6. 6810 Session 6

a. Follow-ups to Session 5 (and earlier)

- **Normalization of $u_{nl}(r)$ from `eigen_tridiagonal.cpp`.** This code implemented the following matrix representation of the Schrödinger eigenvalue problem (see Session 5 notes):

$$\begin{pmatrix} \frac{2}{h^2}+V_1 & -\frac{1}{h^2} & 0 & \cdots & 0 \\ -\frac{1}{h^2} & \frac{2}{h^2}+V_2 & -\frac{1}{h^2} & & \vdots \\ 0 & -\frac{1}{h^2} & \ddots & & \vdots \\ \vdots & & & \ddots & -\frac{1}{h^2} \\ 0 & \cdots & \cdots & -\frac{1}{h^2} & \frac{2}{h^2}+V_{N-1} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ \vdots \\ u_{N-1} \end{pmatrix} = E \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ \vdots \\ u_{N-1} \end{pmatrix}, \qquad (6.1)$$

  where the eigenvector corresponds to the value of the radial wave function at equally spaced r points. The eigenvector is normalized by to unity; this means that $\sum_{i=0}^{N-1} |u_i|^2 = 1$. This is *not* the same as $\int_0^\infty |u(r)|^2\, dr = 1$, which is the quantum mechanics normalization condition; this is apparent if you plot the $\{u_i\}$'s for different mesh spacings. The former *would* be an approximation to the latter if we included a factor of the mesh spacing h. *(Why? What is the simplest integration rule you could use?)* So we could scale each u_i by $\sqrt{h}$ to approximately normalize the wave function.

- **Diagonalizing a million by million matrix.** By one estimate based on timings of the Hilbert matrix diagonalization in class (see Session 5 notes if you didn't get to this in Session 4), if the scaling does not change (it is most likely to only get worse), the time for $N = 10^6$ (that is, a $10^6 \times 10^6$ matrix) would be about 35,000 years. This is too long for any computer outside of *A Hitchhiker's Guide to the Galaxy*. So other approaches are used for such large matrices. For example, usually only a small fraction of the eigenvalues are needed (typically the lowest ones). A numerical technique called the *Lanczos method* is effective in generating the lowest (or highest) eigenvalues for large, symmetric, but sparse (most elements are zero) matrices. This method is adapted for processors working in parallel to tackle the largest problems (in nuclear physics, problems up to and beyond $N = 10^9$ are now routine).

- **Speeding up the eigenvalue program for large basis size.** How can we make the `eigen_basis` code run faster? Consider two possibilities for where most of the time is spent:

  1. Diagonalizing the matrix takes most of the time. Based on our experience, how does this scale with the size of the basis?
  2. Calculating the individual matrix elements takes most of the time. For the Hilbert matrix, this part of the calculation took almost no time, but not in general. How should this part scale with the size of the basis?

  We can identify which possibility dominates for a given range of matrix sizes by doing the timings. For matrices up to 100×100, we find that the time scales as N^2. Which of the two possibilities is indicated by this scaling? Given what we know about the matrix (e.g., it is symmetric), what is a very quick change that should improve the speed by a factor of about two?

- **Note on calculating matrix elements.** The ij matrix element of the Hamiltonian with potential $V(r)$ is given by the integral

$$H_{ij} = \int_0^\infty \phi_j(r) \left[-\frac{\hbar^2}{2M} \frac{d^2}{dr^2} + V(r) \right] \phi_i(r)\, dr \ , \tag{6.2}$$

where ϕ_i and ϕ_j are harmonic oscillator basis wave functions. This is not the best thing to calculate numerically, because we would have to do numerical derivatives. Instead, we use the fact that the $\{\phi_i\}$ satisfy a differential equation with a second derivative ($l = 0$ is assumed):

$$\left[-\frac{\hbar^2}{2M} \frac{d^2}{dr^2} + \frac{1}{2} M\omega^2 r^2 \right] \phi_i(r) = \hbar\omega \left(2i + \frac{3}{2} \right) \phi_i(r) \ . \tag{6.3}$$

Thus, we can eliminate the second derivative to obtain

$$H_{ij} = \int_0^\infty \phi_j(r) \left[\hbar\omega \left(2i + \frac{3}{2} \right) - \frac{1}{2} M\omega^2 r^2 + V(r) \right] \phi_i(r)\, dr \ , \tag{6.4}$$

which is what is calculated in the program `eigen_basis.cpp`.

- **Diagonalization of a truncated basis as a variational problem.** How might you analyze the eigenvalue program if you didn't know the correct answer for the eigenvalue? Instead of looking for the lowest error, we could look for the most stable region in b or when the basis gets larger. Are we guaranteed that the estimate of the energy gets better as the basis size increases? (Be careful: remember we are doing our calculations on a computer, where round-off errors are always waiting for us!) In fact, the calculation we are doing is equivalent to a *variational* estimate for the ground state.

  Recall how a variational calculation works. If $u_{\rm trial}(r)$ is a (real) normalized trial wave function with parameter b (e.g., $u_{\rm trial}(r) \propto r e^{-r^2/b^2}$), then the estimate of the energy for that b is:

$$E(b) \equiv \langle u_{\rm trial} | H | u_{\rm trial} \rangle = \int_0^\infty dr\, u_{\rm trial}(r) \left[-\frac{\hbar^2}{2M} \frac{d^2}{dr^2} + V(r) \right] u_{\rm trial}(r) \ . \tag{6.5}$$

  (What do we do if the trial wave function is *not* normalized? Hint: Divide by another integral.) The *best* estimate is b_0, where

$$\left. \frac{dE}{db} \right|_{b_0} = 0 \ , \tag{6.6}$$

  and $E(b_0)$ is an *upper bound* to the true energy (that is, the actual energy is always lower, which usually means more negative).

  So now suppose our trial wave function is a sum of D basis functions with arbitary coefficients;

$$u_{\rm trial}(r) = \sum_{i=0}^{D-1} C_i \phi_i(r) \ , \tag{6.7}$$

  where the $\{\phi_i(r)\}$ are a complete orthonormal basis (e.g., our harmonic oscillator basis). We want to minimize $\langle u_{\rm trial} | H | u_{\rm trial} \rangle$ subject to the constraint that $|u_{\rm trial}\rangle$ is normalized. The

$\{C_i\}$ are the variational parameters. We use the method of *Lagrange multipliers*. Then for each k, we require

$$\frac{\partial}{\partial C_k}\left[\langle u_{\rm trial}|H|u_{\rm trial}\rangle - \lambda(\langle u_{\rm trial}|u_{\rm trial}\rangle - 1)\right] = 0 \ , \tag{6.8}$$

and $\partial/\partial\lambda[\cdots] = 0$, which gives $\sum_i |C_i|^2 = 1$. Before tackling the general case, let's do the simplest non-trivial special case: two basis states with coefficients C_0 and C_1. The first condition is:

$$\frac{\partial}{\partial C_0}\left[C_0^2\langle\phi_0|H|\phi_0\rangle + C_0C_1\langle\phi_0|H|\phi_1\rangle + C_1C_0\langle\phi_1|H|\phi_0\rangle + C_1^2\langle\phi_1|H|\phi_1\rangle - \lambda C_0^2 - \lambda C_1^2\right] = 0 \ , \tag{6.9}$$

or (using the fact that H is Hermitian)

$$2C_0\langle\phi_0|H|\phi_0\rangle + 2C_1\langle\phi_0|H|\phi_1\rangle - 2\lambda C_0 = 0 \ , \tag{6.10}$$

or switching notation to $\langle\phi_i|H|\phi_j\rangle \equiv H_{ij}$ and dividing by 2:

$$C_0H_{00} + C_1H_{01} = \lambda C_0 \ . \tag{6.11}$$

The $\partial/\partial C_1$ contribution is similar; when we combine them as a matrix equation, we find

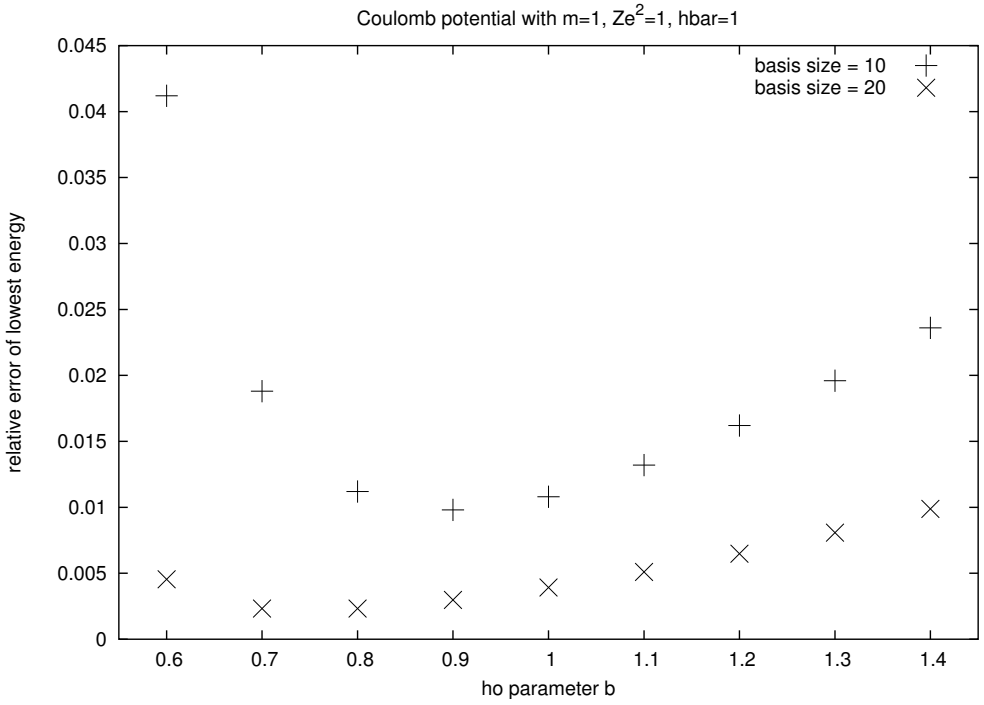
$$\begin{pmatrix} H_{00} & H_{01} \\ H_{10} & H_{11} \end{pmatrix}\begin{pmatrix} C_0 \\ C_1 \end{pmatrix} = \lambda\begin{pmatrix} C_0 \\ C_1 \end{pmatrix} \ , \tag{6.12}$$

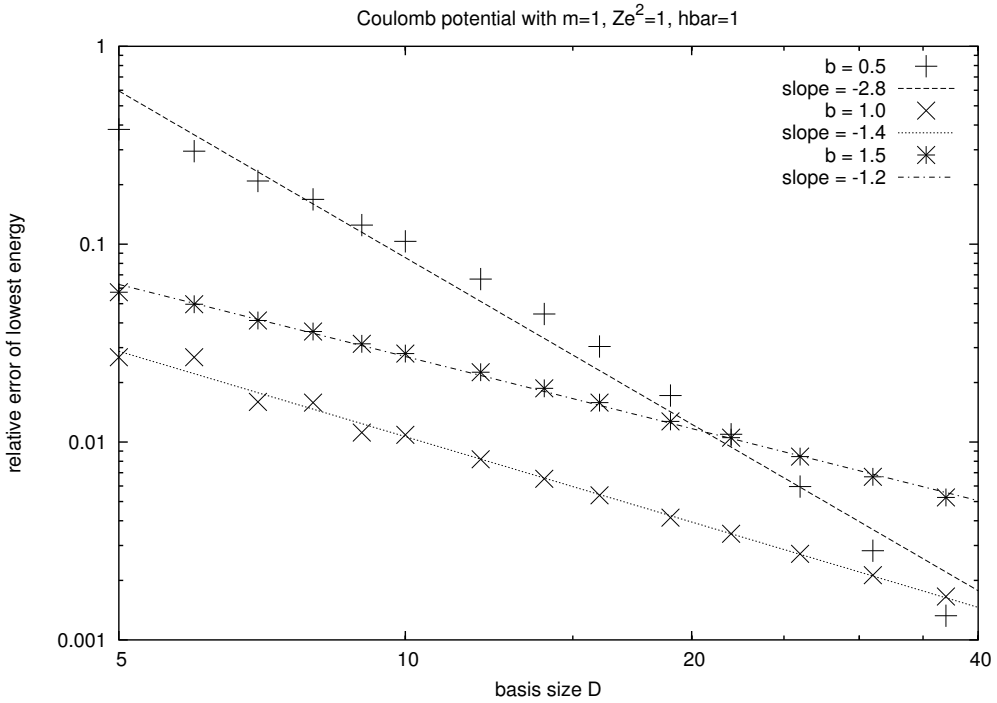
which is precisely our eigenvalue equation in the truncated basis! Note that the Lagrange multiplier will be given by an energy eigenvalue. More generally, we find that we get the $k^{\rm th}$ row of the eigenvalue matrix equation from

$$\frac{1}{2}\frac{\partial}{\partial C_k}\left(\sum_{ij} C_iC_jH_{ij} - \lambda\sum_{ij} C_iC_j\delta_{ij}\right) = \sum_j H_{kj}C_j - \lambda C_k = 0 \ . \tag{6.13}$$

- **Coulomb ground state with a harmonic oscillator basis.** You should have found that the bound states of a square-well potential (and particularly the lowest one) could be found quite accurately using a harmonic oscillator basis. But the Coulomb potential doesn't work as well. Why is it so much harder? Two figures generated using `eigen_basis.cpp` are shown on the next page. In the first, the error of the ground state energy for the Coulomb potential is plotted against the harmonic oscillator parameter b. There are clear minima at fixed basis size $D = 10$ and at $D = 20$, but the minima are at different values of b. Why is this? Recall that the width of the basis functions and the rate of fall-off at large r is set by b. Why would a larger scale work best for a small basis, but a smaller scale do better with a larger basis?

  In the second figure, the relative error as a function of the basis size is plotted on a log-log plot, for several fixed values of b. Note the very different slopes for different values of b. The dependence is reasonably approximated by a constant power law (with some oscillations for the smallest b). The steep slope for $b = 0.5$ means that even though the $b = 0.5$ basis does a poor job for small basis size, eventually it becomes the best of the three shown. Can you explain this? You'll revisit these questions in Assignment #3.


Coulomb potential with m=1, Ze^2=1, hbar=1
basis size = 10
basis size = 20
relative error of lowest energy
ho parameter b
0.045
0.04
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0
0.6
0.7
0.8
0.9
1
1.1
1.2
1.3
1.4


Coulomb potential with m=1, Ze^2=1, hbar=1
b = 0.5
slope = -2.8
b = 1.0
slope = -1.4
b = 1.5
slope = -1.2
relative error of lowest energy
basis size D
1
0.1
0.01
0.001
5
10
20
40

- **GSL error handling.** You may have run into errors when running GSL routines and found that they just caused the program to abort without any particularly useful information. This is the default behavior, but you can change it by modifying the *error handling* (which means how GSL functions report and deal with errors). Here's how to get useful information from errors. Suppose the GSL function you are calling is named `gsl_function`.

  1. Include the header for the GSL error routines:
     ```
     #include <gsl/gsl_errno.h>     // gsl error routines
     ```
  2. Near the beginning of your program, turn off the default behavior with the command:
     ```
     gsl_set_error_handler_off ();   // turn off the GSL error handler
     ```
  3. A GSL function will have an integer (`int`) return value, which we will call `status`:
     ```
     int status = gsl_function (whatever);  // call the GSL function
     ```
  4. If `status=0` (which is the same as "false"), the function terminated normally (no errors), if not, we can get a description of the error from the `gsl_strerror` function. For example, after calling the function we could have the code:
     ```
     if (status)    // if status=0 move on, otherwise print error message
     {
       cout << " GSL error: " << gsl_strerror (status) << endl;
     }
     ```
     With these changes, the program will continue to run but will warn you if there is some problem with the GSL function.

b. Code Design: `eigen_basis` Program

Here we provide some background for the structure of the `eigen_basis` program, which finds the eigenvalues of various potentials by diagonalizing the Hamiltonian in a truncated harmonic oscillator basis. This is a possible scenario for creating this program.

First Pass. First we decide on our approach and thus the equations. We'll use method 3 from section c of the Session 5 notes for solving the Schrödinger equation (the relevant equations are given there). Then we make a rough outline (pseudo-code) to carry out our task.

1. Decide on a potential to use.
2. Decide on the details of the basis.
3. Fill a matrix by calculating matrix elements H_{ij} according to Eq. (5.23).
4. Find the eigenvalues and eigenvectors.
5. Print out the results.

Second Pass. Fill in some of the details for each step of the outline:

1. We'll ask the user what potential to use. Initially the choices will be between Coulomb and square well. Each Hamiltonian will have related parameters, which will become variables.

We'll have the particle mass in all cases for the kinetic energy, and then parameters like e^2 for the Coulomb potential and the radius and depth for the square well potential.

2. We'll need to know the size of the basis, the mass, and the harmonic oscillator parameter b.
3. For the ij matrix element, we need to do an integral over the potential and harmonic oscillator wave functions i and j. We choose to use a GSL routine for the integration and calculate the harmonic oscillator wave functions ourselves in a separate routine.
4. We'll use GSL eigensystem routines to find the eigenvalues and eigenvectors. After looking up the manual and an example, we realize this means we'll have to
   (a) define and allocate space for matrices and vectors we need;
   (b) load the matrix to be diagonalized;
   (c) find the eigenvalues and eigenvectors with `gsl_eigensymmv`;
   (d) sort the results numerically.
5. Print out the matrix elments as we calculate them and print out the eigenvalues in order (also prepare to print out the corresponding eigenvectors).

Third Pass. Implementation of elements of our outline (check in the code to see how it was done in detail).

1. Do this in the main program using `cout` and `cin`. Define a structure called `potential_parameters` to hold up to three generic parameters for the potential. Write separate functions for each potential, which take the radius r and a `potential_parameters` structure as arguments. Call them `V_coulomb` and `V_square_well`.
2. Ask the user for the oscillator parameter `b_ho` and the size of the basis `dimension` with `cout`/`cin` statements. Store these in a structure that we'll name `hij_parameters`. (The particular instance of this structure will be called `ho_parameters`.) We write a separate function `ho_radial` (in a different file) that calculates the radial harmonic oscillator wave function and another function called `ho_eigenvalue` that calculates the harmonic oscillator eigenvalue.
3. Write a function called `Hij` that get passed the `hij_parameters` structure and which calls the GSL routine `gsl_integration_qagiu` to do the integral. That routine in turn needs an integrand function, which we call `Hij_integrand`. The integrand simply multiplies together the terms in Eq. (5.12).
4. Pick out the appropriate GSL routines
   (a) Use `gsl_matrix` together with `gsl_matrix_alloc` (and similarly for the vectors). Note that we define *pointers*.
   (b) Use `gsl_matrix_set` to load `Hmat_ptr`.
   (c) Use `gsl_eigen_symmv`.
   (d) Use `gsl_eigen_symmv_sort` with `GSL_EIGEN_SORT_VAL_ASC` to get ascending order.

5. Use `gsl_vector_get` to get the eigenvalues one by one. At the end, free up the memory used with `gsl_matrix_free` and `gsl_vector_free`.

The `eigen_basis` program evolved from a C code that was written by someone else. If we were starting again in C++, I would make some different choices. For example, here are classes we might use:

- Introduce a class for potentials. Each particular potential is an instance of the class. This would combine the structure that have parameters of the potential with the function that evaluates the potential.
- Introduce a class for the harmonic oscillator basis functions, which would include the function to evaluate them as well as the harmonic oscillator parameters.
- Move the matrix loading and calculation into a class to isolate it from the main function and give the option to use different eigensystem routines (e.g., LAPACK).

In the next section we'll motivate an implementation of a Hamiltonian class for `eigen_tridiagonal` that "wraps" the GSL functions.

c. Follow-up: Wrapping GSL Matrix Functions in a Class

In Session 6 we look at a re-write of the `eigen_tridiagonal.cpp` code to make use of C++ classes. Here we motivate what was done and why.

- **Motivation.** The purpose of the `eigen_tridiagonal` code is simple: Given a potential, set up a Hamiltonian and calculate its eigenvalues and eigenvectors. The only parameter is the dimension of the Hamiltonian matrix. While the method (discretizing in coordinate representation) is important (e.g., it determines the accuracy), the details of *how* the calculation is carried out are not.
  - There are GSL header files, allocation, and function calls, with obscure names and usage that require the GSL manual to decipher.
  - In general, why should the user have to worry about how the Hamiltonian is stored and allocated and so on?
  - And what if we want to do the same type of eigenvalue/eigenvector calculation in another program? We'd have to cut and paste.
  - What if we want to use another library, such as LAPACK, to diagonalize the Hamiltonian? We'd have to change the whole set up, even though the program functionality is unchanged.

  The bottom line is we can (and should!) hide those details ("encapsulate" them) in a class.

d. Computing with OpenMP

In Session 6, we will consider a simple example of parallel processing using using OpenMP ("Open Multi-Processing") on a multi-core computer (which is increasingly the norm now). OpenMP implements *multithreading* to split your running process into multiple sub-processes ("threads") that get allocated to different cores and execute in parallel. The example we use is a very common one: we have a for loop with many iterations and we would like OpenMP to divide these up among available cores (with as little guidance from us as possible). So if there are 1000 iterations and two cores available sharing memory, we would like each of the cores to run 500 of the iterations.

The OpenMP programming model has these features:

- All of the parallel threads have access to the same memory that is shared globally.
- However, each piece of data (e.g., a variable or an array) can be either shared among all threads (that is, each has access) or private to a given thread that owns it.
- Changes in local data are not seen by any other thread, while changes in shared data is seen by all other threads.
- The transfer of data is transparent to the programmer.
- There is synchronization of threads, but it mostly happens behinds the scene (that is, the code does not explicitly force it).

The basic execution model is a "fork and join" model in which a master thread splits into parallel worker threads in a parallel region after which they are joined and synchronized. The parallel region is denoted by the block following a `#pragma omp parallel` compiler directive. There are many options but we will only look at a simple case, which is nevertheless widely applicable.

Here is a simple code snippet illustrating matrix vector multiplication. It is assumed that the `b_matrix` and `c_vector` are defined earlier.

```
#pragma omp parallel for default(none) \
    private(i,j,sum) shared(size,a_vector,b_matrix,c_vector)
for (i=0; i<size; i++)
{
  sum = 0.0;
  for (j=0; j<size; j++)
  {
    sum += b_matrix[i][j] * c_vector[j];
  }
  a_vector[i] = sum;
}  // this is the end of the parallelization block
```

The `default(none)` clause means that the scope of all variables, either `private` or `shared`, must be declared. If there is no `default` clause, it is the same as `default(shared)`, meaning all variables are shared. Because `i`, `j`, and `sum` are listed in the `#pragma` directive, they must be declared outside

of the block. Note that only the outer for loop is parallelized; the inner one is executed by each thread.

In Session 6 you'll try an even simpler example that only uses `#pragma omp parallel for` (so that all variables are shared).

e. Differential Equations

In Session 6, we'll get started with differential equations by investigating two algorithms for solving them. Physics is full of differential equations and as a result, they are a major part of computational physics. Some of the more interesting problems involve *nonlinear* differential equations and *partial differential equations* (PDE's), which we'll study later. A linear differential equation of great interest is the 2nd order Schrödinger equation for energy eigenvalues.

e.1 Reminder of Terminology

I'm assuming that you have seen all (or most) of this before and that this is therefore just a review. The *order* refers to the largest order of derivatives. *Linear* or *nonlinear* refers to whether the dependent variable appears only to the first power or to higher powers. Examples:

$$\frac{dy}{dt} = f(t,y) \qquad \text{1st order;} \tag{6.14}$$

$$\frac{d^2y}{dt^2} = f\left(t, \frac{dy}{dt}, y\right) \qquad \text{2nd order,} \tag{6.15}$$

where these could be linear or nonlinear, depending on f. Here are some linear equations:

$$M\frac{d^2x}{dt^2} = -kx \qquad \text{2nd order, linear;} \tag{6.16}$$

$$i\hbar\frac{\partial\Psi(\mathbf{x},t)}{\partial t} = -\frac{\hbar^2}{2M}\nabla^2\Psi(\mathbf{x},t) + V(\mathbf{x})\Psi(\mathbf{x},t) \qquad \text{PDE, linear;} \tag{6.17}$$

$$\frac{dy}{dt} = g^3(t)y(t) \qquad \text{1st order, linear;} \tag{6.18}$$

and here is a nonlinear one:

$$\frac{dy}{dt} = g^3(t)y(t) - g(t)y^2(t) \qquad \text{1st order, nonlinear.} \tag{6.19}$$

Note that it is not the dependence on the independent variable t that determines whether it is linear or nonlinear, but the dependence on $y(t)$. Furthermore, when there is more than one condition to specify, we can have *initial* conditions [e.g., specify everything at $t = 0$] or *boundary* conditions [e.g., for Eq. (6.15), specify $y(t)$ and dy/dt at $t = 0$, or specify $y(t_i)$ and $y(t_f)$]. The solution methods generally differ for these two possibilities.

e.2 Finite Difference Methods: Euler's Method

The simplest way to solve a first-order differential equation (or a system of first-order differential equations) is to apply the crudest of our derivative formulas: forward difference. This is called Euler's method. Suppose our equation is

$$y'(t) \equiv \frac{dy}{dt} = f(t, y) \ , \tag{6.20}$$

and we want to solve for $y(t)$ in the interval $a \le t \le b$ given the initial condition

$$y_0 \equiv y(t = t_0) \quad \text{where } t_0 = a \ . \tag{6.21}$$

Simple examples of equations of this type are

$$\frac{dy}{dt} = -at \qquad \text{or} \qquad \frac{dy}{dt} = -ky \tag{6.22}$$

(note that the y and t are just dummy variables).

The plan is to divide $[a, b]$ into $N_{\text{intervals}}$ subintervals of equal ("fixed") width h:

$$h = \frac{b - a}{N_{\text{intervals}}} \ , \tag{6.23}$$

so that

$$y_i \equiv y(t_i) \quad \text{with} \quad t_i = t_0 + i * h \ . \tag{6.24}$$

Given y_0, we can find y_1, y_2, ... *sequentially.* The forward-difference formula for a derivative says

$$y'(t_i) \approx \frac{y(t_i + h) - y(t_i)}{h} = \frac{y_{i+1} - y_i}{h} \ , \tag{6.25}$$

but we also have

$$y'(t_i) = f(t_i, y_i) \ , \tag{6.26}$$

so

$$y_{i+1} \approx y_i + hf(t_i, y_i) + \mathcal{O}(h^2) \ . \tag{6.27}$$

(Where did the truncation error come from?) Thus our solution is

$$y_1 = y_0 + hf(t_0, y_0) \tag{6.28}$$

$$y_2 = y_1 + hf(t_1, y_1) \quad \text{[using } y_1 \text{ from the first equation]} \tag{6.29}$$

$$\vdots \tag{6.30}$$

where we evaluate the first equation to find y_1, use it to evaluate the second equation to find y_2, and so on.

Let's apply this to a simple example:

$$\frac{dy}{dt} = -\alpha y \quad \text{with} \quad y(0) = 1 \ , \qquad \text{exact answer: } y(t) = e^{-\alpha t} \ , \tag{6.31}$$

so that $f(y_i, t_i) = -\alpha y_i$. Here's a table of the results for the first few steps:

i	Euler's method for y_i	exact y_i	error
0	1	$e^{-\alpha\cdot 0}=1$	0
1	$y_0+h(-\alpha y_0)=1-\alpha h$	$e^{-\alpha h}=1-\alpha h+\frac{1}{2}\alpha^2h^2+\cdots$	$\frac{1}{2}\alpha^2h^2+\cdots$
2	$y_1+h(-\alpha y_1)=1-2\alpha h+\alpha^2h^2$	$e^{-2\alpha h}=1-2\alpha h+2\alpha^2h^2+\cdots$	$\alpha^2h^2+\cdots$
3	$y_1+h(-\alpha y_2)=1-3\alpha h+3\alpha^2h^2-\alpha^3h^3$	$e^{-3\alpha h}=1-3\alpha h+\frac{9}{2}\alpha^2h^2+\cdots$	$\frac{3}{2}\alpha^2h^2+\cdots$

Note that the error with each step goes like h^2 but that the total error gets bigger as we go. Thus, we have two related errors we can talk about. The *local* error is the mistake we make with each step. From Eq. (6.27), we see that the local error for Euler's method is $\mathcal{O}(h^2)$. But we also have the *global* or *accumulated* error. How do the errors built up as we take $N_{\rm interval}$ additional steps? We'll explore this empirically in Session 6.

How do we keep the errors under control? Since the error will scale as a power of h, the obvious answer is to make h small so the correction is small. However, we know that we will have subtractive cancellations (particularly for decreasing functions, remember the spherical Bessel functions), which will lead to round-off errors. So we will need to compromise based on an error analysis. Note that there is nothing that requires us to use a fixed h throughout the interval. We might use a small value of h when the function is varying rapidly and a large value when it is varying slowly. How might we decide what h to use if we don't know the exact answer to the function we are integrating? (Hint: Remember what we did with integrals in an analogous situation.)

e.3 Runge-Kutta Methods

How can we do better than Euler's method? From our experience with derivatives, we know that the central difference formula is much better than the forward difference formula, with the same number of function evaluations. If we directly applied it to t_i, we'd have

$$\left.\frac{dy}{dt}\right|_{t_i} \approx \frac{y(t_i+h/2)-y(t_i-h/2)}{h}+\mathcal{O}(h^2)\ , \tag{6.32}$$

which is not precisely what we want, since it is $y(t_i+h)$ we want to get. But if we shift by $h/2$, then

$$\left.\frac{dy}{dt}\right|_{t_i+h/2} \approx \frac{y(t_i+h)-y(t_i)}{h}+\mathcal{O}(h^2)=f(y_{i+1/2},t_i+h/2)\ , \tag{6.33}$$

or

$$y_{i+1}=y_i+hf(y_{i+1/2},t_{i+1/2})\ , \tag{6.34}$$

as an improvement to Eq. (6.27). The problem now is that on the right side we need to know what $y_{i+1/2}$ is. For that, we can use Euler again:

$$y_{i+1/2}=y_i+\frac{1}{2}hf(y_i,t_i)+\mathcal{O}(h^2)\equiv y_i+\frac{1}{2}k_1\ . \tag{6.35}$$

So the final rule is

$$y_{i+1}\approx y_i+k_2 \quad\text{with}\quad k_1=hf(y_i,t_i) \quad\text{and}\quad k_2=hf(y_i+k_1/2,t_i+h/2)\ . \tag{6.36}$$

This is called the *second-order Runge-Kutta* method.

Let's check our simple example from Eq. (6.31) with this method. We start with $y_0 = 1$ as before. To get y_1, we first calculate

$$k_1 = h(-\alpha y_0) = -h\alpha \tag{6.37}$$

and then

$$k_2 = hf(y_0 - h\alpha/2, h/2) = h(-\alpha)(y_0 - h\alpha/2) = -h\alpha(1 - h\alpha/2) \ , \tag{6.38}$$

and then finally,

$$y_1 \approx y_0 + k_2 = y_0 - h\alpha + \frac{1}{2}h^2\alpha^2 = 1 - h\alpha + \frac{1}{2}h^2\alpha^2 \ , \tag{6.39}$$

which is now correct through $\mathcal{O}(h^2)$!

Another approach to 2nd and higher-order Runge-Kutta methods is described in the Landau and Paez book [1]. The basic idea is to note that we can integrate both sides of the differential equation:

$$\frac{dy}{dt} = f(y,t) \quad \Longrightarrow \quad y(t) - y(t_0) = \int_{t_0}^{t} f(y(t'),t')\,dt' \ , \tag{6.40}$$

which when applied to $t \to t_{i+1}$ and $t_0 \to t_i$ yields

$$y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f(y,t)\,dt \ . \tag{6.41}$$

Thus, any approximation to the integral on the right side will give us a method for solving differential equations. If we take f constant in the interval and evaluate it at t_i, we recover Euler's method. If we expand minimally about the midpoint instead, we get the 2nd-order Runge-Kutta method. If we expand further, we get higher-order methods (4th-order Runge Kutta is given below).

e.4 Coupled Linear Equations

If we have a second-order ordinary differential equation (ODE), we can recast it as a pair of coupled first-order equations. The implementations of Euler's and Runge-Kutta methods in Session 6 are geared to solving such systems (e.g., the variable N in the programs is the number of equations). For example, take $F = ma$ for a general force:

$$\frac{d^2x}{dt^2} = \frac{F(x,v,t)}{M} \ , \tag{6.42}$$

which can be re-expressed as the vector differential equation

$$\frac{d\mathbf{y}}{dt} = \mathbf{f} \quad \text{where} \quad \mathbf{y} = \begin{pmatrix} y^{(0)} \\ y^{(1)} \end{pmatrix} \quad \text{and} \quad \mathbf{f} = \begin{pmatrix} y^{(1)}(t) \\ \frac{1}{M}F(\mathbf{y},t) \end{pmatrix} \tag{6.43}$$

if we make the definitions

$$y^{(0)}(t) \equiv x(t) \qquad \text{and} \qquad y^{(1)}(t) \equiv v = \frac{dx}{dt} = \frac{dy^{(0)}}{dt} \ . \tag{6.44}$$

(Check it out for yourself!) Applying any rule in vector form basically just means stepping through the components of the vector every time we take another step from i to $i+1$.

The vector form of Euler's algorithm for N equations is

$$\mathbf{y}_{i+1} \approx \mathbf{y}_i + h\mathbf{f}(\mathbf{y}_i, t_i) \ , \quad (i = 0, \cdots, N-1) \tag{6.45}$$

while the vector form of 2nd-order Runge Kutta is $(i = 0, \cdots, N-1)$:

$$\mathbf{y}_{i+1} \approx \mathbf{y}_i + \mathbf{k}_2 \ , \qquad \mathbf{k}_1 = h\mathbf{f}(\mathbf{y}_i, t_i) \ , \qquad \mathbf{k}_2 = h\mathbf{f}(\mathbf{y}_i + \mathbf{k}_1/2, t_i + h/2) \ . \tag{6.46}$$

e.5 Pseduocode to implement 4th-order Runge-Kutta for a system of N coupled, linear differential equations

The fourth-order Runge-Kutta vector formulas are $(i = 0, \cdots, N-1)$:

$$\mathbf{y}_{i+1} \approx \mathbf{y}_i + \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4) \ , \tag{6.47}$$

with

$$\begin{aligned} &\mathbf{k}_1 = h\mathbf{f}(\mathbf{y}_i, t_i) \ ; && \mathbf{k}_2 = h\mathbf{f}(\underbrace{\mathbf{y}_i + \mathbf{k}_1/2}_{\alpha_1}, t_i + h/2) \ ; \\ &\mathbf{k}_3 = h\mathbf{f}(\underbrace{\mathbf{y}_i + \mathbf{k}_2/2}_{\alpha_2}, t_i + h/2) \ ; && \mathbf{k}_4 = h\mathbf{f}(\underbrace{\mathbf{y}_i + \mathbf{k}_3}_{\alpha_3}, t_i + h) \ . \end{aligned} \tag{6.48}$$

The procedure for a function to take us from the vector $\mathbf{y}_i$ of length N (which is input) to $\mathbf{y}_{i+1}$ (which is returned), is

1. calculate $\mathbf{k}_1$ and α_1 ;
2. calculate $\mathbf{k}_2$ and α_2 ;
3. calculate $\mathbf{k}_3$ and α_3 ;
4. calculate $\mathbf{k}_4$;
5. calculate $\mathbf{y}_{i+1}$,

which therefore requires five loops (since we have to calculate the N elements of each vector at every step). This is what is implemented in the `runge4` function (except that $\alpha_{1,2,3}$ are called $y_{1,2,3}$).

f. 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.]