

## 10. 780.20 Session 10

### a. A Couple of Mathematica Follow-ups

- **Exporting Tables from Mathematica for Plotting.** To extract a table for use in a plotter such as gnuplot or xmgrace, it is easiest 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!

- **Don't Be Too Trusting of Mathematica!**

Here's a nice example showing that you must be careful when assuming Mathematica is giving you accurate results (or at least as accurate as it claims). Suppose you were testing your integration programs on the integral  $\int_1^2 3\sin(x)/x dx$  and you decided to get the "exact" answer to 16 digits from Mathematica. Here's what you get:

```
ans1 = NumberForm[Integrate[3*Sin[x]/x, {x, 1.0, 2.0}], 16]
1.977989715418176
```

```
Precision[ans1]
MachinePrecision
```

Mathematica claims the answer is good to machine precision. But it isn't! You get the correct answer using `NIntegrate`:

```
ans2 = NumberForm[NIntegrate[3*Sin[x]/x, {x, 1.0, 2.0}], 16]
1.977989719306536
```

or by evaluating the analytic result separately for the upper and lower limit:

```
ans3 = Integrate[3*Sin[x]/x, x]
3 SinIntegral[x]
```

```
upper = ans3 /. {x -> 2.};
lower = ans3 /. {x -> 1.};
NumberForm[upper - lower, 16]
1.977989719306535
```

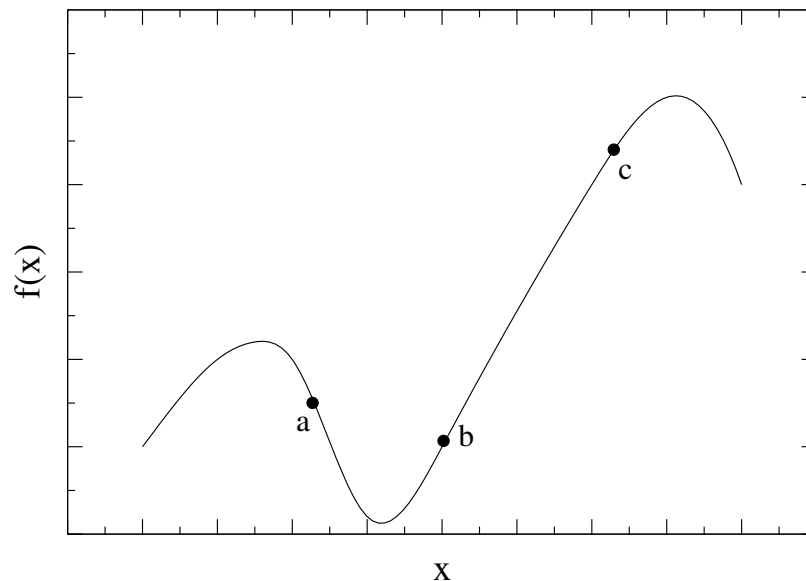
So don't trust your answer from Mathematica. Try doing it more than one way and compare to answers from MATLAB or Maple.

### 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.

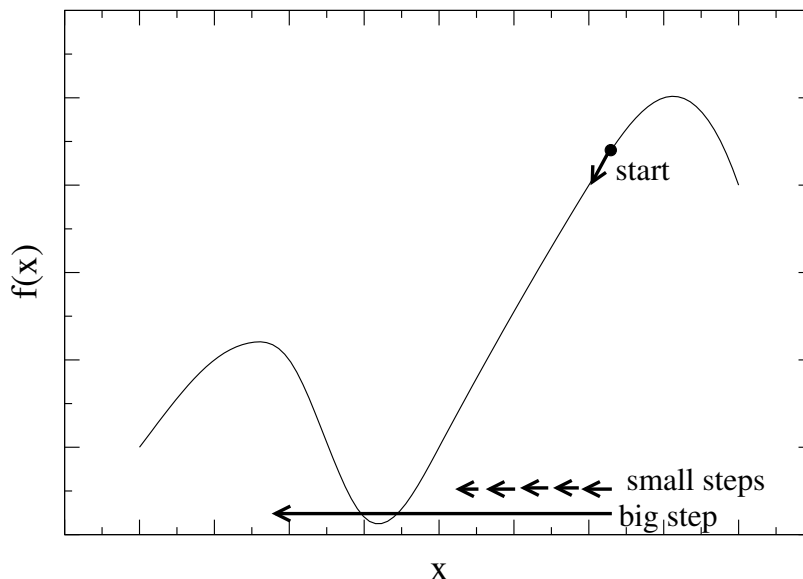
For  $n = 1$ , we can find a minimum between two given points using a *bracketing* method. A minimum is bracketed if there is a triplet of points  $a < b < c$  such that  $f(b)$  is less than both  $f(a)$  and  $f(c)$ . If the function is nonsingular, it has a minimum in the interval  $(a, c)$ . For example, we can *bisect* the interval repeatedly. In the figure below, 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 on the next page). 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 approach is 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 10 (and later in Session 14), we’ll consider a minimization problem from chemistry: given some  $\text{Na}^+$  (sodium) and  $\text{Cl}^-$  (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  $\text{Na}^+$  and  $\text{Cl}^-$  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  $\sigma_i$  (one standard deviation) and our model has  $p$  parameters  $\{x_j\}$ . We’ll denote the model evaluated at the  $i$ ’th point as  $y[i; x_1, \dots, x_p]$ . Using the GSL notation, we minimize  $\Phi(x)$ , the sum of squared residuals of the  $f_i$  with respect to the parameters  $x_i$ :

$$\Phi(x) = \frac{1}{2} \sum_{i=0}^{n-1} f_i(x_1, \dots, x_p)^2 \quad (10.1)$$

$$\equiv \frac{1}{2} \|F(x)\|^2, \quad (10.2)$$

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

$$f_i(x_1, \dots, x_p) \equiv (y[i; x_1, \dots, x_p] - y_i) / \sigma_i. \quad (10.3)$$

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

$$Ae^{-\lambda t} + b, \quad (10.4)$$

to some data, finding the best  $A$ ,  $\lambda$ , and  $b$  in a least-squares sense. Note that while  $A$  and  $b$  appear linearly,  $\lambda$  does not. In order to test the program, we’ll use *pseudodata*. By this we mean we’ll choose values for  $A$ ,  $\lambda$ , 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](http://sources.redhat.com/gsl/ref/gsl-ref_toc.html).
- [2] 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++.