with slope a and intercept b means

$$\begin{aligned} (\log_{10} y) &= a(\log_{10} x) + b \\ &= \log_{10} x^a + \log_{10} 10^b \end{aligned} \quad (4.19)$$

$$= \log_{10}(10^b x^a), \quad (4.20)$$

so the underlying relationship between y and x is

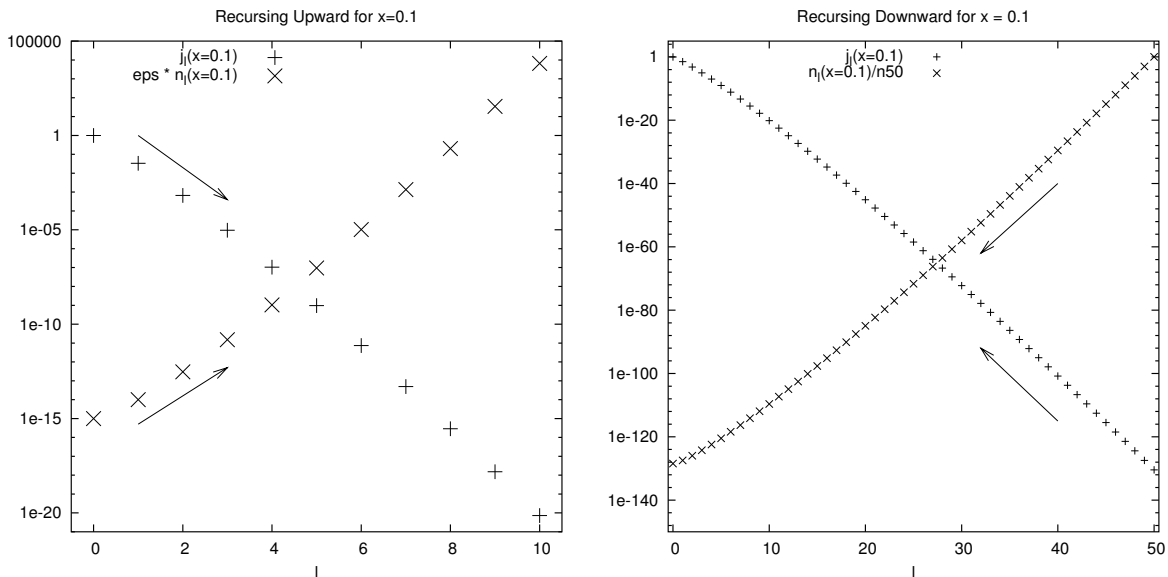
$$y = (10^b)x^a, \quad (4.21)$$

which is “power-law” behavior. From the graph here, the error goes roughly like x^{-20} from $x = 1$ to $x = 10$ (hard to tell more precisely by eye), which is a rather large power!

- The Mathematica notebook `bessel_check.nb`, included in `session04.zip`, includes definitions of the spherical Bessel functions, how to print out lots of digits, and some log-log plots.
 - One plot shows that $n_l(x) \gg j_l(x)$ for $x < 5$. The general expansions for small x are

$$j_{10}(x) = \frac{x^{10}}{13749310575} - \frac{x^{12}}{632468286450} + \dots \quad (4.22)$$

$$n_{10}(x) = -\frac{654729075}{x^{11}} - \frac{34459425}{2x^9} + \dots \quad (4.23)$$



- So suppose there are round-off errors in j_0 or j_1 . We'll use a superscript (c) to indicate the computer representation. From round-off, $j_0^{(c)}(x) \neq j_0(x)$ at fixed x , but the difference can be written as a small number (ϵ_x) times $n_0(x)$. That is because these are two independent solutions to the recurrence relations, so they are preserved in the upward recursion. Thus, at each x

$$j_0^{(c)}(x) = j_0(x) + \epsilon_x n_0(x) \quad (4.24)$$

Consider upward recursion in double precision from this starting point, as seen in the figure on the left on the next page for $x = 0.1$. [Note: $|n_l(x)|$ is plotted.]

- By the time we get to $l = 5$, the spurious n_l solution has passed the desired j_l solution, then leaves it in the dust by $l = 10$. The ratio of these Bessel functions at $l = 10$ in the transition region $1 \leq x \leq 10$ gives us the approximate error we can expect:

$$\frac{n_l(x)}{j_l(x)} \propto \frac{1/x^{l+1}}{x^l} = \frac{1}{x^{2l+1}} \xrightarrow{l=10} x^{-21} \quad (4.25)$$

which explains the downward, almost linear, slope in the graph of the relative error between 1 and 10.

- In summary, starting with a very small error at the beginning, of order the machine precision, by the time we get to $l = 10$, the result is completely dominated by the $n_l(x)$ part. How do we do better? Apply the recurrence relation in the other direction. Then the relevant graph is the one on the right. Even a bad initial guess will be of the form $a j_{50}(x) + b n_{50}(x)$, and the $n_l(x)$ part will be negligible rapidly! When we get down to $l = 0$, we determine a by comparing to the known value of $j_0(x)$.

This type of thing also happens with differential equations (later!).

f. C/C++ “Review” of Structures and Pointers

f.1 Structures

Suppose you are doing calculations involving a quadratic equation. It is specified by the three coefficients a , b , and c :

$$ax^2 + bx + c = 0 . \quad (4.26)$$

It is very useful to keep these quantities grouped together. We can do this in C/C++ with a *structure*:

```
struct coefficients {
    double a;
    double b;
    double c;
};
```

This defines the form of the structure. Then we can make an *instance* called `my_coefficients`:

```
struct coefficients my_coefficients;
```

(this is like defining a double by `double my_double;`). An equivalent construction, which defines the names `quad_parameters` is:

```
typedef struct {
    double a;
    double b;
    double c;
} quad_parameters;
```

Then we'd make an instance with:

```
quad_parameters my_coefficients
```

You can put whatever you want in the structure: double's, float's, int's, other structures. How do we get at the a , b , or c parts? With the “dot” notation:

```
my_coefficients.a = 3.0;    // set a=3.
alpha = my_coefficients.b; // set alpha to the value of b
```

Structures are a useful concept, but would be even better if we could package together *functions* with the data as well. This leads to the idea of a *class* in C++. Stay tuned for further developments!

f.2 Pointers and Void Pointers

See the brief handout on pointers from “Practical C++” and look at the `pointer_test.cpp` code. Here we add a bit to our previous discussion of pointers to functions. When you define a double named `alpha` by:

```
double alpha = 5.3;
```

the value 5.3 in binary floating point is stored in 64 bits (8 bytes) in a definite area of the computer memory. If the program is to do something with `alpha`, we can refer either to its value (at the moment), which is 5.3, *or* the starting address where it is stored, along with the knowledge that it is stored as a double. The latter approach uses a *pointer* to `alpha`, which has the address of `alpha` and which is designated a pointer to a double. Here's an example:

```
double alpha, gamma; // ordinary doubles
double * beta_ptr; // The * in the definition says that beta_ptr
// is a pointer. The _ptr suffix is just a mnemonic.

alpha = 3.;
beta_ptr = &alpha; // & means "address of", both sides are addresses
gamma = *beta_ptr + 2.1; // * is the "dereferencing" operator, it means
// look up the value at the address beta_ptr
```

So what is `gamma`? (Answer: 5.1)

If a function has a parameter such as α or β , then when we define

```
double funct (double x)           double funct (double x)
{                                  {
    return exp(-alpha*x);         or    return alpha * pow(x,beta);
}                                  }
```

how do we change the values of α and β ? This is the problem that GSL faces, since the number of passed parameters is not fixed. The (less than ideal) solution: use a *void pointer*.

As a pointer, a void pointer points to an address, but we don't specify whether it points to a double, an int, a structure, or whatever until it is used. Suppose we have the double `alpha` that we want to pass. We set up `params_ptr` as:

```
double alpha;
void *params_ptr;
params_ptr = &alpha; // the value of the pointer is the address (&)
```

In the function to which we pass `params_ptr`, we might think that we recover `alpha` by

```
alpha_passed = *params_ptr; // **INCORRECT**
```

but that is not enough: we must *type cast* `params_ptr`:

```
alpha_passed = *(double *)params_ptr; // **CORRECT**
```

Note that what we've done is inserted `(double *)` into our first attempt. In `pointer_test.cpp` you will find several other examples.

g. Hilbert Matrix

In Session 4, we do a warm-up problem with the GSL matrix diagonalization routines in preparation for diagonalizing Hamiltonians in Session 5. We'll use the "Hilbert matrix" as an example. It is

defined by

$$\mathbf{H}_{ij} \equiv \frac{1}{i+j-1} \quad (4.27)$$

or

$$\mathbf{H} = \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdots \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \cdots \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \cdots & \\ \vdots & \vdots & \vdots & & \end{pmatrix} \quad (4.28)$$

where the matrix terminates at size $N \times N$ for given N .

h. References

- [1] 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++.
- [2] R.H. Landau and M.J. Paez, *Computational Physics: Problem Solving with Computers* (Wiley-Interscience, 1997).
- [3] M. Hjørth-Jensen, *Computational Physics*. These are notes from a course offered at the University of Oslo. See the 780.20 webpage for links.