Computational Physics (6810): Session 12

Dick Furnstahl
Nuclear Theory Group
OSU Physics Department

March 29, 2017

Various recaps and followups

Random stuff (like RNGs :)

6810 Computational Physics Endgame

- April 21 is our last class period!
- Problem set #4 has been released
  - Just a progress report on your project
  - Due April 11 in a BuckeyeBox `Project` subfolder
- Projects are due at the end of Monday, May 1
  - For graduating seniors: noon on Monday, May 1
  - Use a `Project` subfolder of your BuckeyeBox folder
  - Include codes, makefiles, plots, etc. (like for problem sets)
  - Include an explanation of how it all fits together in a separate file or in comments of the codes.
  - It is recommended that you turn in something earlier than the due date to get feedback while there's time for iteration.
- Session guides and homework will be accepted until 5pm on Thursday, April 27
  - Remember that you can always upgrade a check
  - Get in *some* version (even if incomplete) soon!

Various recaps and follow-ups

- Power spectrum (Fourier transform)
  - How is the power (energy per time) distributed among different frequencies?
  - What does a peak at zero frequency mean? How can you get rid of it?
  - See Session 11 handout for examples (to intepret!)
- Using classes makes it easier to generalize
  - e.g., GSL adaptive ODE solver for Van der Pol oscillator with different parameters
- Interpolation vs. least-squares fitting (See Session 12 notes!)
  - Summary of different interpolation issues
  - See linear least-squares fitting discussion: non-linear vs. linear; linear as a matrix problem (covariance matrix, etc.)
- Set up your code to be able to use multiple algorithms
  - for integration, interpolation, minimization, . . .
  - GSL allows this, but there are other options

Various recaps and follow-ups (cont.)

- Lessons from Session 9
  - Use GDB (or equivalent) to find segmentation faults
  - Compiler optimization is your (great!) friend, but don't turn it on until the program is debugged
  - Write first for correctness and clarity, then for speed (but only where you need speed!)
  - Why use `set` and `get` functions?
  - `rsync` is awesome!
- Python scripts to run C++ programs
  - Here: set up input, run a C++ code, and process output
  - Replace interactive input with command-line arguments
    - conventional: `main (int argc, char *argv[])`
    - `argc` is # of arguments, including program name
    - `argv[i]` is i'th argument (0 is name)
  - Python by example! Use generic scripts and adapt to needs
- Multidimensional minimization is hard!
  - Session 11: try out some GSL algorithms for *local* minima
  - Coming: global methods (we'll do simulated annealing)

Multidimensional minimization stopping

```
double tolerance = 1.e-4;
// check for convergence (state is either GSL_CONTINUE or GSL_SUCCESS)
status = gsl_multimin_test_gradient (minimizer_ptr->gradient, tolerance);
if (status == GSL_SUCCESS)   // if we're done, print out the details
{
  cout << "Minimum found at: " << endl;
}
```

- The variable `tolerance` is used in two places (bad coding!)
  - The test above is the relevant one for overall convergence
  - "line minimization" means to pick a direction and find the (one-d) minimum along that line:

    ```
    The minimum along this line occurs when the function gradient
    g and  the search direction p are orthogonal. The line
    minimization terminates when dot(p,g) < tol |p| |g|. }
    ```

- Use other criteria for stopping if derivatives (gradient) are not used
- GSL documentation is not great, but better than usual state-of-the-art!
  - check examples first! (look online for others)
  - read *everything* before proceeding (RTFM, as they say.)

From GSL manual: Multidimensional minimization

```
36.6 Stopping Criteria

A minimization procedure should stop when one of the following conditions
is true:
    A minimum has been found to within the user-specified precision.
    A user-specified maximum number of iterations has been reached.
    An error has occurred.
The handling of these conditions is under user control. The functions
below allow the user to test the precision of the current result.

int gsl_multimin_test_gradient (const gsl_vector * g, double epsabs)

  This function tests the norm of the gradient g against the absolute
  tolerance epsabs. The gradient of a multidimensional function goes
  to zero at a minimum. The test returns GSL_SUCCESS if the following
  condition is achieved,
     |g| < epsabs
  and returns GSL_CONTINUE otherwise. A suitable choice of epsabs can
  be made from the desired accuracy in the function for small variations
  in x. The relationship between these quantities is given by
  \delta f = g \delta x.

Function: int gsl_multimin_test_size (const double size, double epsabs)
```

GSL error handling (from Session 6 notes)

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.

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 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 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 message
{
  cout << " GSL error: " << gsl_strerror (status) << endl;
}
```
The program will continue to run but will warn you about problems.

Things to watch for in Session 12 and beyond

- For Monte Carlo methods, we need a set of random numbers
  - *uncorrelated* means we can't predict x_{i+1} given $x_1, x_2, \cdots, x_i$
- Doesn't have to be uniform
  - the histogram of random numbers will approach the shape of the corresponding probability distribution function (PDF)
- We'll use *pseudo*-random numbers from GSL random number generators (rng)
  - trade-offs between period and speed
  - see Session 12 notes for tests (e.g., using your eye) and Session 13 notes for pitfalls!
  - You need to *seed* the rng, or you'll get the same numbers!

Things to watch for in Session 12 and beyond (cont.)

- Random walks in two dimensions: N steps
  - Be able to derive standard deviation distance $R \approx \sqrt{N} r_{\text{rms}}$
  - Remember how to find the average of a function in an interval:

$$\langle f \rangle = \frac{1}{b-a} \int_a^b f(x)\, dx \implies \langle r^2 \rangle = \frac{1}{(b-a)^2} \int_a^b dx \int_a^b dy\, (x^2+y^2)$$

- (Crude) approximation of an integral I

$$\langle f \rangle = \frac{1}{b-a} \int_a^b f(x)\, dx \approx \frac{1}{N} \sum_{i=1}^{N} f(x_i) \implies I = \int_a^b f(x)\, dx \approx \frac{b-a}{N} \sum_{i=1}^{N} f(x_i)$$

- Better approximation: sample according to an appropriate pdf

$$I = \int_a^b dx\, f(x) = \int_a^b dx\, w(x) \frac{f(x)}{w(x)} = \left\langle \frac{f}{w} \right\rangle_w \approx \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{w(x_i)}$$