

9. 780.20 Session 9

a. Follow-ups to Session 8 and earlier

- **“Visualization of the Pendulum’s Dynamics.”** The handout with this heading shows the progression of phase-space trajectories and Poincaré sections for a pendulum as the value of the driving force amplitude is varied. The idea of a Poincaré section is to plot a point in phase space once every period of the external force, $2\pi/(\text{external frequency})$. The resulting pattern gives information about the periodicity of the signal (or indicates chaos). The green points in the gnuplot plots from Session 8 form a Poincaré section. You will also find the Mathematica version in the `nonlinear.nb` notebook in Session 9. The figure caption to Fig. 3.4 in the handout indicates the key characteristics of each plot.

Figures 3.8 and 3.9 show another analysis tool: the *power spectrum*. The power spectrum is found by taking a Fourier transform (FFT) of the “signal” (e.g., the angle of the pendulum as a function of time). In Fig. 3.8, we see discrete frequencies, while in Fig. 3.9 we see a continuous distribution, which is an indicator of chaos. In the `nonlinear.nb` and `pendulum.nb` Mathematica notebooks, you’ll be able to create power spectra. Try to find chaos!

- **Follow-up to Streams.** In the `filename_test.cpp` code, we created stringstream objects and used them to create filenames to open. In doing so, we had an intermediate step where we defined a string `filename2`:

```
// next, create a string with the stringstream class
ostringstream filename_stream; // declare a stringstream object

// you can load the string stream just like output streams
filename_stream << "test_stream" << i << ".out";
// use .str() to convert to a string
string filename2 = filename_stream.str();

ofstream file2; // print to file2, which is named by filename2
file2.open (filename2.c_str()); // use .c_str() to convert to a char *
```

We used the intermediate string `filename2` because we wanted to first convert `filename_stream` to a string, and then convert the string to a C-style string (using `.c_str()`). We can skip the intermediate step and do it all at once with `.str().c_str()`:

```
// next, create a string with the stringstream class
ostringstream filename_stream; // declare a stringstream object

// you can load the string stream just like output streams
filename_stream << "test_stream" << i << ".out";

ofstream file2; // print to file2, which is named by filename_stream
file2.open (filename_stream.str().c_str()); // convert to a char *
```

- **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 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

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

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

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

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. Background on GnuplotPipe

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 first version has limited functionality and has too much direct translation of C into C++. But this means 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 a brief overview of some features of C++ classes using it as an example. We'll revisit the `GnuplotPipe` class later to see how it has evolved further.

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) of 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 `~` 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");
```

c. Adaptive Differential Equation Solvers

In general, it's best to adjust the step size h in solving a differential equation because the optimal size will vary for different parts of the function. A routine that varies the step size automatically to keep the local error under control is called "adaptive". We'll try out the adaptive routines from GSL in this session with the `ode_test.cpp` code. This code is based on the example included with the GSL reference manual.

The test program will solve the Van der Pol oscillator, which is defined by the equation

$$\frac{d^2x}{dt^2} + \mu \frac{dx}{dt}(x^2 - 1) + x = 0, \quad (9.1)$$

where μ is the only parameter. To specify a solution, we give initial values for x and $v = dx/dt$ (which we call x_0 and v_0 , respectively). We'll take $\mu = 2$ with a variety of initial conditions.

Choosing different initial conditions means starting at different points in a phase space plot (v vs. x). In Session 9, you'll start at three different initial conditions. You should find that the phase-space trajectories all end up on the same curve. (Does this work for *any* initial conditions?) According to the Landau-Paez discussion, this is called an "isolated attractor" (the phase-space trajectories are *attracted* to the universal curve).

This is just one example. You are invited to explore further!

d. Interpolation vs. Data Fitting

In this session, we will look at *interpolation*, which is sometimes confused with *data fitting*. Our basic interpolation problem will be to take a table of function values $\{x_i, y_i = f(x_i)\}$ for which an analytic form is not available, and estimate $f(x)$ for $x \neq x_i$ (for interpolation, x should be *between* two of the x_i 's, otherwise it is extrapolation, which is much harder to do with controlled errors). Here are some possible applications for interpolation:

- we want to calculate $\int_a^b [f(x)]^2 dx$ using Gaussian quadrature;
- we want the derivative (or 2nd derivative) of the tabulated function f ;
- we want to solve an ode involving f using a GSL routine.

There is an important assumption in all of these applications: the values of f should not be *noisy* (although they invariably have round-off errors). An example of a noisy function f would be experimental data. If you want to interpolate a noisy function, it's usually best to first fit a curve to the data and then to interpolate on the fit function.

If we assume the values y_i are not noisy, then between x_i and x_{i+1} , $f(x)$ should look like a polynomial, if the points are spaced closely enough. (How close is close enough? See if you can answer this after going through this section.) But what polynomial should we use? The GSL provides several easily interchangeable interpolation methods, which you'll compare for a simple application. The handout "Using GSL Interpolation Functions" takes you through the steps needed to use the GSL interpolation functions.

Lagrange interpolation fits an $(n - 1)^{\text{th}}$ degree polynomial to $f(x_i)$ for n values of x_i . That is, the polynomial is constructed to exactly pass through those n points; if we call this polynomial $P_N(x_i)$, then

$$P_N(x_i) = f(x_i) = y_i, \quad i = 0, 1, \dots, N. \quad (9.2)$$

The formula for $N = 1$ (linear interpolation) is:

$$P_1(x) = \frac{x - x_0}{x_1 - x_0} y_1 + \frac{x - x_1}{x_0 - x_1} y_0, \quad (9.3)$$

while for $N = 3$ (a parabolic approximation) we have:

$$P_3(x) = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} y_2 + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} y_1 + \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} y_0. \quad (9.4)$$

The general formula is

$$P_N(x) = \sum_{i=0}^N \prod_{k \neq i} \frac{(x - x_k)}{(x_i - x_k)} y_i. \quad (9.5)$$

You might think that applying this to n points with $N = n$ would be optimal. It is not! You should only apply polynomial interpolation to a relatively small region (you'll see why in the Session 9 example!).

A spline function is built from polynomial pieces defined on subintervals of the entire interval for the function. The most commonly used spline is the *cubic spline*. The idea in this case is to fit $f(x)$ in the interval $[x_i, x_{i+1}]$ with a cubic polynomial

$$f_i(x) = f_i + f_i^{(1)}(x - x_i) + \frac{1}{2} f_i^{(2)}(x - x_i)^2 + \frac{1}{6} f_i^{(3)}(x - x_i)^3, \quad (9.6)$$

with the requirement that the function $f(x_i)$ is reproduced at all of the x_i and the first and second derivatives be continuous with the next interval. Thus the function and the first and second

derivatives are continuous through the entire interval. We still need boundary conditions for $f^{(2)}$ at the endpoints. A “natural” spline chooses $f^{(2)}(a) = f^{(2)}(b) = 0$. The spline coefficients are determined by a GSL library routine. In the process, we get approximations to the first and second derivatives for free.

e. Segmentation Faults

[This discussion assumes access to gdb and is based in large part on the article at <http://www.cprogramming.com/debugging/segfaults.html>.]

One of the most useful functions of a debugger is to track down the source of “Segmentation fault” error messages. (Note: on Windows you may just get a pop-up window saying that the program has encountered an error and must terminate.) In general, a segmentation fault means you have tried to access memory that doesn’t belong to your program. This is not allowed, and the segmentation fault or “segfault” is the result. One region of memory is called the “stack,” which is where local variables are stored. Another region is called the “heap,” which is dynamically allocated when your program is running is you use a “new” command in C++.

Here are some sources of segmentation faults (not a complete list!):

1. Accessing an array beyond its bounds. When you declare or allocate an array, it has a certain size. If you try to write beyond it, you will get a segmentation fault error *if* you go outside the memory assigned to your program. In this case, you can use the gdb debugger; it will identify that your program crashes at a line involving an array access and you can print the argument of the array. The tricky case is if you stay within the memory assigned, which means you will not get an error but will unpredictably write over other variables. This is not good!
2. Dereferencing NULL, uninitialized, or deleted pointers. This code illustrates the first one:

```
int *my_ptr = 0;
*my_ptr = 3;
```

This fails because the pointer is initialized to NULL (which is the same as 0) in the declaration, and then in the next line it is accessed. In gdb, this would be revealed by

```
(gdb) print my_ptr
$1 = 0x0
```

where memory address 0x0 is, in fact, NULL. To avoid problems with uninitialized pointers, you should simply set them to NULL when you declare them (unless you are using them with a “new” statement to allocated memory).

3. Using all of the stack space (such as by a recursive function); this is sometimes reported as a “stack overflow” rather than a segmentation fault.

The use of a debugger such as gdb to track down segmentation faults will be demonstrated in a later session.

f. References

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