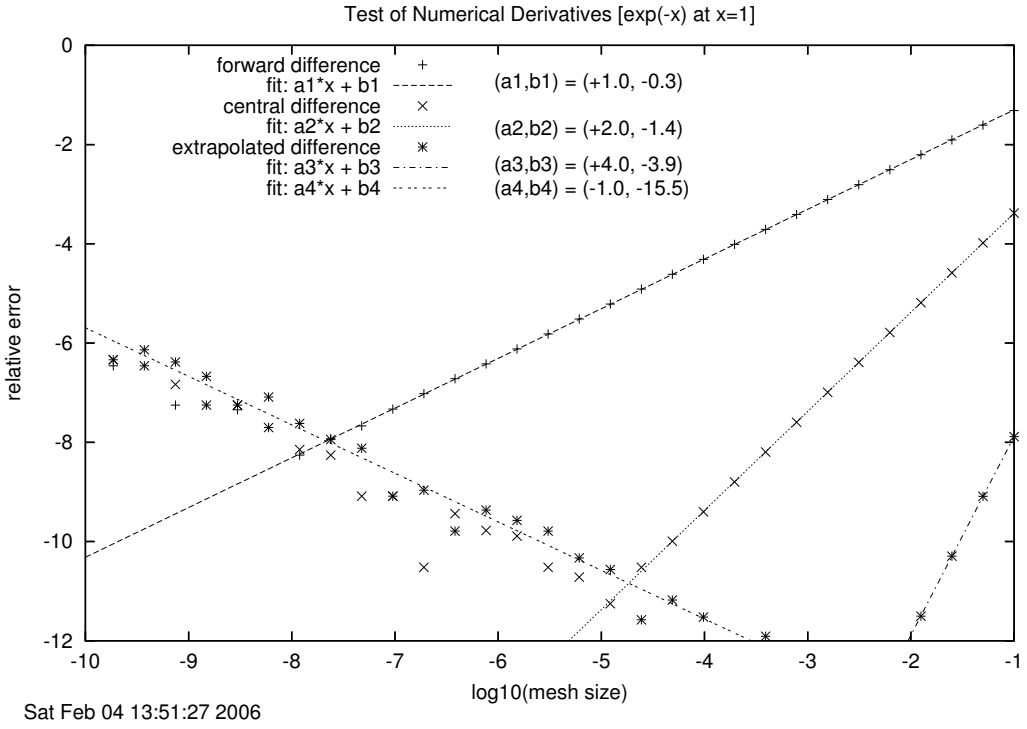
5. 6810 Session 5

a. Follow-up to Numerical Derivative Round-Off Errors

In Session 4, we looked at error plots for various ways to calculate numerical derivatives. Here is a plot showing results for forward difference, central difference, and a Richardson extrapolation of the central difference results:



Since we are plotting the logarithms of the errors, we have fit straight lines to the various plots and find that the algorithm errors are $\mathcal{O}(h)$, $\mathcal{O}(h^2)$, and $\mathcal{O}(h^4)$, respectively. We have also fit the region of round-off errors and find a slope of -1. Let's review why this happens.

Consider the forward difference case for simplicity. Recall from the Session 1 notes that any number z is related to its machine number computer representation z_c by

$$z_c = z(1+\epsilon) \quad \text{for some } \epsilon \text{ in the interval: } \quad -\epsilon_m \le \epsilon \le \epsilon_m \ , \tag{5.1}$$

where ϵ_m is the machine precision. Apply that to the forward difference formula as calculated on the computer:

$$\begin{aligned} \frac{f_c(x+h) - f_c(x)}{h} &= \frac{f(x+h)(1+\epsilon) - f(x)(1+\epsilon')}{h} \\ &\approx \frac{f(x+h) - f(x)}{h} + f(x)\frac{\epsilon - \epsilon'}{h} \\ &\approx \frac{df}{dx} + \mathcal{O}(h) + f(x)\frac{\epsilon - \epsilon'}{h} \ . \end{aligned} \tag{5.2}$$

Because the relative round-off errors ϵ and ϵ' are (apparently) randomly distributed from $-\epsilon_m$ to $+\epsilon_m$, they are unlikely to cancel closely, so in the second line we used that $f(x+h) \approx f(x)$. In the final line, we identify the algorithm error, which will contribute to the relative error a term proportional to h, and the net round-off error, which will contribute to the relative error the term $(\epsilon - \epsilon')/h$. This explains not only the slope of -1, but the intercept, which is of order $\log_{10} \epsilon_m$ (given that $f(x_0)$ is order unity). The same argument applies to the other derivative formulas, so the round-off error curve is universal.

b. Other Follow-ups to Session 4 (and earlier)

- **Adding Coefficients from Fits to a Plot**

  You may have noticed in the plot on the first page that values for the fit parameters determined by gnuplot are listed on the plot. These were not put in by hand but inserted automagically by using a `set label` command in the plot file. For example, the last two fits and the corresponding labels were made using:

```
f3(x) = a3*x + b3
fit [-2.:-1] f3(x) "derivative_test.dat" using ($1):($4) via a3,b3

f4(x) = a4*x + b4
fit [-10:-3] f4(x) "derivative_test.dat" using ($1):($4) via a4,b4

set label "(a3,b3) = (%-+4.1f", a3, ", %-+4.1f", b3, ")" at graph .44,.80
set label "(a4,b4) = (%-+4.1f", a4, ", %-+4.1f", b4, ")" at graph .44,.76
```

  The labels were positioned using `at graph` x,y, where the x, y coordinates are in the "graph" coordinate system that runs from $(0,0)$ in the lower left to $(1,1)$ in the upper right. (You can also use the plot coordinate system by omitting the word "graph".) The values of `a3`, `a4`, and so on were printed using C language-style format specifications, namely `%-+4.1f`. This makes the printed number take up four spaces, with one digit to the right of the fixed decimal point, left justified and with a + or − sign. (To find out more about such formatting, give Google the search string "man 3 printf" and look at the first return page.)

  An alternative is to define a "string" variable in gnuplot using the `sprintf` command and the same C-style formatting. This is illustrated in the handout on "Using a Plot File with Gnuplot" in the plot file `test1.plt`. The idea of `sprintf` is to print values into a string variable (see the line defining `slope_title` in that handout). For example,

```
forward_title = sprintf("forward slope = %-+4.1f",a1)
```

  which you can then use in a `plot` command as a `title`. For example:

```
plot "derivative_test.dat" using ($1):($2) title 'forward difference',\
  f1(x) title forward_title
```

- **Calculating relative errors.** If you have an approximate result `approx` and an exact result `exact`, you would typically calculate the relative error as

$$\text{relative error} = \left| \frac{\texttt{exact} - \texttt{approx}}{\texttt{exact}} \right| . \tag{5.3}$$

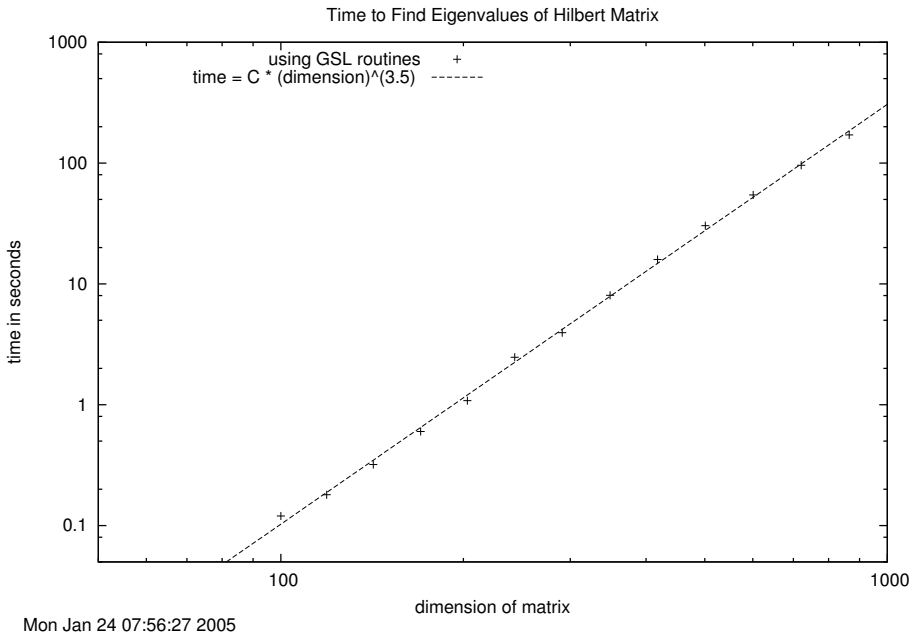
(Note that you take the absolute value of both numerator and denominator, so that the relative error is always positive, even if `exact` is negative.) But what if you have two answers and it is not clear which is better (or maybe that one result is better in one region and the the other result is better elsewhere). An additional problem is that if your result goes through zero, the error in Eq. (5.3) blows up, which can be misleading in a plot of the error. Both problem are fixed if you find the relative error between `result1` and `result2` from

$$\text{relative error} = 2\left|\frac{\texttt{result1} - \texttt{result2}}{\texttt{result1} + \texttt{result2}}\right| \ . \tag{5.4}$$

This uses the average of the two results for the denominator (note the factor of 2). If the two results are close in value, this will give effectively the same result as Eq. (5.3) and the graph of the error will be much better behaved if the results go through zero in different places. (You might even want the absolute value of each term in the denominator.)

- **Timing of matrix operations.**

  In Session 4, you were asked to check how the timing for a GSL eigenvalue routine scaled with the size N of the matrix (that is, how many seconds does an $N \times N$ matrix take). If you had time to finish this, you might have found a plot like this:



  As usual, to check scaling we use a log-log plot. The plot here was generated from a plot file that looked like:

```
# plot file for new eigen_test
set timestamp

set title 'Time to Find Eigenvalues of Hilbert Matrix'
set xlabel 'dimension of matrix'
```

```
set ylabel 'time in seconds'
set key left

set logscale
set xrange [50:1000]
set yrange [.05:1000]

f(x) = a*x + b
fit f(x) "eigen_test.dat" using (log10($1)):(log10($2)) via a,b

set term x11
plot "eigen_test.dat" using ($1):($2) title 'using GSL routines', \
   10**b * x**a title 'time = C * (dimension)^(3.5) '

set out "eigen_test_new.ps"
set term postscript
replot
```

Note how the fit and the plot of the fit are done using the $1 and $2 variables. The result of the fit was $a \approx 3.5$ and $b \approx -7.9$. Thus $\log(\text{time}) \approx 3.5 \log N - 7.9$, or the time scaled like $N^{3.5}$.

- Let's see if we can make sense of this scaling result. Consider matrix multiplication as a typical matrix operation. How should it scale with the size of the matrix? Consider $\mathbf{A} \times \mathbf{B} = \mathbf{C}$ or

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & \cdots \\ \vdots & & \ddots & \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1N} \\ b_{21} & b_{22} & \cdots & \cdots \\ \vdots & & \ddots & \\ b_{N1} & b_{N2} & \cdots & b_{NN} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1N} \\ c_{21} & c_{22} & \cdots & \cdots \\ \vdots & & \ddots & \\ c_{N1} & c_{N2} & \cdots & c_{NN} \end{pmatrix} \tag{5.5}$$

  [Note: When we use C++ arrays in our programs to represent matrices, the numbering will be from 0 to $N-1$ rather than from 1 to N.] Let's do some counting. There are N^2 matrix elements c_{ij} to calculate, and each one follows from N terms:

$$c_{ij} = \sum_{k=1}^{N} a_{ik} \times b_{kj} \ , \tag{5.6}$$

  so there are N^3 operations at a minimum. So if we have a full matrix, we would expect the matrix operations to scale roughly as N^3, unless we have "tricks" to speed things up. (Why we get an exponent of 3.5 is not entirely clear to me.)

- *By the scaling in the plot, how long would a $10^6 \times 10^6$ matrix take? How much memory would you need?* In fact, it is routine to find eigenvalues from such large matrices (and larger). Clearly it must be done another way!

- **Pointers and void pointers again.**

  The example of void pointers in the Session 4 program `pointer_test.cpp` is always confusing. The *only* reason we introduce them now is to take advantage of the GSL library routines, which use them repeatedly. *You are not expected to understand everything about pointers, but only to be able to imitate their use based on the examples.* Some follow-up comments:

  - **Pointers on pointers.** For a particularly good reference on pointers, check out `http://www.cplusplus.com/doc/tutorial/pointers.html`
  - There are two ways to define structures, but you only have to use the one you find convenient. (They are both introduced only for completeness; we choose to use the typedef form in `pointer_test.cpp`.) See section f.1 in the Session 4 notes for a discussion of the distinction between them.
  - Let's dissect what happens in the function `f_osu_parameters` in `pointer_test.cpp`. (see also section f.2 from Session 4).

    ```
    double passed_double_1 = ((osu_parameters *) params_ptr)->a;
    ```

    Our goal here is to recover one element of a structure whose address is passed to the function as the pointer `params_ptr`. The structure `osu_parameters` is defined with `a`, `b`, `c`, and `num` components; here we recover the value of the `a` component. We use `->` instead of a period when we have a *pointer* to the structure. The rest is specifying that `params_ptr` is a pointer to a structure of type `parameters`. The `( )`'s are important to include so that the statement is unambiguous! (Note that it is easiest to read the statement from right to left.)

c. Additional Notes on Programming in C++

- **Comments of the 6810 codes so far.** The programs used in class evolved for the most part from programs supplied by Landau and Paez along with their "Computational Physics" text. They were originally written in C and translated without major changes to C++. As such, they are compact but often not written optimally. For example, in the `eigen_basis.cpp` code, parts should be split to separate functions and different files, and it should be easier to add additional potentials. We should take advantage of C++ *classes. As we proceed, we'll look back occasionally and see how they could be improved.*
- **Namespaces.** You'll have noticed the statement:

  ```
  using namespace std;
  ```

  at the beginning of our programs, following the `#include <iostream>` and other include statements. Namespaces are used in C++ to avoid collisions between variables or functions that have the same name (e.g., you might decide you want to have your own function called "cout"). The real full name of the `cout` function is `std::cout`, which says that it is in the `std` (short for "standard") namespace. The `using` command lets us skip the `std::` prefix, but in general it would be better to use the prefix. Later we'll see how to define our own namespaces.

- **Scope of variables and global variables.** The "scope" of a variable refers to the region of the code where that variable is recognized. Some observations:
  - The scope of a variable declared within a function (including `main`) is, at most, that function (we say it is "local" to that function).
  - If declared within a "block" delimited by `{}`'s, then the variable is only known within that block (we say it is "local" to that block). We've seen this with dummy index variables in `for` loops. That is:

```
// i is not recognized here
for (int i=1; i < 10; i++)
{
  // i is recognized here
}
// i is not recognized here
```

  - Global variables, which are defined outside of any function (at the top of the file) and are accessible to all functions within that file, seem very convenient. However, in general they are not a good idea. We'll discuss later how to use classes, in which variables are hidden from the outside on purpose, to create more robust programs that can be modified and reused without fear of introducing subtle (or unsubtle!) bugs.

- **Parameters.** In many cases we will want to run a program multiple times with different values of parameters that are used by the code. We'll take as an example the summing up vs. summing down code you wrote for the first problem set. We may want to change the start value of N, the finish value, or the increment. How should we do this?

  First, we should define variables rather than just code the numbers. For example:

```
int Nstart = 100;
int Nmax = 1e8;
int Ntimes = 2;
```

  (Note: we would put `const` in front of these declarations if we don't intend these values to change while the program is running.) Then the `for` loop running through the N's would be:

```
for (N = Nstart; N <= Nmax; N* = Ntimes)
{
   <do stuff>
}
```

  We have several options for letting a user change the values of `Nstart`, `Nmax`, and `Ntimes`:

  1. edit the values in the code each time and recompile;
  2. read values from an input file;
  3. read values from the command line;

4. set the values in a *script*, which runs the code appropriately from the command line or from an input file created by the script.

We will look at examples of scripts in Python to do this (note that this is very useful if you inherit a code that you don't understand or which you are not allowed to change). You can also write such scripts in Perl or in the shell language `tcsh` or `bash` (this is the language you use at the command line).

d. Solving the Schrödinger Equation Numerically

The abstract bound-state, time-independent Schrödinger equation,

$$H\Psi = E\Psi \ , \tag{5.7}$$

can be solved in a variety of ways numerically. Here is a partial laundry list:

1. *Solve as a differential equation in coordinate representation* (when we have a *local* potential). In one dimension, this means solving the equation:

$$\left(-\frac{\hbar^2}{2M}\frac{d^2}{dx^2} + V(x)\right)\Psi_n(x) = E_n\Psi_n(x) \ . \tag{5.8}$$

If we have a central potential in three dimensions, the potential is purely radial

$$V(\mathbf{x}) = V(|\mathbf{x}|) \equiv V(r) \ , \tag{5.9}$$

and we can use a partial wave decomposition (which means we separate the equation in spherical coordinates). That is, we write

$$\Psi_{nlm}(\mathbf{x}) = \frac{u_{nl}(r)}{r}Y_{lm}(\theta,\phi) \ , \tag{5.10}$$

where Y_{lm} is a spherical harmonic, and solve the radial one-dimensional Schrödinger equation:

$$-\frac{\hbar^2}{2M}\frac{d^2u_{nl}(r)}{dr^2} + \underbrace{\left[V(r) + \frac{\hbar^2 l(l+1)}{2Mr^2}\right]}_{\equiv V_{\rm eff}(r)} u_{nl}(r) = E_n u_{nl}(r) \ , \tag{5.11}$$

with

$$u_{nl}(r=0) = 0 \qquad \text{and} \qquad \int_0^\infty |u_{nl}(r)|^2\, dr = 1 \ . \tag{5.12}$$

We'll be studying differential equations in the near future, at which time we'll come back to the details of solving Eq. (5.11).

2. *Matrix diagonalization in coordinate representation.* Return to Eq. (5.11) but now replace the second derivative with a finite difference formula (supressing the nl subscript):

$$\frac{d^2u}{dr^2} = \frac{u(r+h) - 2u(r) + u(r-h)}{h^2} + \mathcal{O}(h^2) \ . \tag{5.13}$$

(You can verify this approximation by expanding $u(r+h)$ and $u(r-h)$ in Taylor series about $u(r)$.) Let's suppose we solve this system knowing the *boundary conditions* at $r = 0$ and $r = R_{\max}$. We know the former from Eq. (5.12), $u(0) = 0$, and we'll suppose $R_{\max}$ is large enough so that $u(R_{\max}) \approx 0$ for any bound states. (This also makes the continuum a discrete set of states.) We'll need a labeling system for the points; we'll follow the one in Ref. [2]:

$$x_i = i \times h \ , \qquad i = 0, 1, 2, \cdots, N \tag{5.14}$$

where N is the number of steps and the step size h is given by:

$$h = \frac{R_{\max}}{N} \ . \tag{5.15}$$

Thus, $x_0 = 0$, $x_1 = h$, and so on up to $x_N = Nh = R_{\max}$. So we can approximate the Schrödinger equation at point x_k as

$$-\frac{\hbar^2}{2M}\frac{u(x_k+h) - 2u(x_k) + u(x_k-h)}{h^2} + V(x_k)u(x_k) = Eu(x_k) \ . \tag{5.16}$$

If we work in units where $\hbar = 1$ and also $M = 1/2$, and if we use the notation:

$$u_k \equiv u(x_k) \, , \quad u_{k\pm1} \equiv u(x_k \pm h) \, , \quad V_k \equiv V(x_k) \ , \tag{5.17}$$

then the equation at k takes the form

$$-\frac{u_{k+1} - 2u_k + u_{k-1}}{h^2} + V_k u_k = Eu_k \ . \tag{5.18}$$

We know two values, $u_0 = 0$ and $u_N = 0$. We can put the rest, u_1 to u_{N-1} in a column vector, which then satisfies a matrix eigenvalue problem:

$$\begin{pmatrix} \frac{2}{h^2} + V_1 & -\frac{1}{h^2} & 0 & \cdots & 0 \\ -\frac{1}{h^2} & \frac{2}{h^2} + V_2 & -\frac{1}{h^2} & & \vdots \\ 0 & -\frac{1}{h^2} & \ddots & & \vdots \\ \vdots & & & \ddots & -\frac{1}{h^2} \\ 0 & \cdots & \cdots & -\frac{1}{h^2} & \frac{2}{h^2} + V_{N-1} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ \vdots \\ u_{N-1} \end{pmatrix} = E \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ \vdots \\ u_{N-1} \end{pmatrix} \tag{5.19}$$

Make sure that you see how this matrix equation is just a summary of the $(N-1)$ *equations of the form Eq. (5.18) for* $k = 1$ *to* $N-1$. This is a *tridiagonal* matrix with a simple structure: the only non-zero off-diagonal matrix elements are the ones adjacent to the diagonal, and they all have the same value, $-1/h^2$. There are special algorithms that can rapidly find the eigenvalues and eigenvectors of such a matrix.

3. *Introduce a (truncated) orthonormal basis in which to expand* $u_{nl}(r)$ *and diagonalize the matrix of coefficients.* [Note: we'll assume $l = 0$ from here on, and drop the l label.] Imagine we have a set of basis functions:

$$\{\phi_i(r)\} \, , \quad i = 0, 1, \cdots, D-1 \ , \tag{5.20}$$

which we've truncated at D states (since we can only use a finite number in the computer), although in principle there are an infinite number. Orthonormality means that

$$\int_0^\infty \phi_i(r)\phi_j(r)\,dr = \delta_{ij} = \begin{cases} 1 & \text{if} \quad i = j\,, \\ 0 & \text{if} \quad i \neq j\,. \end{cases} \tag{5.21}$$

We can take the ϕ's to be real. (*Why?*) Then the expansion and coefficients are:

$$u_n(r) \approx \sum_{i=0}^{D-1} C_i^{(n)}\phi_i(r) \quad \Longrightarrow \quad C_j^{(n)} = \int_0^\infty \phi_j(r)u_n(r)\,dr\ . \tag{5.22}$$

(Can you derive the expression for $C_j^{(n)}$?) If we substitute the expansion for $u_n(r)$ in the Schrödinger equation (5.11), multiply by $\phi_i(r)$ and integrate over r,

$$\sum_{j=0}^{D-1} \underbrace{\int_0^\infty \phi_i(r)\left[-\frac{\hbar^2}{2M}\frac{d^2}{dr^2} + V_{\text{eff}}(r)\right]\phi_j(r)\,dr}_{\equiv H_{ij}} \cdot C_j^{(n)} = E_n \sum_{j=0}^{D-1} C_j^{(n)} \int_0^\infty \phi_i(r)\phi_j(r)\,dr = E_n C_i^{(n)}\ , \tag{5.23}$$

or

$$\sum_{j=0}^{D-1} H_{ij}C_j^{(n)} = E_n C_i^{(n)}\ . \tag{5.24}$$

This is simply a matrix eigenvalue problem (take the time to make sure you see that this is true!):

$$\begin{pmatrix} H_{00} & H_{01} & \cdots & \cdots & H_{0D-1} \\ H_{10} & H_{11} & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ H_{D-10} & \cdots & \cdots & \cdots & H_{D-1D-1} \end{pmatrix} \begin{pmatrix} C_0^{(n)} \\ \vdots \\ \vdots \\ \vdots \\ C_{D-1}^{(n)} \end{pmatrix} = E_n \begin{pmatrix} C_0^{(n)} \\ \vdots \\ \vdots \\ \vdots \\ C_{D-1}^{(n)} \end{pmatrix} \tag{5.25}$$

which we can give to a packaged routine (e.g., from GSL or LAPACK).

In Session 5, we'll use harmonic oscillator radial wave functions as a basis. The potential for these wave functions is

$$V(r) = \frac{1}{2}M\omega^2 r^2\ . \tag{5.26}$$

We define the *oscillator parameter* b by

$$\hbar\omega = \frac{\hbar^2}{Mb^2}\ , \tag{5.27}$$

and use units in which $\hbar = 1$. This means that b sets the length scale and $q \equiv r/b$ is the natural dimensionless coordinate. The oscillator state $u_{nl}(r)$ is specified by the radial

quantum number n and the angular momentum quantum number l (we'll be using $l = 0$ in class), with normalization

$$\int_0^\infty dr\,[u_{nl}(r)]^2 = 1\ . \tag{5.28}$$

Here are some questions to think about:

- What do the lowest wave functions look like? E.g., how many nodes does the ground state wave function have? The 1st excited state? Sketch the first few.
- At large r, the wavefunctions will fall off like $e^{-r^2/2b^2}$. If you want to accurately build up the ground state wave function for a square well, how would you choose b so that you can use the smallest number of terms to get a given accuracy? How about a hydrogen-like wave function? It turns out that such questions are important in nuclear structure calculations; we'll discuss further (and play in a problem set) the answers.
- The diagonalization of a Hamiltonian in a truncated basis can be viewed as a *variational* calculation (we'll discuss this further in future notes). What are the implications for:
  - What state (ground state or an excited state) is determined best?
  - How should the difference from the exact answer change as the basis size is increased?

4. *Solve an integral equation in momentum representation.* (We'll come back to this later!).

e. 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 a new 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.