11. 6810 Session 11

a. Follow-ups to Sessions 9 and 10

- **Common Segmentation Fault.** Perhaps the most frequently occurring segmentation fault in computational physics programs comes from accessing arrays "out of bounds". When you declare an array like:

  ```
  double x[10];
  ```

  or

  ```
  const int Nmax = 10;
  double x[Nmax];
  ```

  then space is allocated for 10 doubles. They are referenced as `x[0]`, `x[1]`, ...`x[9]`, so the statement:

  ```
  x[10] = 5.;
  ```

  tries to reference an unassigned memory address, which causes a segmentation fault (if you are lucky!). If you get a segmentation fault in a statement involving arrays, always check the index against the dimension of the array (this is easy using GDB).

- **More on Optimization.** A good overview of different compiler optimizations are available is given online in the article "Compiler Optimization" in Wikipedia. Some brief comments:

  – Use `-O0` ("minus Oh zero") when debugging, to prevent the compiler from reordering your code in the course of optimizing it.

  – The actual set of optimizations included with `-O2` and `-O3` varies with the version of the g++ compiler (and with other compilers, such as the one from Intel). Check `man g++` to see exactly what is done.

  – In our session 9 example, we had a function to square two numbers declared as an `inline` function and another that was a regular function. The inline function is supposed to be as fast as just substituting the code, but they took the same time with `-O0`. That is because that option discards the `inline` keyword, so it is treated like any other function. Inlining is carried out with `-O2`, so we saw a speed difference, and then no difference again with `-O3`. In the latter case, the compiler automagically makes short functions into inline functions. (In more recent compilers, this may happen with `-O2`.)

  – An example of an optimization is `-fgcse`, which is "global common subexpression elimination", which is implemented starting with `-O2`. The Wikipedia example notes that if an expression is repeated, it could be replaced by a temporary variable to replace repeated calculation by fast look-up:

    ```
    a = b*c + g;
    d = b*c*d;
    ```

    becomes

    ```
    tmp = b*c;
    a = tmp + g;
    d = tmp*d;
    ```

    which the compiler does automatically with this option.

– Other compiler options can switch the order of nested loops to improve "locality of reference", or cut down on branching (as from `if` statements) so that the compiler can allow instructions to be fetched ahead of time. And many more!

- **Exporting Tables from Mathematica for Plotting.** To extract a table for use in a plotter such as gnuplot or xmgrace, it is convenient to use the `Export` command. For example, create the table called `myTable`:

```
myTable = Table[{x, N[Exp[x]]}, {x, 0, 100, 1}];
```

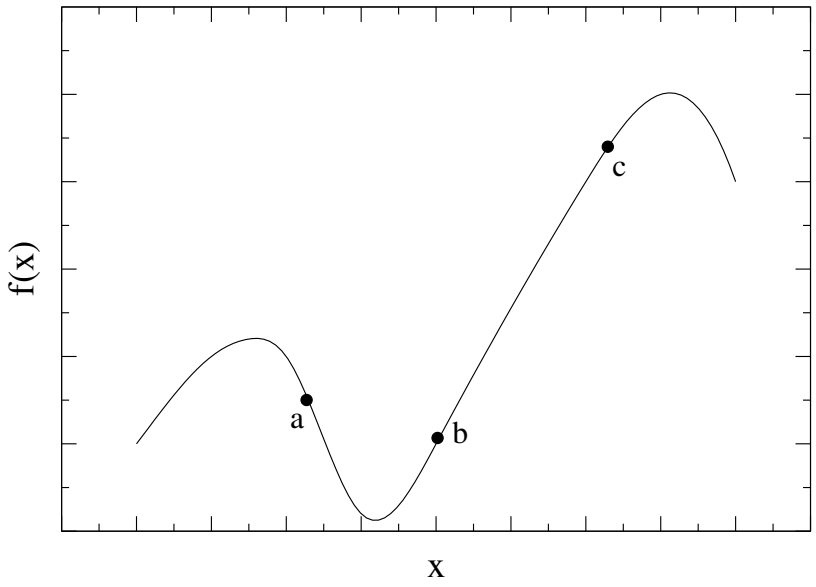
(this is just an example with some big numbers). To output this to the file `my_output_file.dat` in gnuplot or xmgrace readable form:

```
Export["my_output_file.dat", myTable, "Table"]
```

Try it!

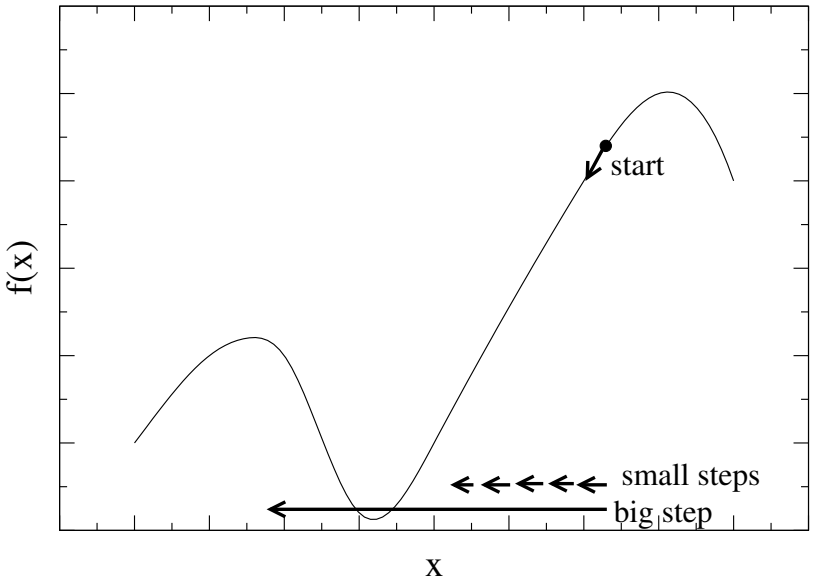
b. Multidimensional Minimization

The basic problem of multidimensional minimization is that we have a function f of $n > 1$ independent variables, and we want to find the values of these variables for which f is a minimum (to find a maximum, look for the minimum of $-f$). This topic is discussed in *Numerical Recipes* [2] in Chap. 10, entitled "Minimization or Maximization of Functions." Take a look at the online version for a good introduction to this extensive topic.



For $n = 1$, we can find a minimum between two given points using a *bracketing* method. A minimum is bracketed if there is a triplet of points $a < b < c$ such that $f(b)$ is less than both $f(a)$ and $f(c)$. If the function is nonsingular, it has a minimum in the interval (a, c). For example, we can *bisect* the interval repeatedly. In the figure above, we have identified a, b, and c (see Ref. [2] for strategies to find the initial bracketing). Now consider an x value between b and c and evaluate $f(x)$. If $f(b) < f(x)$, then the new triple is (a, b, x), while if $f(b) > f(x)$ the new triple is (b, x, c).

We then repeat the process starting with the new triple. When the size of the subsequent interval defined by the outer points of the triple is below a specified tolerance, we have found the minimum to that tolerance. See *Numerical Recipes* [2] for more details.



Alternatively, we can identify the downhill direction and take successive small steps in that direction (see the figure above). We take small steps to avoid overshooting the minimum. After each step, we check the downhill direction and continue, taking smaller steps as we get close (and the function becomes less steep). When we are close to the minimum, we can jump more rapidly to the minimum using additional information; for example fitting a bracketed triple to a parabola and jumping to the minimum of the parabola. Various strategies and algorithms are described in Ref. [2].

In many dimensions, there is no way to bracket, so the usual approaches are variations of the downhill method. In this case, the *gradient* of the function points in the steepest direction and various algorithms have been developed to accelerate the convergence to a minimum. The details of the various methods are available in the GSL documentation and in *Numerical Recipes*. However, the details are often less important than having different methods available:

i) to check your result;

ii) because some methods may converge much faster or find a different local minimum.

In one dimension, when looking for a root or a minimum of a function, *always* plot it first. Then you can see the *global* structure of the function and ensure that you find the global minimum (if that is what you want). It is easy to start in the "wrong" place and find a *local* minimum by the downhill method, but not a global minimum. In many dimensions, plotting is not usually an option, so finding a global minimum is a difficult problem. We'll revisit this when we discuss simulated annealing in a future session.

In Session 11 (and in a later session), we'll consider a minimization problem from chemistry:

given some Na^+ (sodium) and Cl^- (chlorine) ions, what is the most stable configuration of the molecule formed from them? The plan is to look for the absolute minimum of the (classical) potential energy (we can ignore the kinetic energy). (Question: What keeps Na^+ and Cl^- from collapsing into each other?) Implementing this problem is slightly tricky with the GSL minimizers, since there are $3N$ coordinates and $3N-6$ degrees of freedom (in general). So we need to freeze some coordinates. This is done in the code `multimin_nacl.cpp`, which you will be given. You'll find for the larger configurations that finding a global minimum is tough, because you'll get different answers from the code for different starting points.

c. Nonlinear Curve Fitting

Given a set of data from an experiment (real or numerical), we often want to fit it to a model that depends on a set of parameters. The model could be a line or a more general polynomial, in which case the parameters are the coefficients of the polynomial. Or it could be a sum of gaussians. Or something really crazy. We might want to fit the data to a functional form in order to find its derivative or to interpolate between the measured data.

The important feature here is that the data is noisy, either with real experimental errors or with theoretical or computational errors. So we need a *figure-of-merit function* that provides a measure of how well the data and the model agree. Then the problem of fitting the function is reduced to minimizing the merit function with respect to the model parameters, which yields the *best-fit parameters* and a measure of the *goodness-of-fit* in a statistical sense [2]. This is, in general, a multidimensional minimization problem.

Chapter 15 in *Numerical Recipes* [2], entitled "Modeling of Data", has details (and further references) on figure-of-merit functions. We'll simply use *least squares*. Suppose we have n data points $\{y_i\}$ with errors σ_i (one standard deviation) and our model has p parameters $\{x_j\}$. We'll denote the model evaluated at the i'th point as $y[i; x_1, \cdots, x_p]$. Using the GSL notation, we minimize $\Phi(x)$, the sum of squared residuals of the f_i with respect to the parameters x_i:

$$\Phi(x) = \frac{1}{2}\sum_{i=0}^{n-1} f_i(x_1, \cdots, x_p)^2 \tag{11.1}$$

$$\equiv \frac{1}{2}||F(x)||^2 \ , \tag{11.2}$$

where the i'th residual is defined as the difference of predicted and observed values divided by the error:

$$f_i(x_1, \cdots, x_p) \equiv (y[i; x_1, \cdots, x_p] - y_i)/\sigma_i \ . \tag{11.3}$$

The Jacobian of the f_i is used to linearize the problem around an initial guess (i.e., this checks locally the downhill direction).

GSL routines are available for both linear models, in which the model depends linearly on the parameters, and nonlinear models. We'll consider the latter here. We'll use the example provided

in the GSL reference manual: fit a function

$$Ae^{-\lambda t} + b \ , \tag{11.4}$$

to some data, finding the best A, λ, and b in a least-squares sense. Note that while A and b appear linearly, λ does not. In order to test the program, we'll use *pseudodata*. By this we mean we'll choose values for A, λ, and b, determine function values at a set of t_i's, but then make it like data by adding errors distributed according to a gaussian distribution. (We'll discuss how to generate random numbers according to a given distribution in the next session. For now, just trust that it works!)

d. References

[1] *GSL Reference Manual*, http://sources.redhat.com/gsl/ref/gsl-ref_toc.html.

[2] 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.