7.   6810 Session 7

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

In Session 6 we looked at a re-write of the `eigen_tridiagonal.cpp` code to make use of C++ classes. Here we review 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.

- **Choosing Classes.** There is no fixed rule for choosing classes; there is generally more than one reasonable choice. Here we'll follow the philosophy of evolving our code with minimal changes. So I did not introduce a general matrix class or have a class for the mesh or the potential, but simply introduced one for the Hamiltonian. As noted above, the basic steps are to define the elements of the Hamiltonian element-by-element, diagonalize it, and then access the eigenvalues and corresponding eigenvectors.

- **How was it done?**

  - I stepped through the code and cut every GSL function and pasted them in a new file I called `GslHamiltonian.cpp`, without worrying yet about the order or the syntax of the class.

  - Where the allocation statements used to be, I put in a single declaration of a Hamiltonian:
    ` Hamiltonian my_hamiltonian(dimension);`
    Since the dimension of the Hamiltonian was the only important defining parameter (at least in the present version), it is the only argument for creating the Hamiltonian.

  - For each former operation (namely diagonalizing or getting eigenvalues and eigenvectors), I introduced a Hamiltonian class function (often called a "method"). These were named `set_element`, `find_eigenstuff` (I have no idea why I didn't call this "diagonalize" instead), `get_eigenvalue`, and `get_eigenvector`. The arguments are basically the same

as for the GSL functions in the original code. Methods for a class are invoked like elements of a structure, using a period following the class name, such as:
`my_hamiltonian.set_element(i,j,Hij);`

− Next I turned to `GslHamiltonian.cpp`. There are two include files, one for the GSL header file and the other for the as-yet unwritten `GslHamiltonian.h` file; every class file will have a corresponding header file. It is this header file that has the only definitions needed by the main program (so it is also included in `eigen_tridiagonal_class.cpp`).

− Each function is specified in the class file with a name starting with the class name and two colons (e.g., `Hamiltonian::`). There are several special class functions, two of which we use here. These are the constructor and destructor.

− The constructor function, which always has the same name as the class (so here it is `Hamiltonian::Hamiltonian`), is called whenever a new instance of the class is declared (i.e., with the `Hamiltonian my_hamiltonian(dimension)` statement). It can take any number of arguments. Note that it doesn't have a return value or a function type. Since we need to allocate space for the various GSL vectors, matrices, and workspace every time we have a new Hamiltonian, these statements go here. But the declarations of the type go in the `GslHamltonian.h` header file.

− The destructor function always has the same name as the class but with a tilde in front (so here it is `Hamiltonian::~Hamiltonian`), and is called when the class goes out of scope. So if `my_hamiltonian` is declared within a loop over the dimension, it is local to that loop, and is destroyed (and then created again) with each cycle. Thus we need the destructor to have all of the statements that free up the GSL vectors and matrices.

− As you can see, each of the other functions just calls the corresponding GSL function(s). In this way, the details are hidden from the user. If we wanted to use LAPACK instead of GSL, we'd just change what is in each of the class functions with corresponding LAPACK calls, but the names would be unchanged. (We will in Session 7 use LAPACK through the Armadillo library, and we'll call the class file `ArmadilloHamiltonian.cpp`.)

− Finally, we have the header file `GslHamiltonian.h`. This has function prototypes and declarations for each of the variables. The main point is that these are divided into "public" and "private" functions and variables. The public things are available outside the class while the private things are hidden. In general, we want to keep all variables hidden ("encapsulated") — note that this is the opposite of declaring global variables! The public prototypes simply match those in the class file. Note that `dimension` is a private variable set equal to the argument of the constructor function.

− One final element of the header file is the `#ifndef GSLHAMILTONIAN_H` and `#define GSLHAMILTONIAN_H` at the beginning and the `#endif` at the end. The names here are just a convention: take the name of the file and make it uppercase to create a unique name. The statements between ifndef and endif are executed only if `GSLHAMILTONIAN_H` is undefined. Since it is defined when the header is read in for the first time, this construction ensures that the header is only included once in any distinct part of the program.

Ok, that's it for now. We'll follow up later with other applications of this class and first consider an alternative Armadillo-based implementation.

b. Other Follow-ups to Session 6 and Earlier

- **What h should we use for 4th-order Runge-Kutta if we don't know the exact answer?** [This discussion is based on Numerical Recipes, Sect. 16.2 [2].] One way is to compare stepping from $y(t)$ to $y(t + 2h)$ in one step (call the answer y_1) and in two steps (call the answer y_2). From Numerical Recipes (or some other reference) we will find that the error for Runge-Kutta with a step h goes like $h^5\phi$, where $\phi \approx y^{(5)}(t)/5!$ (which comes from a Taylor expansion). So y_1 and y_2 should be the same up to this error, except that there is one step by $2h$ in one case and two steps by h in the other:

$$\text{one step} \quad \Longrightarrow \quad y(t + 2h) \approx y_1 + (2h)^5\phi + \mathcal{O}(h^6) \,, \tag{7.1}$$

$$\text{two steps} \quad \Longrightarrow \quad y(t + 2h) \approx y_2 + 2(h)^5\phi + \mathcal{O}(h^6) \,. \tag{7.2}$$

Then $\Delta \equiv y_2 - y_1$ is a measure of the truncation error. [Note that we could use Richardson extrapolation here; do you see how?] If the desired accuracy is $\Delta_0 = \epsilon y$ (that is, ϵ is the desired *relative* error), then having used an arbitrary h_1 to get Δ_1 (that is, we make a first guess), we can determine the h_0 needed to get Δ_0 from

$$\frac{\Delta_0}{\Delta_1} = \left(\frac{h_0}{h_1}\right)^5 \qquad \Longrightarrow \qquad h_0 = \left|\frac{\Delta_0}{\Delta_1}\right|^{1/5} h_1 \,. \tag{7.3}$$

We could make this adjustment at each step, and so make the solution algorithm *adaptive*. GSL has such an adaptive routine using 4th-order Runge-Kutta.

c. Forced, Nonlinear Oscillator

In Session 7, we'll apply our differential equation solving routines to a basic problem: a forced (that is, driven), oscillator. We'll give an overview here based on discussions in the Landau/Paez text [1].

The force we'll use is a generalization of Hooke's law for springs:

$$F_k(x) = \begin{cases} -kx^{p-1}, & x > 0 \,, \\ +k|x|^{p-1}, & x < 0 \,. \end{cases} \tag{7.4}$$

The parameter p is defined so the potential energy is $V(x) = \frac{1}{p}k|x|^p$. So $p = 2$ reduces to the usual spring. The potential energy as a function of x for a range of p is shown on the top of the next page. Notice that an infinite square-well potential is found in the $p \to \infty$ limit. (Classically, the mass bounces back and forth between $x = \pm 1$, moving freely in between bounces.)

We'll add to this an external, time-dependent force $F_{\text{ext}}(x, t)$. Newton's second law gives us a differential equation for $x(t)$:

$$F = Ma \quad \Longrightarrow \quad \frac{d^2x}{dt^2} = \frac{dv}{dt} = \frac{1}{M}\left[F_k(x) + F_{\text{ext}}(x, t)\right]; \qquad v \equiv \frac{dx}{dt} \,. \tag{7.5}$$
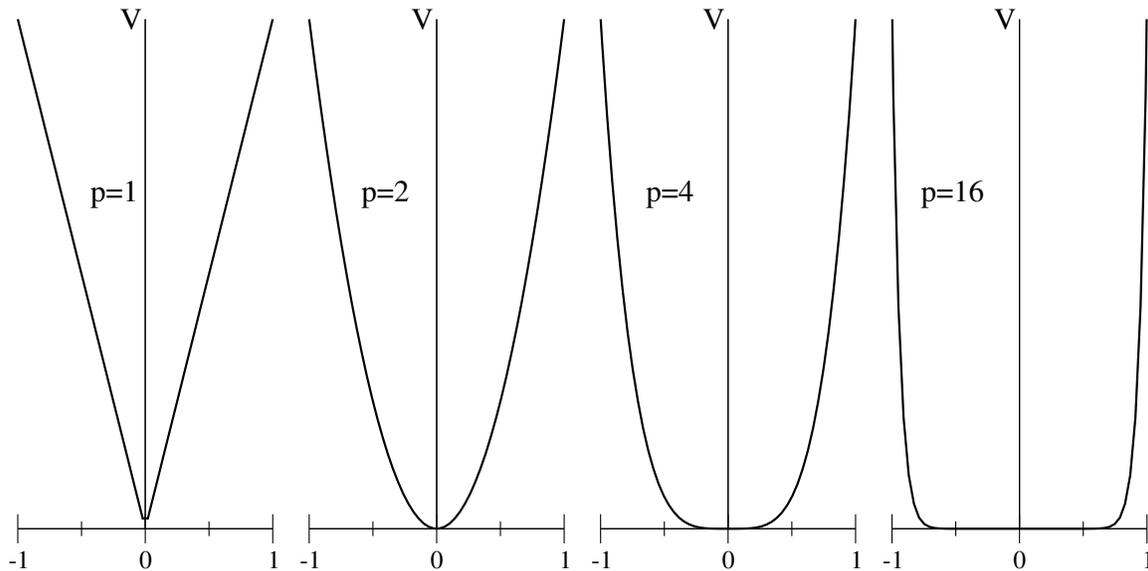
To use the Runge-Kutta routine in `diffeq_routines.cpp`, we define the $y^{(i)}$'s as

$$y^{(0)}(t) = x(t) , \qquad y^{(1)}(t) = v(t) , \tag{7.6}$$

so that the coupled equations are

$$\frac{dy^{(0)}}{dt} = y^{(1)} , \qquad \frac{dy^{(1)}}{dt} = \frac{1}{M}F_k(x) + \frac{1}{M}F_{\text{ext}}(x,t) \tag{7.7}$$

(with the "rhs" functions defined by the right sides of these equations). Since we have two equations, we need two initial conditions, which we take as the initial position x_0 and initial velocity v_0:

$$y^{(0)}(t=0) = x_0 , \qquad y^{(1)}(t=0) = v_0 . \tag{7.8}$$

Check the `diffeq_oscillations.cpp` code to see how this works in practice.

Some things to note:

- "Anharmonic" oscillator means $p \neq 2$. What is special about a harmonic oscillator? (E.g., what is the relation between the amplitude and the period when $p = 2$? Is this true for $p \neq 2$?)

- The total energy is the sum of the kinetic energy (KE) and potential energy (PE):

$$E(t) = \text{KE}(t) + \text{PE}(t) = \frac{1}{2}Mv(t)^2 + V[x(t)] . \tag{7.9}$$

What do you expect the plot of this to be like if undriven (i.e., no external force)? If driven with a sinusoidal external force? What if the oscillator is damped (consider both cases)?

- In the undamped, undriven case, the total energy is conserved, but the kinetic and potential energies are individually time dependent. However, when they are each averaged over a period (or many periods), they satisfy a *virial theorem*:

$$\langle \text{KE} \rangle = \frac{p}{2}\langle \text{PE} \rangle , \tag{7.10}$$

where the $\langle\rangle$'s denote a time average. Do you know how to derive this? [Hint: Consider the time average over a period of the quantity Mvx.] How would you check numerically if the virial theorem is satisfied?

d. Damped Oscillations

In the real world of macroscopic physical systems, there is friction, which will damp oscillations. Landau and Paez present three models for friction: *static*, *kinetic* (or sliding), and *viscous*, with the associated force laws [1]:

$$F_f \ \leq \ -\mu_s N \,, \qquad \text{static}\,, \tag{7.11}$$

$$F_f \ = \ -\mu_k N \frac{v}{|v|} \,, \quad \text{kinetic}\,, \tag{7.12}$$

$$F_f \ = \ -bv \,, \qquad \text{viscous}\,, \tag{7.13}$$

where N is the normal force, the μ's are coefficients of static and kinetic friction, b is a damping parameter, and v is the velocity. Note that the $v/|v|$ piece of the kinetic friction is just ± 1, which ensures that the friction opposes the motion.

There are other types of viscous friction (e.g., other powers of v can appear) under different circumstances, but we'll use the one from Eq. (7.13) in Session 7. When included in a harmonic or slightly anharmonic oscillator, we can identify three regimes (look for them!) based on how the damping parameter to mass ratio b/M compares to the natural (undamped, undriven) frequency ω_0:

1. **Underdamped:** The solution oscillates within bounds that decay exponentially (i.e., there is an *envelope*) when $\frac{b}{2M} < \omega_0$.

2. **Critically damped:** The solution goes directly to the equilibrium position without any oscillations when $\frac{b}{2M} = \omega_0$.

3. **Overdamped:** The solution decays slowly, reaching the equilibrium position only after an infinite time (in principle), when $\frac{b}{2M} > \omega_0$. How might you distinguish this from critically damped behavior?

Can you think of applications to real life where you would like the oscillations to be critically damped (or, really, just slight underdamped)?

e. Resonance and Beating

If we have a sinusoidal, external driving force

$$F_{\text{ext}}(t) = F_0 \sin \omega t \,, \tag{7.14}$$

then we can look for *resonance* and *beating* in either harmonic or anharmonic oscillations. When a *harmonic* oscillator ($p = 2$) that has natural frequency ω_0 is driven at $\omega = \omega_0$, the system resonates.

If there is no damping, the amplitude of the oscillation increases with time indefinitely (well, until the approximation of harmonic forces breaks down). How does the nature of the resonance change when $p \neq 2$? (You will explore this!)

If the magnitude of the external driving force F_0 is large compared to the natural restoring force of the system, then we observe *mode locking*, where the driving force dominates and (after a transient period) the system oscillates in phase with the driving frequency. If the external driving force is comparable in magnitude, then we can observe beating when the driving frequency is close to (but not equal to) the natural frequency of the system. The characteristic beating behavior follows from trignometric identities applied to the sum of the natural response at ω_0 and the driven response at ω:

$$x \approx x_0 \sin \omega t + x_0 \sin \omega_0 t = 2x_0 \cos\left(\frac{\omega - \omega_0}{2}t\right) \sin\left(\frac{\omega + \omega_0}{2}t\right) . \tag{7.15}$$

When $\omega \approx \omega_0$, the cosine term varies slowly, so the system appears to oscillate at an average frequency $(\omega + \omega_0)/2$ with a slowly varying amplitude at frequency $(\omega - \omega_0)/2$.

f.   Phase-Space Plots

The classical solution to the one-dimensional oscillator is conventionally given as the position $x(t)$ and the velocity $v(t)$ as functions of time. A useful visualization of the motion is obtained by plotting $v(t)$ versus $x(t)$, which is called a *phase-space plot*. Here are some questions you can ask using phase-space plots of the oscillator studied in this session:

- If we followed points being plotted as time increases, is the motion clockwise or counter-clockwise? Is this a general result?

- What shape(s) are the closed curves for a harmonic $p = 2$ oscillator?

- What differences are there in the shape of the orbits for anharmonic oscillations? (E.g., are they like ellipses?) Do the initial conditions matter?

There are various characteristic structures to be found in phase space plots. These include [1]:

- **Limit cycles.** These are closed ellipse-like figures, with frequencies greater than ω_0. For this motion, the average energy input to the system (after transients die out) is balanced by the average energy dissipated by friction. They do not have to be stable and may jump to another limit cycle.

- **Mode locking.** As mentioned earlier, mode locking occurs when the driving force overwhelms the natural oscillations, such that there is steady-state motion at the driving frequency.

- **Predictable attractors.** These are well-defined periodic patterns that are not highly sensitive to initial conditions. The system settles into these orbits (which could be fixed points or limit cycles) or returns to them regularly. If at some time you are near one, you will be "attracted" to it as your orbit evolves.

- **Strange attractors.** These are well-defined semiperiodic patterns that are very sensitive to initial conditions. They are fractal chaotic, in contrast to predicaable attractors.
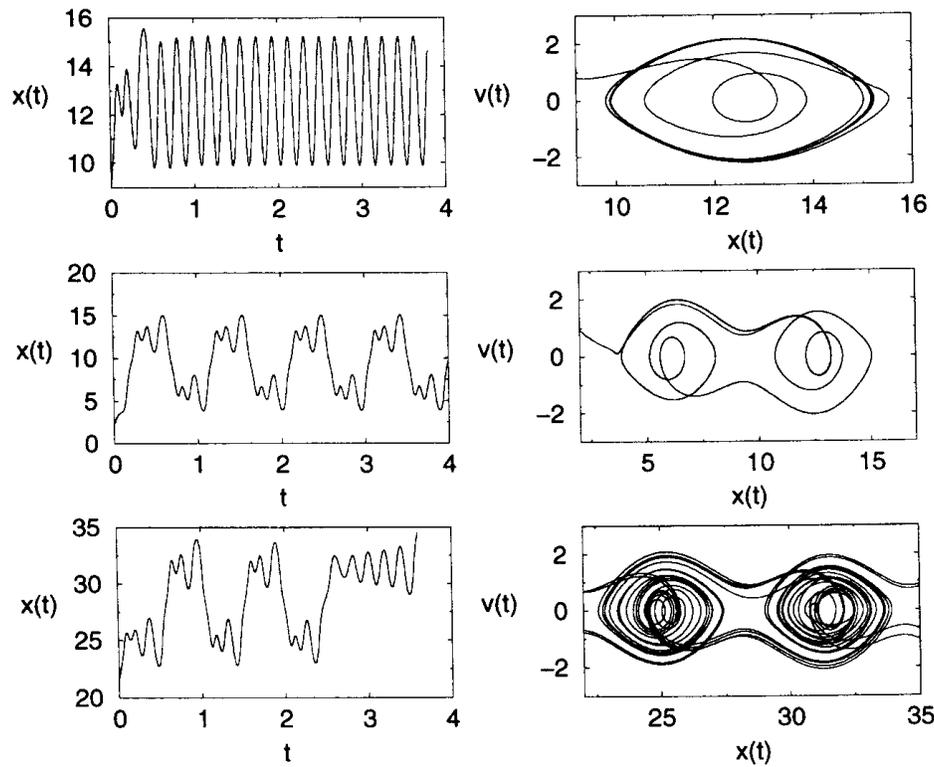
Figure 1: Plots for a chaotic pendulum with $\omega_0 = 1$, $\alpha = 0.2$, $f = 0.52$, and $\omega = 0.666$ with different initial conditions. The angular position is $x(t)$ and the angular velocity is $v(t)$. From Ref. [1].

- **Chaotic paths.** These are regions of phase space that fill in as continuous bands rather than just lines. There is both some structure to the general motion but at the same time very complicated $x(t)$ behavior.

We won't have time to look in detail at these different possibilities, but will just touch upon some characteristic examples.

In Fig. 1, which is from Ref. [1], there are plots of the time evolution of a damped, driven pendulum (the angular position is denoted x and the angular velolity v). We will study this system in Session 8, where we will explore some features of the chaotic behavior that is hinted at in the figure.

g.   References

[1] R.H. Landau and M.J. Paez, *Computational Physics: Problem Solving with Computers* (Wiley-Interscience, 1997).

[2] W. Press *et al.*, *Numerical Recipes in C* (Cambridge, 1992). Individual chapters are available online from http://www.nrbook.com/a/. There are also versions for Fortran and C++.