8. 6810 Session 8

a. Follow-up to Richardson Extrapolation

Let's consider a hypothetical situation. Suppose I had a function called `f` that calculated a quantity of interest (e.g., an integral, a derivative, a special function) given a parameter `h` (which could be a mesh spacing, time step, or something else entirely). Suppose also that I knew that the error in $f(h)$ was a constant times h^n (plus higher-order corrections). How can I use Richardson extrapolation to eliminate the h^n error without knowing anything else? Answer: simply calculate at two different h values and use the results to eliminate the h^n error. For example, calculate at h and $h/2$ and call a_1 the result of $f(h)$ and a_2 the result of $f(h/2)$. Then

$$\begin{aligned} a_1 &= \text{exact} + \alpha h^n + \cdots \\ a_2 &= \text{exact} + \alpha (h/2)^n + \cdots = \text{exact} + 2^{-n}\alpha h^n \end{aligned}$$

We can easily eliminate the h^n term and solve for "exact":

$$\text{exact} = \frac{2^n a_2 - a_1}{2^n - 1} + \cdots \ .$$

The updated version of `extrap_diff` uses this with $n = 2$; i.e., the improved derivative is calculated as:

```
( 4.*central_diff(h/2.) - central_diff(h) )/3.
```

since the `central_diff` error is h^2. You can do likewise using `extrap_diff` to eliminate the h^4 error!

b. Follow-up to Session 7: Damping

Let's review the harmonic oscillator (the $p = 2$ case from Session 7) and the mathematical conditions for underdamped, critically damped, and overdamped motion. At the same time, you should look at the Session 8 handout showing phase-space plots for these cases.

We start with the differential equation for an undriven harmonic oscillator with a damping force proportional to the velocity:

$$F = Ma \quad \Longrightarrow \quad \frac{d^2x}{dt^2} = -\omega_0^2 x - \frac{b}{M}\frac{dx}{dt} \ , \tag{8.1}$$

or

$$\frac{d^2x}{dt^2} + \frac{b}{M}\frac{dx}{dt} + \omega_0^2 x = 0 \ . \tag{8.2}$$

Since this is a *homogeneous* equation, we look for solutions of the form $x(t) = Ae^{i\omega t}$, where ω could be positive or negative, real or complex. Substitute $x(t)$ into Eq. (8.2):

$$(i\omega)^2 Ae^{i\omega t} + \frac{b}{M}(i\omega)Ae^{i\omega t} + \omega_0^2 Ae^{i\omega t} = 0 \ . \tag{8.3}$$

The factor $Ae^{i\omega t}$ never vanishes, so we can divide it out, leaving

$$\omega^2 - \frac{ib}{M}\omega - \omega_0^2 = 0 \ . \tag{8.4}$$

In general, this quadratic equation has two (complex) solutions, which will lead to the two independent solutions we expect for a 2nd-order differential equation. We'll denote the solutions ω_+ and ω_-, where

$$\omega_\pm = \frac{\frac{ib}{M} \pm \sqrt{-\frac{b^2}{M^2} + 4\omega_0^2}}{2} = \frac{ib}{2M} \pm \sqrt{\omega_0^2 - \left(\frac{b}{2M}\right)^2} \ . \tag{8.5}$$

Now let's consider cases. For simplicity, we'll work with units in which $M = 1$ and the spring constant $k = 1$, which means $\omega_0 = \sqrt{k/M} = 1$. For initial conditions, let's take $x_0 = 0$, $v_0 = 1$ (so if we're thinking of a spring, it starts at the equilibrium position with a nonzero speed).

1. $b = 0 \implies$ "undamped"
   From Eq. (8.5), $\omega_\pm = \pm\omega_0$. A general solution is therefore

   $$x(t) = Ae^{i\omega_0 t} + Be^{-i\omega_0 t} \tag{8.6}$$

   and the initial conditions $x_0 = 0$ and $v_0 = 1$ yield

   $$A + B = 0 \ , \qquad i\omega_0(A - B) = 1 \ , \tag{8.7}$$

   or

   $$A = \frac{1}{2i}\frac{1}{\omega_0} \ , \qquad B = -\frac{1}{2i}\frac{1}{\omega_0} \ . \tag{8.8}$$

   Recalling that

   $$\sin x = \frac{e^{ix} - e^{-ix}}{2i} \qquad \text{and} \qquad \cos x = \frac{e^{ix} + e^{-ix}}{2} \ , \tag{8.9}$$

   we obtain the solutions

   $$x(t) = \frac{1}{\omega_0}\sin(\omega_0 t) \ , \qquad v(t) = \cos(\omega_0 t) \ . \tag{8.10}$$

   The properties of trig functions tell us immediately that the phase-space plot of v vs. x will have the equation

   $$\omega_0^2 x^2 + v^2 = 1 \ , \tag{8.11}$$

   which is an ellipse.

2. $\frac{b}{2M} < \omega_0 \implies$ "underdamped"
   Now we have a real and an imaginary part to $e^{i\omega t}$. The general solution is:

   $$x(t) = e^{-\frac{b}{2M}t}\left(Ae^{i\sqrt{\omega_0^2 - (b/2M)^2}\,t} + Be^{-i\sqrt{\omega_0^2 - (b/2M)^2}\,t}\right) \ . \tag{8.12}$$

   The decaying exponential on the outside will form an *envelope* $\pm e^{-\frac{b}{2M}t}$ about the oscillations. We can read off the oscillation frequency (if the oscillations die slowly) as

   $$\omega = \sqrt{\omega_0^2 - (b/2M)^2} \ . \tag{8.13}$$

The phase-space trajectory is a *spiral* into a *fixed point* at the origin (that is, it eventually stops at the equilibrium point!). What would a plot of the time dependence look like? (Predict and then look at the corresponding plot on the handout.)

3. $\frac{b}{2M} = \omega_0 \Longrightarrow$ "critically damped"
   The solution for $x(t)$ is a degenerate form now, which does not look like the previous forms since the oscillating parts vanish. Rather, we have a general solution

$$x(t) = (A + Bt)e^{-\frac{b}{2M}t} \ . \tag{8.14}$$

   For the initial conditions $x_0 = 0$ and $v_0 = 1$, we find

$$x(t) = te^{-\frac{b}{2M}t} \ , v(t) = e^{-\frac{b}{2M}t}\Big(1 - \frac{b}{2M}t\Big) \ . \tag{8.15}$$

   (See the plots to help visualize these.) Note that $x(t)$ is always positive but $v(t)$ goes negative. What would the plots look like for initial conditions $x_0 = 1$ and $v_0 = 0$?

4. $\frac{b}{2M} > \omega_0 \Longrightarrow$ "overdamped"
   Now we find that ω is purely imaginary, which implies exponential decay. The general solution is

$$x(t) = e^{-\frac{b}{2M}t}\left(Ae^{\sqrt{\omega_0^2 - (b/2M)^2}t} + Be^{-\sqrt{\omega_0^2 - (b/2M)^2}t}\right) \ . \tag{8.16}$$

   Except for unusual values of b, the "B" term dies much faster, while the "A" term dies off slowly once B is no longer relevant because of cancellation between the exponentials. In that region, $v(t) \propto x(t)$ so the phase-space trajectory is close to a straight line. Is it possible for $v(t)$ to go to zero without turning negative?

Try to reproduce these different regimes in real time using `diffeq_pendulum.cpp`. When we now add a *driving* force

$$\frac{f_{\text{ext}}}{M}\cos(\omega_{\text{ext}}t) \tag{8.17}$$

(we've chosen the phase to be zero at $t = 0$), then the above solutions to the *homogeneous* equation ("transients") are added to a specific solution to get the general solution. The transients die out (hence the name!), leaving the specific solution in the long-time limit.

c. Gnuplot Stuff

c.1 A Common Error with Continuation Lines

If you have a gnuplot plot file `my_plotfile.plt` and you get an error message like

```
"my_plotfile.plt", line 27: invalid character
```

when you try to load it, the most likely problem is that there is at least one blank space after a continuation character \. That \ has to be the absolute last thing on a line or you'll get this error. Check it out!

c.2 Multiple Plot Windows

In gnuplot under Linux, Cygwin, or (newer) Macs the "terminal type" corresponding to the screen is called "x11" (for the X11 windowing system). You can have more than one plot window open at a time, however, by labeling them 1, 2, 3, and so on. For example, to plot $\sin(x)$ in one window while plotting columns 1 and 2 from the file `data.dat` in another:

```
gnuplot> set term x11 1
gnuplot> plot sin(x)
gnuplot> set term x11 2
gnuplot> plot "data.dat" using 1:2
```

By default, the windows will have titles like "Gnuplot 1" and "Gnuplot 2" on their title bars. To get window 1 to have the title "Time Dependence", use:

```
gnuplot> set term x11 1 title "Time Dependence"
gnuplot> plot sin(x)
```

You can close a windowing by first selecting it and then typing the letter q. (If you exit gnuplot entirely, all windows will close.) [Multiple windows will not work on older Macs if you have to use the "aqua" terminal type. On Lion or Mountain Lion or beyond, gnuplot uses "x11" and it works fine.]

c.3 Piping to Gnuplot with GnuplotPipe

In the `diffeq_pendulum.cpp` program, the output is sent to gnuplot more-or-less directly. The actual execution is not very elegant and may not work perfectly all the time. The GnuplotPipe class distributed in Session 8 is the first pass at a rewriting of the `gnuplot_pipe.c` C code (which was adapted from a C code found on the web) into a C++ class. This version has limited functionality and has too much direct translation of C into C++ (anyone up for making a second pass?). But this does mean that it can be a good example of how to evolve from C-style (or Fortran-77 style) code to more object-based programming. Here we'll give another brief overview of some features of C++ classes using it as an example.

Let's start with the `GnuplotPipe.h` header file. The header file conventionally has the same name as the class, with `.h` (or sometimes other endings like `.hpp`) appended. It is also conventional to name classes with capitalized words concatenated (e.g., as opposed to `gnuplot_pipe`). Some things to note about the header file:

- It will be included (using `#include "GnuplotPipe.h"` or with an appropriate path to where it is located) in `GnuplotPipe.cpp` and in any file that will use the class. Everything that external programs need to know is included here.
- The lines:

  ```
  #ifndef GNUPLOTPIPE_H
  #define GNUPLOTPIPE_H
  ```

at the beginning and the `#endif` at the end are a standard device to keep the file from being included multiple times. The first line checks if `GNUPLOTPIPE_H` is already defined (this name follows another convention; it is arbitrary but needs to be unique). If it is, nothing further is done; if it isn't, it is defined and the rest of the file is compiled.

- The class definition is like a generalization of the structures we've seen before. They can have both data (like `xlabel` and `xmin`) *and* functions (like `set_filename`); these are often referred to as "data members" and "member functions". [Member functions are also known as "methods".] The data and functions are divided here into "public" and "private" categories (another category, called "protected" lies in between and is relevant when we have subclasses). External programs can access public data and functions, but not private ones.
- In general, we want to hide the data away from prying external eyes, so it should be declared as private. Any purely internal functions should also be private (we don't happen to have any here). The "accessor functions" are somewhat trivial functions that let an outside user retrieve or change the value of private data. So, for example, instead of letting the user just change the value of `xmin`, he has to call the member function `set_xmin`. This might seem like an unnecessary layer of bureaucracy, but it is really a key feature of the approach, because you are able to constrain what the external user is able to do and hide implementation features from him/her.
- Every class has a "constructor" and a "destructor" function, which are invoked when an object of that type is allocated and deallocated, respectively (the latter might only happen when the program terminates). [Note: There should also be a "copy constructor", which is used to make a copy of a class. If omitted, a default version is used. This can lead to problems if you have dynamically allocated memory.]

The header file told us what functions are available as well as the types of the internal variables. The functions themselves are defined in the `GnuplotPipe.cpp` file.

- We put `GnuplotPipe::` in front of each function name. This indicates that the function is in the `GnuplotPipe` namespace (just as things like `cout` are in the `std` namespace).
- Any of the functions can directly access (i.e., get or change the values) any of the private data. So the data act like global variables within the class and we don't have to pass them back and forth!
- The constructor function, which always has the exact same name as the class name, is called in `diffeq_pendulum.cpp` with the command:

  ```
  GnuplotPipe myPipe;
  ```

  This creates a `GnuplotPipe` object called `myPipe` (just like `double my_double` creates a `double` called `my_double`). When it is created, the private variables for `myPipe` are set equal to their defaults according to the constructor (e.g., `delay` is set to 10000).
- The destructor function, which has the same name as the class name but with a $\sim$ in front, doesn't do anything at present.

- The rest of the functions do similar things to our standard functions. Again, we don't need to pass around the private data, so the only arguments to the functions come from outside the class (e.g., the x and y coordinates passed to `plot` or `plot2`).
- The only special feature is the `popen` command in `init` (and its counterpart `pclose` in `finish`). This is just like opening a file to which we can print, except that we're printing to the gnuplot process. At present we use the C-style commands like `fopen` and `fprintf`; we'll change these to C++ stream output in the future.
- To call a public function, we use the dot notation, e.g.,

  ```
  myPipe.set_xlabel ("theta");
  ```

  If I had a second `GnuplotPipe` object called `myPipe2`, we would set its `xlabel` using

  ```
  myPipe2.set_xlabel ("alpha");
  ```

The C++ class in `GnuplotPipe.cpp` was adapted from a code found on the web. To see how to use it, look at the `GnuplotPipe.h` header file. There are many improvements possible!

d. C++ Strings and Things

When we have names or a phrase or a sentence in C, we store it as a *char* type, or actually a pointer to an array of a fixed length. For example, the declaration

```
char filename[30];
```

In C++, we can use C-style string and associated commands (such as `sprintf` or `strcomp`) *or* we can use C++ strings. I recommend C++ strings for several reasons:

- they can be any length, and adjust automagically (i.e., we do *not* declare the length);
- they work with `==` and `+` as you would hope and expect;
- they are overall safer.

Here's a quick guide to the use of C++ strings, with the goal of constructing filenames. The `filnename_test.cpp` program in Session 8 illustrates the summary given here.

Action	**How to do it**
include file	`#include <string>`
declaring	`string my_name;` `string my_name = "Furnstahl";`
assigning	`my_name = "Smith";`
comparing	`if (my_name == "Furnstahl") {` *(do something)* `}`

joining	`string my_full_name;`
	`first_name = "Dick", last_name = "Furnstahl";`
	`my_full_name = first_name + last_name;`

printing	`cout << "My name is " << my_full_name << endl;`
converting to char	`my_full_name.c_str();`

Suppose we want to open a file with the name stored in the string `my_full_name`. Here's how:

```
ofstream my_file;   // declare the output file stream
file.open (my_full_name.c_str());   // notice the use of c.str()
```

If we tried `file.open(my_full_name);` we would get *many* errors. Now suppose we want to create a filename with the current value of the integer `i` as part of the name. We do this with an "output string stream" or `ostringstream`. Here's what it looks like:

```
int i = 5;   // define our integer
ostringstream my_filename_stream;         // declare the output string stream
my_filename_stream << "output_" << i;     // just like using cout
string my_filename = my_filename_stream.str();  // convert stream to a string
file.open (my_filename.c_str());   // open the file, again using c.str()
```

You could also skip defining the string `my_filename` and open the file using:

```
file.open (my_filename_stream.str().c_str());
```

e. Damped Driven Pendulum

There is an apt quote from physicist/mathematician Stanislaus Ulam about nonlinear phenomena:

> "Calling the subject nonlinear dynamics is like calling zoology 'non-elephant studies'."

Indeed, most macroscopic real-world phenomena is *nonlinear*, unlike the impression you might receive from typical physics curricula, which emphasize linear problems.

In Session 8, we consider another simple yet rich example of nonlinear dynamics: the damped driven pendulum. We have in mind a physical pendulum with moment of inertia I for which the dependent variable is the angle θ. It is subject to gravity, a damping force proportional to $d\theta/dt \equiv \dot{\theta}$, and an external periodic driving torque. Newton's second law for the torque can be rearranged to give the differential equation:

$$\frac{d^2\theta}{dt^2} + \alpha\frac{d\theta}{dt} + \omega_0^2 \sin\theta = f_{\rm ext}\cos(\omega_{\rm ext}t) \ . \tag{8.18}$$

The presence of $\sin\theta$ rather than θ makes this problem inherently nonlinear.

To study this system, we'll look at plots of θ versus t and the phase-space plot $\dot{\theta}$ vs. θ. Under what conditions will these look like the results for the $p = 2$ harmonic oscillator from Session 7? Try to decide what range of θ_0 gives harmonic behavior, given $\dot{\theta}_0 = 0$.

Let's think about *chaos*. Consider the following brief discussion as a teaser. Here are some characteristics of chaos:

- The system does not repeat past behavior (cf. periodic behavior).

- An uncertainty (or variation) in initial conditions grows *exponentially* (rather than linearly) in time. The consequence is that the system is deterministic (as opposed to having a random component) but not predictable, since there is a finite precision in specifying the initial conditions (e.g., think about round-off error!).
- The system has a distributed power spectrum (more next time!).

The following are necessary conditions for chaotic behavior:

a) The system has at least *three* independent dynamical variables. That is, the system can be written as

$$\frac{dy_0}{dt} = F_0(y_0, \cdots, y_n) \ , \tag{8.19}$$
$$\frac{dy_1}{dt} = F_0(y_0, \cdots, y_n) \ , \tag{8.20}$$
$$\vdots \tag{8.21}$$
$$\frac{dy_n}{dt} = F_0(y_0, \cdots, y_n) \ , \tag{8.22}$$
$$\tag{8.23}$$

with $n \geq 3$.

b) The equations of motion contain nonlinear term(s) that couple several of the variables.

You might not think that our pendulum example qualifies, since there only seem to be two independent dynamical variables, θ and $\omega \equiv \dot{\theta}$. We find the third by introducing ϕ as

$$\phi = \omega_{\rm ext} t \quad \Longrightarrow \quad \frac{d\phi}{dt} = \omega_{\rm ext} \ . \tag{8.24}$$

Thus, the three necessary equations are

$$\frac{d\theta}{dt} = \omega \ , \tag{8.25}$$
$$\frac{d\omega}{dt} = -\alpha\omega - \omega_0^2 \sin\theta - f_{\rm ext} \cos\phi \ , \tag{8.26}$$
$$\frac{d\phi}{dt} = \omega_{\rm ext} \ . \tag{8.27}$$

Now we satisfy a) with θ, ω, and ϕ, and we satisfy b) since the $\sin\theta$ and $\cos\phi$ terms couple the equations nonlinearly. So we should be able to find chaos!

f. Armadillo Linear Algebra Library

To quote from the Armadillo webpage at http://arma.sourceforge.net/, "Armadillo is a high quality C++ linear algebra library, aiming towards a good balance between speed and ease of use." It is open source (free!), runs on all platforms (Linux-based, Mac OS X, and Windows), and is

actively under development (so features are added and bugs are fixed). Armadillo lets us define and manipulate matrices and vectors, and provides a convenient interface to LAPACK or similar libraries. In the Session 8 activities we'll use it as an example of setting up a private version of a library and an example of an alternative to the GSL implementation of the Hamiltonian class.

g. 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.]

[2] M. Hjorth-Jensen, *Computational Physics* (2013). 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.