9
Data Fitting

Data fitting is an art worthy of serious study. In this unit we just scratch the surface. We examine how to interpolate within a table of numbers and how to do a least-squares fit for linear functions. If a least-squares fit is needed for nonlinear functions, then some of the search routines, either your own or those obtained from scientific subroutine libraries, may be used. We also describe how to do that. In a recent work [12], simulated annealing *(which we describe in Chap. 28) is used to assist the search in least-squares fitting.*

9.1
Fitting Experimental Spectrum (Problem IIIA)

The cross section measured for the resonant scattering of a neutron from a nucleus is given in Tab. 9.1. The first row is the energy; the second row, the experimental cross section; and the third row, the experimental error for each measurement. Your **problem** is to determine values for the cross sections at values of energy lying between those measured by experiment.

Tab. 9.1 Experimental values for a scattering cross section Σ as a function of energy.

i	1	2	3	4	5	6	7	8	9
E_i (MeV) $[\equiv x_i]$	0	25	50	75	100	125	150	175	200
$f(E_i)$ (mb)	10.6	16.0	45.0	83.5	52.8	19.9	10.8	8.25	4.7
Error $= \pm\sigma_i$ (mb)	9.34	17.9	41.5	85.5	51.5	21.5	10.8	6.29	4.09

You can view your **problem** in a number of ways. The most direct is to numerically *interpolate* between the values of the experimental $\Sigma(E_i)$ given in Tab. 9.1. This is direct and easy, but it ignores the possibility of there being experimental noise in the data in that it assumes that the data can be represented as a polynomial in E, at least over some small range.

In Section 9.4 we discuss another way to view this problem. Specifically, we start with what one believes to be the "correct" theoretical description of the

Computationyal Physics. Problem Solving with Computers (2nd edn).
Rubin H. Landau, Manuel José Páez, Cristian C. Bordeianu

ISBN: 978-3-527-40626-5

data,

$$f(E) = \frac{f_r}{(E - E_r)^2 + \Gamma^2/4} \tag{9.1}$$

where are constants to be determined by the fitting, and then adjust the parameters f_r, E_r, and Γ to obtain a *best fit* to the data.[1] This is a "best" fit in a statistical sense, but, in fact, may not pass through all (or any) of the data points. For an easy, yet effective, introduction to statistical data analysis, we recommend [13].

These techniques of interpolation and least-squares fitting are powerful tools that let you treat tables of numbers as if they were analytic functions, and sometimes let you deduce statistically meaningful constants or conclusions from measurements. In general, you can view data fitting as *global* or *local*. In global fits, a single function in x is used to represent the entire set of numbers in a table such as Tab. 9.1. While it may be spiritually satisfying to find a single function that passes through all the data points, if that function is not the correct function for describing the data, the fit may have nonphysical behavior (such as large oscillations) between the data points. The rule of thumb is that if you must interpolate, then keep it local. Although we ask you to do one, global fits should be looked at with a jaundiced eye.

9.1.1 Lagrange Interpolation (Method)

Consider Tab. 9.1 as ordered data that we wish to interpolate. We call the independent variable x, with tabulated values x_i $(i = 1, 2, \ldots)$, and we assume that the dependent variable is the function $g(x)$, with tabulated values $g_i = g(x_i)$. We assume that $g(x)$ can be approximated as a polynomial of degree $(n-1)$ in each interval i:

$$g_i(x) \simeq a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} \qquad (x \simeq x_i) \tag{9.2}$$

Because our fit is local, we do not assume that one $g(x)$ can fit all the data in the table, but instead will use a different polynomial, that is, a different set of a_i values, for each region of the table. While each polynomial is of low degree, there are several polynomials that are needed to cover the entire table. If some care is taken, the set of polynomials so obtained behaves well enough to be used in further calculations without introducing much unwanted noise or discontinuities.

The classic of interpolation formulas was created by Lagrange. He figured out a closed-form one that directly fits the $(n-1)$-order polynomial (9.2) to n

[1] Chapter 17 on Fourier analysis discusses yet another way to fit data.

values of the function $g(x)$ evaluated at the points x_i. The formula is written as the sum of polynomials:

$$g(x) \simeq g_1\lambda_1(x) + g_2\lambda_2(x) + \cdots + g_n\lambda_n(x) \tag{9.3}$$

$$\lambda_i(x) = \prod_{j(\neq i)=1}^{n} \frac{x - x_j}{x_i - x_j} = \frac{x - x_1}{x_i - x_1}\frac{x - x_2}{x_i - x_2}\cdots\frac{x - x_n}{x_i - x_n} \tag{9.4}$$

For three points, (9.3) provides a second-degree polynomial, while for eight points it gives a seventh-degree polynomial. For example, here we use a four-point Lagrange interpolation to determine a third-order polynomial that reproduces each of the tabulated values:

$$\begin{aligned} g(x) = {} & \frac{(x-1)(x-2)(x-4)}{(0-1)(0-2)(0-4)}(-12) + \frac{x(x-2)(x-4)}{(1-0)(1-2)(1-4)}(-12) \\ & + \frac{x(x-1)(x-4)}{(2-0)(2-1)(2-4)}(-24) + \frac{x(x-1)(x-2)}{(4-0)(4-1)(4-2)}(-60) \end{aligned}$$

$$\Rightarrow \qquad g(x) = x^3 - 9x^2 + 8x - 12$$

As a check we see that

$$g(4) = 4^3 - 9(4^2) + 32 - 12 = -60 \qquad g(0.5) = -10.125 \tag{9.5}$$

If the data contain little noise, this polynomial can be used with some confidence within the range of data, but with risk beyond the range of data.

Notice that Lagrange interpolation makes no restriction that the points in the table be evenly spaced. As a check, it is also worth noting that the sum of the Lagrange multipliers equals one, $\sum_{i=1}^{n} \lambda_i = 1$. Usually the Lagrange fit is made to only a small region of the table with a small value of n, even though the formula works perfectly well for fitting a high-degree polynomial to the entire table. The difference between the value of the polynomial evaluated at some x and that of the actual function is equal to the *remainder*

$$R_n \approx \frac{(x - x_1)(x - x_2)\cdots(x - x_n)}{n!} g^{(n)}(\zeta) \tag{9.6}$$

where ζ lies somewhere in the interpolation interval, but is otherwise undetermined. This shows that if significant high derivatives exist in $g(x)$, then it cannot be approximated well by a polynomial. In particular, if $g(x)$ is a table of experimental data, then it is likely to contain noise, and then it is a bad idea to fit a curve through all the data points.

Listing 9.1: `Lagrange.java` makes a Lagrange interpolation of tabulated data.

```
//    Lagrange.java: Langrange interpolation of tabulated data

 import java.io.*;                                  //Location of PrintWriter

public class Lagrange     {

  public static void main(String[] argv) throws IOException,
                                                   FileNotFoundException  {
    PrintWriter w = new PrintWriter(
                    new FileOutputStream("Lagr.dat"), true);
    PrintWriter ww = new PrintWriter(
                     new FileOutputStream("Lagr_input.dat"), true);
    double x, y;
    int  i,j,k; int end = 9;
                                                                  //Input data
    double xin[] = {0, 25, 50, 75, 100, 125, 150, 175, 200};
    double yin[] = {10.6, 6, 45, 83.5, 52.8, 19.9, 10.8, 88.25, 4.7};

    for (k=0; k<9; k++) ww.println( xin[k] +" " + yin[k]);
    for ( k=0; k<=1000;k++){
      x = k*0.2 ;
      y = inter(xin, yin, end, x);
      System.out.println("Lagrange  x=" +x+" , y= "+y);     // to file
      w.println( x  +" "  +y);
    }
   System.out.println("Lagrange Program Complete.");
   System.out.println("Fit in Lagr.dat, input in Lagr_input.dat");
  }

   public static double inter ( double xin[], double yin[],
                                                 int end, double x)   {
     double  lambda,y;  int i,j;
     y = 0.0;
     for ( i=1; i <= end; i++) {
       lambda = 1.0;
       for ( j=1; j<=end; j++) if (i != j)
              {lambda *= ((x - xin[j-1])/(xin[i-1] - xin[j-1]));}
       y += (yin[i-1] * lambda);
      }
  return y;
} }
```

9.1.2 Lagrange Implementation and Assessment

Consider the experimental neutron scattering data in Tab. 9.1. The expected theoretical functional form which describes these data is (9.1), and our empirical fits to these data are shown in Fig. 9.1.

1. Write a subroutine to perform an n-point Lagrange interpolation using (9.3). Treat n as an arbitrary input parameter. (You can also do this

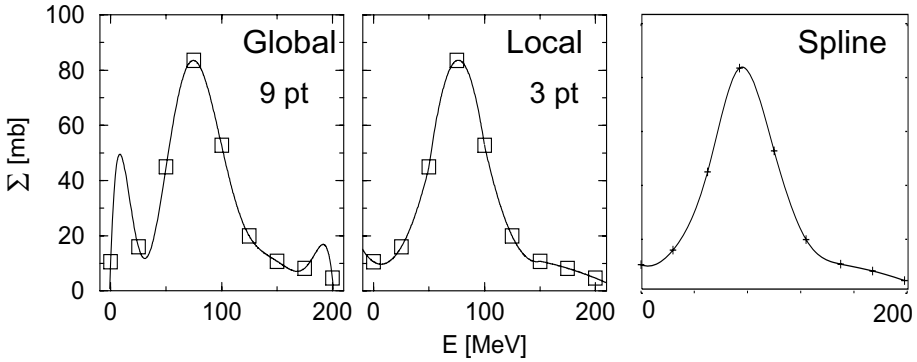

Fig. 9.1 *Left:* fit to cross section that places an eighth degree polynomial through all data. *Middle:* fit to cross section using Lagrange interpolation to place a series of third-degree polynomials through the data. *Right:* a cubic-splines fit to same data provides continuous first and second derivatives. Note the differences at low energies of the spline fit from the other two.

exercise with the spline fits discussed in Section 9.1.4.) Our program is given in Listing 9.1.

2. Use the Lagrange interpolation formula to fit the entire experimental spectrum with one polynomial. (This means that you want to fit all nine data points with an eight-degree polynomial.) Then use this fit to plot the cross section in steps of 5 MeV.

3. Use your graph to deduce the resonance energy E_r (your peak position) and Γ (the full width at half maximum). Compare your results with those predicted by our theorist friend, $(E_r, \Gamma) = (78, 55)$ MeV.

4. A more realistic use of Lagrange interpolation is for local interpolation with a small number of points, such as three. Interpolate the preceding cross-section data in 5 MeV steps using three-point Lagrange interpolation. (Note, the end intervals may be special cases.)

This example shows how easy it is to go wrong with a high-degree polynomial fit. Although the polynomial is guaranteed to actually pass through all the data points, the representation of the function away from these points can be quite unrealistic. Using a low-order interpolation formula, say, $n = 2$ or 3, in each interval usually eliminates the wild oscillations. If these local fits are then matched together, as we discuss in the next section, a rather continuous curve results. Nonetheless, you must recall that if the data contain errors, a curve that actually passes through them may lead you astray. We discuss how to do this properly in Section 9.4.

9.1.3 Explore Extrapolation

We deliberately have not discussed *extrapolation* of data because it can lead to serious *systematic* errors; the answer you get may well depend more on the function you assume than on the data you input. Add some adventure to your life and use the programs you have written to extrapolate to values outside of the table. Compare your results to the theoretical Breit–Wigner shape (9.1).

9.1.4 Cubic Splines (Method)

If you tried to interpolate the resonant cross section with Lagrange interpolation, then you saw that fitting parabolas (three-point interpolation) within a table may avoid the erroneous and possibly catastrophic deviations of a high-order formula. (Two-point interpolation, which connects the points with straight lines, may not lead you far astray, but it is rarely pleasing to the eye or precise.) A sophisticated variation of $n = 4$ interpolation, known as *cubic splines*, often leads to surprisingly eye-pleasing fits. In this approach (Fig. 9.1 right), cubic polynomials are fit to the function in each interval, with the additional constraint that the first and second derivatives of the polynomials must be continuous from one interval to the next. This continuity of slope and curvature is what makes the spline fit particularly eye-pleasing. It is analogous to what happens when you use the flexible drafting tool (a lead wire within a rubber sheath) from which the method draws its name.

The series of cubic polynomials obtained by spline-fitting a table can be integrated and differentiated, and is guaranteed to have well-behaved derivatives. The existence of meaningful derivatives is an important consideration. As a case in point, if the interpolated function is a potential, you can take the derivative to obtain the force. The complexity of simultaneously matching polynomials and their derivatives over all the interpolation points leads to many simultaneous, linear equations to be solved. This makes splines unattractive for hand calculations, yet easy for computers. Splines have made recent gains in popularity and applicability in both calculations and graphics. To illustrate, the smooth curves connecting points in most "draw" programs are usually splines.

The basic approximation of splines is the representation of the function $g(x)$ in the subinterval $[x_i, x_{i+1}]$ with a cubic polynomial:

$$g(x) \simeq g_i(x) \qquad \text{for } x_i \le x \le x_{i+1} \tag{9.7}$$

$$g_i(x) = g_i + g_i'(x - x_i) + \tfrac{1}{2}g_i''(x - x_i)^2 + \tfrac{1}{6}g_i'''(x - x_i)^3 \tag{9.8}$$

This representation makes it clear that the coefficients in the polynomial equal the values of $g(x)$ and to its first, second, and third derivatives at the tabulated

points x_i. Derivatives beyond the third vanish. The computational chore is to determine these derivatives in terms of the N tabulated values g_i. The matching of g_i from one interval to the next (at the *nodes*) provides the equations

$$g_i(x_{i+1}) = g_{i+1}(x_{i+1}) \qquad i = 1, N-1 \tag{9.9}$$

The matching of the first *and* second derivatives at each subinterval's boundary provides the equations

$$g'_{i-1}(x_i) = g'_i(x_i) \qquad g''_{i-1}(x_i) = g''_i(x_i) \tag{9.10}$$

To provide the additional equations needed to determine all constants, the third derivatives at adjacent nodes are matched. Values for the third derivatives are found by approximating them in terms of the second derivatives:

$$g'''_i \simeq \frac{g''_{i+1} - g''_i}{x_{i+1} - x_i} \tag{9.11}$$

As discussed in Chap. 15 a *central difference approximation* would be better than the forward difference, yet (9.11) keeps the equations simpler.

It is straightforward though complicated to solve for all the parameters in (9.8). We leave that to other reference sources [9, 14]. We can see, however, that matching at the boundaries of the intervals results in only $N-2$ linear equations for N unknowns. Further input is required. It usually is taken to be the boundary conditions at the endpoints $a = x_1$ and $b = x_N$, specifically, the second derivatives $g''(a)$, and $g''(b)$. There are several ways to determine these second derivatives.

Natural spline: Set $g''(a) = g''(b) = 0$, that is, permit the function to have a slope at the endpoints but no curvature. This is "natural" because the derivative vanishes for the flexible spline drafting tool (its ends being free).

Input values for g' at boundaries: The computer uses $g'(a)$ to approximate $g''(a)$. If you do not know the first derivatives, you can calculate them numerically from the table of g_i values.

Input values for g'' at boundaries: Knowing values is of course better than assuming values, but it requires more input. If the values of g'' are not known, they can be approximated by applying a forward-difference approximation to the tabulated values:

$$g''(x) \simeq \frac{[g(x_3) - g(x_2)]/[x_3 - x_2] - [g(x_2) - g(x_1)]/[x_2 - x_1]}{[x_3 - x_1]/2} \tag{9.12}$$

9.1.4.1 Cubic Spline Quadrature (Exploration)

A powerful integration scheme is to fit an integrand with splines, and then integrate the cubic polynomials analytically. If the integrand $g(x)$ is known

only at its tabulated values, then this is about as good an integration scheme as possible; if you have the ability to actually calculate the function for arbitrary x, Gaussian quadrature may be preferable. We know that the spline fit to g in each interval is the cubic (9.8),

$$g(x) \simeq g_i + g_i'(x - x_i) + \tfrac{1}{2}g_i''(x - x_i)^2 + \tfrac{1}{6}g_i'''(x - x_i)^3 \tag{9.13}$$

It is easy to integrate this to obtain the integral of g for this interval, and then to sum over all intervals:

$$\int_{x_i}^{x_{i+1}} g(x)dx \;\simeq\; \left(g_i x + \tfrac{1}{2}g_i' x_i^2 + \tfrac{1}{6}g_i'' x^3 + \tfrac{1}{24}g_i''' x^4\right)\Big|_{x_i}^{x_{i+1}} \tag{9.14}$$

$$\int_{x_j}^{x_k} g(x)dx = \sum_{i=j}^{k} \int_{x_i}^{x_{i+1}} g(x)dx \tag{9.15}$$

Making the intervals smaller does not necessarily increase precision as subtractive cancellations in (9.14) may get large.

9.1.5 Spline Fit of Cross Section (Implementation)

Listing 9.2: The program `SplineAppl.java` performs a cubic spline fit to data. The arrays `x[]` and `y[]` are the data to fit, and the values of the fit at `Nfit` points are output into the file `Spline.dat`.

```
/* SplineAppl.java: Application version of cubic spline fitting
               interpolates array x[n], y[n], x0 < x1 ... < x(n-1)
               yp1, ypn: y' at ends, evaluated internally,
               y2[]: y" array; yp1, ypn > e30 for natural spline   */

import java.io.*;

public class SplineAppl  {

  public static void main(String[] argv)
                        throws IOException, FileNotFoundException {

    PrintWriter w =
          new PrintWriter(new FileOutputStream("Spline.dat"), true);
    PrintWriter q =
           new PrintWriter(new FileOutputStream("Input.dat"), true);
                                                          // input
    double x[] = {0.,1.2,2.5,3.7,5.,6.2,7.5,8.7,9.9};
    double y[] = {0.,0.93,.6,-0.53,-0.96,-0.08,0.94,0.66,-0.46};
    int  i,n = x.length,  np = 15, klo, khi, k;
    double y2[] = new double[9];  double u[] = new double[n];
    double h, b, a, Nfit, p, qn, sig, un, yp1, ypn, xout, yout;
                                                    // Output data
    for (i=0; i < n; i++) q.println (" " + x[i] + " " + y[i] + " ");
    Nfit = 30;                                       // N output pts
                                                 // Initial 1st deriv
```

```
    yp1 = (y[1]-y[0])/(x[1]-x[0])
                - (y[2]-y[1])/(x[2]-x[1]) + (y[2]-y[0])/(x[2]-x[0]);
    ypn = (y[n-1]-y[n-2])/(x[n-1]-x[n-2]) - (y[n-2]-y[n-3])
                /(x[n-2]-x[n-3]) + (y[n-1]-y[n-3])/(x[n-1]-x[n-3]);
                                                    // Natural spline
    if (yp1 > 0.99e30) y2[0] = u[0] = 0. ;
     else  {
       y2[0] = (-0.5);
       u[0] = (3./(x[1]-x[0]))*((y[1]-y[0])/(x[1]-x[0])-yp1);
     }
                                                  // Decomposition loop
    for ( i=1;  i <= n-2;  i++ )  {
      sig = (x[i]-x[i-1])/(x[i + 1]-x[i-1]);
      p = sig*y2[i-1] + 2. ;
      y2[i] = (sig-1.)/p;
      u[i] = (y[i+1]-y[i])/(x[i+1]-x[i])-(y[i]-y[i-1])/(x[i]-x[i-1]);
      u[i] = (6.*u[i]/(x[i+1]-x[i-1])-sig*u[i-1])/p;
    }
                                                    // Test for natural
    if (ypn > 0.99e30) qn = un = 0. ;
    else {
      qn = 0.5;
      un =
           (3./(x[n-1]-x[n-2]))*(ypn-(y[n-1]-y[n-2])/(x[n-1]-x[n-2]));
    }
    y2[n-1] = (un-qn*u[n-2])/(qn*y2[n-2] + 1.);
    for ( k = n-2;  k>= 0; k--)  y2[k] = y2[k]*y2[k + 1] + u[k];
                                    // initialization ends, begin fit
    for ( i=1;  i <= Nfit;  i++ ) {                  // Loop over xout
      xout = x[0] + (x[n-1]-x[0])*(i-1)/(Nfit);
      klo = 0;
      khi = n-1;                                          // xout value
                                  // Bisection algor, klo, khi bracket
      while (khi-klo > 1) {
        k = (khi + klo) >> 1;
        if (x[k] > xout) khi = k;    else klo = k;
      }
      h = x[khi]-x[klo];
      if (x[k] > xout) khi = k;    else klo = k;
      h = x[khi]-x[klo];
      a = (x[khi]-xout)/h;
      b = (xout-x[klo])/h;
      yout = (a*y[klo] + b*y[khi] + ((a*a*a-a)*y2[klo]
                                      + (b*b*b-b)*y2[khi])*(h*h)/6.);
      w.println (" " + xout + " " + yout + " ");
    }
    System.out.println("data stored in Spline.dat");
  }
}
```

Fitting a series of cubics to data is a little complicated to program up yourself, so we recommend using a library routine. While we have found quite a few Java-based spline applications available on the internet, none seemed appropriate for interpreting a simple set of numbers. That being the case, we have adapted the `splint.c` and the `spline.c` functions from [9] to produce

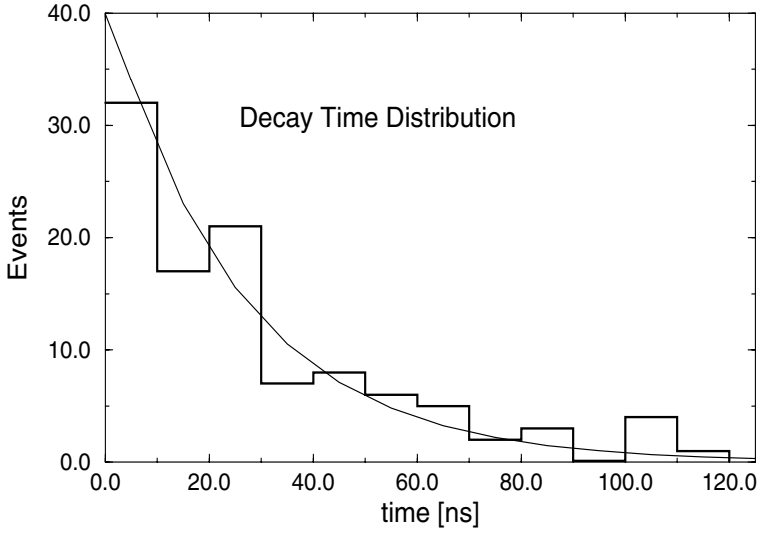

Fig. 9.2 A reproduction of the experimental measurement in [15] of the number of decays of a π meson as a function of time. Measurements are made during time intervals of 10-ns length. Each "Event" corresponds to a single decay.

the `SplineAppl.java` program shown in Listing 9.2. Your problem now is to carry out the assessment of Section 9.1.2 using cubic spline interpolation rather than Lagrange interpolation.

9.2 Fitting Exponential Decay (Problem IIIB)

Figure 9.2 presents actual experimental data on the number of decays ΔN of the π meson as a function of time [15]. Notice that the time has been "binned" into $\Delta t = 10$ ns intervals, and that the smooth curve gives the theoretical exponential decay law. Your **problem** is to deduce the lifetime τ of the π meson from these data (the tabulated lifetime of the pion is 2.6×10^{-8} s).

9.2.1 Theory to Fit

Assume that we start with N_0 particles at time $t = 0$ that can decay to other particles.[2] If we wait a short time Δt, then a small number ΔN of the particles will decay *spontaneously*, that is, with no external influences. This decay is a stochastic process, which means that there is an element of chance involved

[2] Spontaneous decay is discussed further and simulated in Section 11.2.

in just when a decay will occur, and so no two experiments are expected to give exactly the same results. The basic law of nature for spontaneous decay is that the number of decays ΔN in the time interval Δt is proportional to the number of particles $N(t)$ present at that time, and to the time interval

$$\Delta N(t) = -\frac{1}{\tau} N(t) \Delta t \qquad \Rightarrow \qquad \frac{\Delta N(t)}{\Delta t} = -\lambda N(t) \tag{9.16}$$

Here $\tau = 1/\lambda$ is the *lifetime* of the particle, with λ the rate parameter. The actual decay *rate* is given by the second equation in (9.16). If the number of decays ΔN is very small compared to the number of particles N, and if we look at vanishingly small time intervals, then the difference equation (9.16) becomes the differential equation

$$\frac{dN(t)}{dt} \simeq -\lambda N(t) = \frac{1}{\tau} N(t) \tag{9.17}$$

This differential equation has an exponential solution for the number, as well as for the decay rate:

$$N(t) = N_0 e^{-t/\tau} \qquad \frac{dN(t)}{dt} = -\frac{N_0}{\tau} e^{-t/\tau} = \frac{dN}{dt}(0)\, e^{-t/\tau} \tag{9.18}$$

Equation (9.18) is the theoretical formula we wish to "fit" to the data in Fig. 9.2. The output of such a fit is a "best value" for the lifetime τ.

9.3 Theory: Probability and Statistics

The field of statistics is an attempt to apply the mathematical theory of probability to describe natural events, such as coin flips, in which there is an element of chance or randomness. Some basic elements of probability upon which statistics is based are as follows.

1. The probability $P(x)$ of an event x equals the fraction of times that the event occurs in a series of experiments.
2. Probability lies in the range $0 \leq P(x) \leq 1$, with 0 corresponding to an impossible event, and 1 to a sure thing.
3. If $P(x)$ is the probability of an event occurring, then $1 - P(x)$ is the probability of the event *not* occurring (the complement or opposite).
4. Given a population to be sampled, the probability of one sample having a particular value equals the fraction of the population having that value.

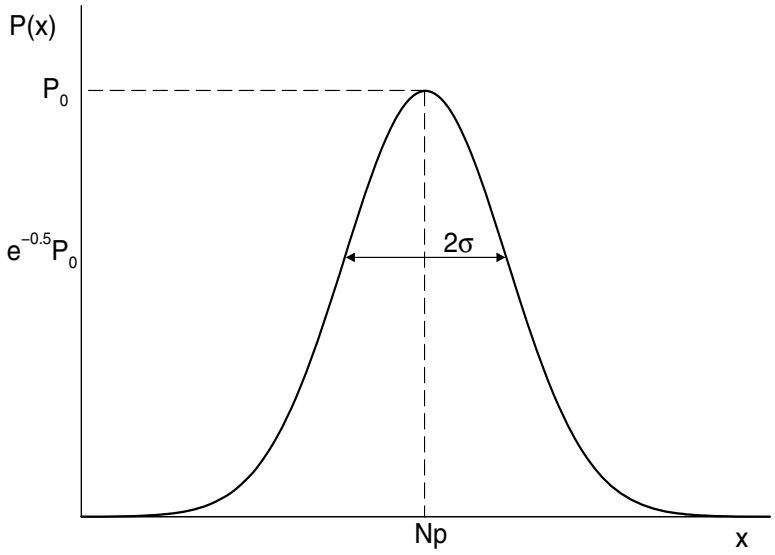

Fig. 9.3 A Gaussian distribution of m successes in N trials, each with probability p.

5. The mean value $\overline{f}$ of a function of x is given by the weighted sum

$$\overline{f(x)} \equiv \langle f(x) \rangle \quad \stackrel{\text{def}}{=} \quad \sum_{i=1}^{\infty} P(x_i) f(x_i) \tag{9.19}$$

The basic building block of statistics is the *binomial distribution* function

$$P_B(x) = \binom{N}{x} p^x (1-p)^{N-x} = \frac{N!}{(N-x)!x!} p^x (1-p)^{N-x} \tag{9.20}$$

This function gives the probability $P_B(x)$ that an independent event (say heads) will occur x times in the N trials. Here p is the probability of occurrences of an individual event; for example, the probability of "heads" in any one toss is $p = \frac{1}{2}$. The variable N is the number of *trials* (measurements) in which that event can occur; for example, the number of times we flip the coin. For coin flipping, the probability of success p and the probability of failure $(1-p)$ are both $\frac{1}{2}$, but in the general case p can be any number between 0 and 1.

As a matter of convenience, we eliminate one of the factorials in the binomial distribution (9.20) by considering the limit in which the number of measurements of trials $N \to \infty$. If, in addition, the probability p of an individual event (heads) remains finite as $N \to \infty$, we have **Gaussian** or **normal** statistics, and the probability function takes the simple form (Fig. 9.3)

$$P_G(x) = \lim_{N\to\infty, p\neq 0} P_B(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right] \tag{9.21}$$

Here $\mu \equiv \overline{x}$ is the mean and σ is the variance:

$$\mu = Np \quad \sigma = \sqrt{Np(1-p)} \tag{9.22}$$

The Gaussian distribution is generally a very good approximation to the binomial distribution for $N > 10$, where is used to describe an experiment in which N measurements of the variable x are made. The average of these measurements is μ and the "error" or uncertainty in μ is σ. As an example, in N =1000 coin flips, the probability of a head is $p = \frac{1}{2}$ and the average number of heads μ should be $Np = N/2 = 500$. As shown in Fig. 9.3, the Gaussian distribution has a width

$$\sigma = \sqrt{Np(1-p)} \propto \sqrt{N} \tag{9.23}$$

This means that the distribution actually gets wider and wider as more measurements are made. Yet the *relative width*, whose inverse gives us an indication of the probability of obtaining the average μ, decreases with N:

$$\frac{\text{width}}{N} \propto \frac{\sqrt{N}}{N} = \frac{1}{\sqrt{N}} \to 0 \quad (N \to \infty) \tag{9.24}$$

Using a normal distribution to describe events that follow some other distribution function can lead to incorrect results. For example, another limit of the binomial distribution is the **Poisson** distribution. In the Poisson distribution, the number of trials $N \to \infty$, yet the probability of an individual success $p \to 0$ in such a way that the product Np remains finite:

$$P_P(x) = \lim_{N\to\infty, p\to 0} P_B(x) = \frac{\mu^x e^{-\mu}}{x!} \tag{9.25}$$

A **Poisson** distribution describes radioactive decay experiments or telephone interchanges where there may be a very large number of trials (each microsecond when the counter is on), but a low probability of an event (a decay or phone call) occurring in this 1 μs. As we see in Fig. 9.4, the Poisson distribution is quite asymmetric for small μ, and in this way quite different from a Gaussian distribution. For $\mu \gg 1$, the Poisson distribution approaches a Gaussian distribution.

We use these distribution functions to determine probabilities as if events were continuous. For example, we take

$$P(x)\,dx = \text{ probability that } x \text{ lies in the interval } x \le x \le x + dx. \tag{9.26}$$

If we then want to calculate the *mean* of some x, it would be

$$\overline{x} \quad \stackrel{\text{def}}{=} \quad \sum_{i=1}^{\infty} P(x_i)x_i = \int dx\, P(x)x \tag{9.27}$$

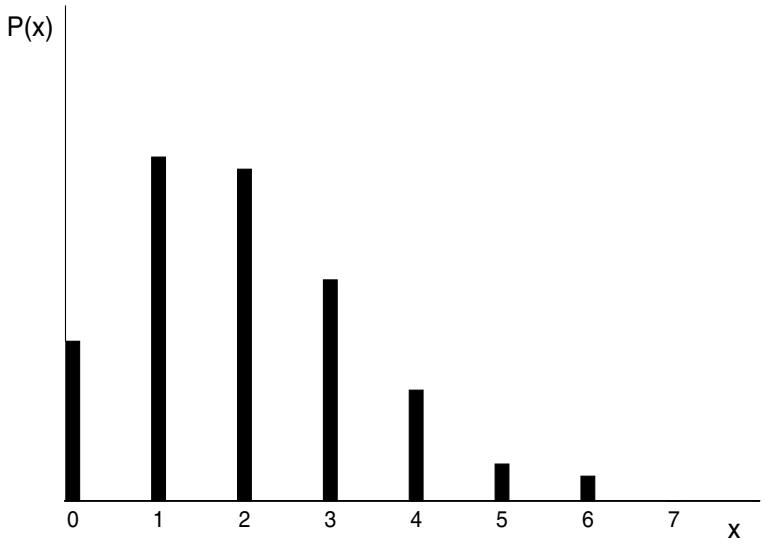

Fig. 9.4 A Poisson distribution for m successes with total success $a = 2$.

where the integral form would be used in those cases where we may approximate discrete values by a continuous distribution function. Likewise, the mean of an arbitrary function of x is

$$\overline{f(x)} \quad \stackrel{\text{def}}{=} \quad \sum_{i=1}^{\infty} P(x_i) f(x_i) = \int dx\, P(x) f(x) \tag{9.28}$$

When we apply these techniques to the normal distribution (9.21), we verify that μ is the mean value of x, and that σ is the root of the mean-square deviation of x from its average:

$$\overline{x} = \int_{-\infty}^{+\infty} dx \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right] x = \mu \tag{9.29}$$

$$\overline{x - \overline{x}} = \int_{-\infty}^{+\infty} dx \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right] (x - \overline{x})^2 = \sigma^2 \tag{9.30}$$

9.4 Least-Squares Fitting (Method)

Books have been written and careers have been spent discussing what is meant by a "good" fit to experimental data. We cannot do justice to the subject here and refer the reader to [9,13,14,16]. However, we will emphasize two points:

1. If the data being fit contain errors, then the "best" fit in a statistical sense should not pass through all the data points.

2. Only for the simplest case of a linear, least-squares fit can we write down a closed-form solution to evaluate and obtain the fit. More realistic problems are usually solved by *trial-and-error* search procedures, sometimes using sophisticated subroutines libraries. However, in Section 9.4.5 we show how to conduct such a nonlinear search using familiar tools.

Imagine that you have measured N_D data values of the independent variable y as a function of the dependent variable x:

$$(x_i, y_i \pm \sigma_i) \qquad i = 1, N_D \tag{9.31}$$

where $\pm\sigma_i$ is the uncertainty in the ith value of y. (For simplicity we assume that all the errors σ_i occur in the dependent variable, although this is hardly ever true [14].) For our problem, y is the number of decays as a function of time, and x_i are the times. Our goal is to determine how well a mathematical function $y = g(x)$ (also called *theory* or *model*) can describe these data. Alternatively, if the theory contains some parameters or constants, our goal can be viewed as determining best values for these parameters. We assume that the model function $g(x)$ contains, in addition to the functional dependence on x, an additional dependence upon M_P parameters $\{a_1, a_2, \ldots, a_{M_P}\}$. Notice that the parameters $\{a_m\}$ are not variables, in the sense of numbers read from a meter, but rather are parts of the theoretical model such as the size of a box, the mass of a particle, or the depth of a potential well. For the exponential decay function (9.18), the parameters are the lifetime τ and the initial decay rate $dN(0)/dt$. We indicate this as

$$g(x) = g(x; \{a_1, a_2, \ldots, a_{M_P}\}) = g(x; \{a_m\}) \tag{9.32}$$

We use the chi-squared (χ^2) measure as a gauge of how well a theoretical function g reproduces data

$$\chi^2 \quad \stackrel{\text{def}}{=} \quad \sum_{i=1}^{N_D} \left(\frac{y_i - g(x_i; \{a_m\})}{\sigma_i} \right)^2 \tag{9.33}$$

where the sum is over the N_D experimental points $(x_i, y_i \pm \sigma_i)$. The definition (9.33) is such that smaller values of χ^2 are better fits, with $\chi^2 = 0$ occurring if the theoretical curve went through the center of every data point. Notice also that the $1/\sigma_i^2$ weighting means that measurements with larger errors[3] contribute less to χ^2.

[3] If you are not given the errors, you can guess them on the basis of the apparent deviation of the data from a smooth curve, or you can weigh all points equally by setting $\sigma_i \equiv 1$ and continue with the fitting.

Least-squares fitting refers to adjusting the theory until a minimum in χ^2 is found, that is, finding a curve that produces the least value for the summed squares of the deviations of the data from the function $g(x)$. In general, this is the best fit possible or the best way to determine the parameters in a theory. The M_P parameters $\{a_m, m = 1, M_P\}$ that make χ^2 an extremum are found by solving the M_P equations:

$$\frac{\partial \chi^2}{\partial a_m} = 0 \quad \Rightarrow \quad \sum_{i=1}^{N_D} \frac{[y_i - g(x_i)]}{\sigma_i^2} \frac{\partial g(x_i)}{\partial a_m} = 0 \quad (m = 1, M_P) \tag{9.34}$$

More usually, the function $g(x; \{a_m\})$ has a sufficiently complicated dependence on the a_m values for (9.34) to produce M_P simultaneous, nonlinear equations in the a_m values. In these cases, solutions are found by a trial-and-error search through the M_P-dimensional parameter space, as we do in Section 9.4.5. To be safe, when such a search is completed you need to check that the minimum χ^2 you found is *global* and not *local*. One way to do that is to repeat the search for a whole grid of starting values, and if different minima are found, to pick the one with the lowest χ^2.

9.4.1 Goodness of Fit (Theory)

When the deviations from theory are due to random errors and when these errors are described by a Gaussian distribution, there are some useful rules of thumb to remember [13]. You know that your fit is good if the value of χ^2 calculated via the definition (9.33) is approximately equal to the number of degrees of freedom $\chi^2 \approx N_D - M_P$, where N_D is the number of data points and M_P the number of parameters in the theoretical function. If your χ^2 is much less than $N_D - M_P$, it does not mean that you have a "great" theory or a really precise measurement; instead, you probably have too many parameters or have assigned errors (σ_i values) that are too large. In fact, too small a χ^2 may indicate that you are fitting the random scatter in the data rather than missing $\sim \frac{1}{3}$ of the error bars, as expected for Gaussian statistics. If your χ^2 is significantly greater than $N_D - M_P$, the theory may not be good, you may have significantly underestimated your errors, or you may have errors which are not random.

9.4.2 Least-Squares Fits Implementation

The M_P simultaneous equations (9.34) simplify considerably if the functions $g(x; \{a_m\})$ depend *linearly* on the parameter values a_i:

$$g(x; \{a_1, a_2\}) = a_1 + a_2 x \tag{9.35}$$

In this case (also known as *linear regression*) there are $M_P = 2$ parameters, the slope a_2 and the y intercept a_1. Notice that while there are only two parameters to determine, there still may be an arbitrary number N_D of data points to fit. Remember that a unique solution is not possible unless the number of data points is equal to or greater than the number of parameters.

For this linear case, there are just two derivatives,

$$\frac{\partial g(x_i)}{\partial a_1} = 1 \qquad \frac{\partial g(x_i)}{\partial a_2} = x_i \tag{9.36}$$

and after substitution, the χ^2 minimization equations (9.34) can be solved [9]:

$$a_1 = \frac{S_{xx}S_y - S_x S_{xy}}{\Delta} \qquad a_2 = \frac{SS_{xy} - S_x S_y}{\Delta} \tag{9.37}$$

$$S = \sum_{i=1}^{N_D} \frac{1}{\sigma_i^2} \qquad S_x = \sum_{i=1}^{N_D} \frac{x_i}{\sigma_i^2} \qquad S_y = \sum_{i=1}^{N_D} \frac{y_i}{\sigma_i^2} \tag{9.38}$$

$$S_{xx} = \sum_{i=1}^{N_D} \frac{x_i^2}{\sigma_i^2} \qquad S_{xy} = \sum_{i=1}^{N_D} \frac{x_i y_i}{\sigma_i^2} \qquad \Delta = SS_{xx} - S_x^2 \tag{9.39}$$

Statistics also gives you an expression for the *variance* or uncertainty in the deduced parameters:

$$\sigma_{a_1}^2 = \frac{S_{xx}}{\Delta} \qquad \sigma_{a_2}^2 = \frac{S}{\Delta} \tag{9.40}$$

This is a measure of the uncertainties in the values of the fitted parameters arising from the uncertainties σ_i in the measured y_i values. A measure of the dependence of the parameters on each other is given by the *correlation coefficient*:

$$\rho(a_1, a_2) = \frac{\mathrm{cov}(a_1, a_2)}{\sigma_{a_1}\sigma_{a_2}} \qquad \mathrm{cov}(a_1, a_2) = \frac{-S_x}{\Delta} \tag{9.41}$$

Here $\mathrm{cov}(a_1, a_2)$ is the *covariance* of a_1 and a_2 and vanishes if a_1 and a_2 are independent. The correlation coefficient $\rho(a_1, a_2)$ lies in the range $-1 \leq \rho \leq 1$. Positive ρ indicates that the errors in a_1 and a_2 are likely to have the same sign; negative ρ indicates opposite signs.

Listing 9.3: `fit.java` makes a linear least-squares-fit to some tabulated data.

```
// Fit.java: Least-squares fit to tabulated data

public class fit  {
  static int  data =  12;                    // Number of data points
```

```
  public static void main(String[] argv)  {

    int i, j;
    double s, sx, sy, sxx, sxy, delta, inter, slope;
    double x[] = new double[data]; double y[] = new double[data];
    double d[] = new double[data];
                                                    // Input data x
    for (i = 0; i<data; i++) x[i] = i*180+5;
                                                    // Input data y
    y[0] = 382; y[1] = 187;  y[2] = 281; y[3] = 78;
    y[4] = 88;  y[5] = 86;   y[6] = 85;  y[7] = 28;
    y[8] = 28;  y[9] = 0.81; y[10] = 84; y[11] = 81;
                                                // Input data delta y
    for (i = 0; i<data; i++) d[i] = 18.;
    s = sx = sy = sxx = sxy = 0.;                       // Init sums
                                                  // Calculate sums
    for (i = 0;i<data;i++) {
      s   +=          1. / (d[i]*d[i]);
      sx  +=       x[i] / (d[i]*d[i]);
      sy  +=       y[i] / (d[i]*d[i]);
      sxx +=  x[i]*x[i] / (d[i]*d[i]);
      sxy +=  x[i]*y[i] / (d[i]*d[i]);
    }
    delta  =  s*sxx - sx*sx;
    slope =   (s*sxy - sx*sy) / delta;
    inter = (sxx*sy - sx*sxy) / delta;
    System.out.println("intercpt= "+inter+", "+Math.sqrt(sxx/delta));
    System.out.println("slope =  "+slope+" , "+Math.sqrt(s/delta));
    System.out.println("Fit Program Complete.");
} }
```

Our program to perform a linear least-square fit is given in Listing 9.3. In it we realize that the preceding analytic solutions for the parameters are of the form found in statistics books, but are not optimal for numerical calculations because subtractive cancellation can make the answers unstable. As discussed in Chap. 3, a rearrangement of the equations can decrease this type of error. For example, [14] gives improved expressions that measure the data relative to their averages:

$$a_1 = \overline{y} - a_2\overline{x} \qquad a_2 = \frac{S_{xy}}{S_{xx}} \qquad \overline{x} = \frac{1}{N}\sum_{i=1}^{N_d} x_i \qquad \overline{y} = \frac{1}{N}\sum_{i=1}^{N_d} y_i$$
$$S_{xy} = \sum_{i=1}^{N_d} \frac{(x_i - \overline{x})(y_i - \overline{y})}{\sigma_i^2} \qquad S_{xx} = \sum_{i=1}^{N_d} \frac{(x_i - \overline{x})^2}{\sigma_i^2} \tag{9.42}$$

9.4.3 Exponential Decay Fit Assessment

Fit the exponential decay law (9.18) to the data in Fig. 9.2. This means finding values for τ and $\Delta N(0)/\Delta t$ that provides a best fit to the data, and then judging how good the fit is.

1. Construct a table $(\Delta N/\Delta t_i, t_i)$, for $i = 1, N_D$ from Fig. 9.2. Because time was measured in bins, t_i should correspond to the middle of a bin.

2. Add an estimate of the error σ_i to obtain a table of the form $(\Delta N/\Delta t_i \pm \sigma_i, t_i)$. You can estimate the errors by eye, say, by estimating how much the histogram values appear to fluctuate about a smooth curve, or you can take $\sigma_i \simeq \sqrt{\text{Events}}$. (This last approximation is reasonable for large numbers, which this is not.)

3. In the limit of very large numbers, we would expect that a plot of $\ln|dN/dt|$ versus t is a straight line:

$$\ln\left|\frac{\Delta N(t)}{\Delta t}\right| \simeq \ln\left|\frac{\Delta N_0}{\Delta t}\right| - \frac{1}{\tau}\Delta t.$$

   This means that if we treat $\ln|\Delta N(t)/\Delta t|$ as the dependent variable and time Δt as the independent variable, we can use our linear fit results. Plot $\ln|\Delta N/\Delta t|$ versus Δt.

4. Make a least-squares fit of a straight line to your data and use it to determine the lifetime τ of the π meson. Compare your deduction to the tabulated lifetime of 2.6×10^{-8} s and comment on the difference.

5. Plot your best fit on the same graph as the data and comment on the agreement.

6. Deduce the goodness of fit of your straight line and the approximate error in your deduced lifetime. Do these agree with what your "eye" tells you?

9.4.4 Exercise: Fitting Heat Flow

The table below gives the temperature T along a metal rod whose ends are kept at fixed constant temperatures. The temperature is a function of the distance x along the rod.

x_i (cm)	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0
T_i (°C)	14.6	18.5	36.6	30.8	59.2	60.1	62.2	79.4	99.9

1. Plot up the data to verify the appropriateness of a linear relation

$$T(x) \simeq a + bx \tag{9.43}$$

2. Because you are not given the errors for each measurement, assume that the least-significant figure has been rounded off and so $\sigma \geq 0.05$. Use that to compute a least-squares, straight-line fit to these data.
3. Plot your best $a + bx$ on the curve with the data.
4. After fitting the data, compute the variance and compare it to the deviation of your fit from the data. Verify that about one-third of the points miss the σ error band (that is what is expected for a normal distribution of errors).
5. Use your computed variance to determine the χ^2 of the fit. Comment on the value obtained.
6. Determine the variances σ_a and σ_b and check if it makes sense to use them as the errors in the deduced values for a and b.

9.4.5 Nonlinear Fit of Breit–Wigner to Cross Section

Problem: Recall how we started Unit III of this chapter by interpolating the values in Tab. 9.1 giving the experimental cross section Σ as a function of energy. Although we did not use it, we also gave the theory describing these data, namely, the Breit–Wigner resonance formula (9.1):

$$f(E) = \frac{f_r}{(E - E_r)^2 + \Gamma^2/4} \tag{9.44}$$

Your **problem** is to determine what values for the parameters E_r, f_r, and Γ in (9.44) provide the best fit to the data in Tab. 9.1.

Because (9.44) is not a linear function of the parameters (E_r, Σ_0,Γ), the three equations that result from minimizing χ^2 are not linear equations and so cannot be solved by the techniques of *linear* algebra (matrix methods). However, in our study of the weights on a string problem in Unit I, we showed how to use the Newton–Raphson algorithm to search for solutions of simultaneous nonlinear equations. That technique involved expansion of the equations about the previous guess to obtain a set of linear equations, and then solving the linear equations with the matrix libraries discussed in Unit II. We now use this same combination of fitting, trial-and-error searching and matrix algebra, to conduct a nonlinear least-squares fit of (9.44) to the data in Tab. 9.1.

Recall that the condition for a best fit is to find values of the M_P parameters a_m in the theory $g(x, a_m)$ that minimize $\chi^2 = \sum_i [(y_i - g_i)/\sigma_i]^2$. This leads to the M_P equations (9.34) to solve:

$$\sum_{i=1}^{N_D} \frac{[y_i - g(x_i)]}{\sigma_i^2} \frac{\partial g(x_i)}{\partial a_m} = 0 \qquad (m = 1, M_P) \tag{9.45}$$

To find the form of these equations appropriate to our problem, we rewrite our theory function (9.44) in the notation of (9.45). We let

$$a_1 = f_r \qquad a_2 = E_R \qquad a_3 = \Gamma^2/4 \qquad x = E \tag{9.46}$$

$$\Rightarrow \qquad g(x) = \frac{a_1}{(x - a_2)^2 + a_3} \tag{9.47}$$

The three derivatives required in (9.45) are then

$$\frac{\partial g}{\partial a_1} = \frac{1}{(x - a_2)^2 + a_3} \qquad \frac{\partial g}{\partial a_2} = \frac{-2a_1(x - a_2)}{[(x - a_2)^2 + a_3]^2} \qquad \frac{\partial g}{\partial a_3} = \frac{-a_1}{[(x - a_2)^2 + a_3]^2}$$

Substitution of these derivatives into the best fit condition (9.45) yields the three simultaneous equations in a_1, a_2, and a_3 that we will need to solve in order to fit the $N_D = 9$ data points (x_i, y_i) in Tab. 9.1:

$$\sum_{i=1}^{9} \frac{y_i - g(x_i, a)}{(x_i - a_2)^2 + a_3} = 0 \qquad \sum_{i=1}^{9} \frac{y_i - g(x_i, a)}{[(x_i - a_2)^2 + a_3]^2} = 0,$$

$$\sum_{i=1}^{9} \frac{\{y_i - g(x_i, a)\}\,(x_i - a_2)}{[(x_i - a_2)^2 + a_3]^2} = 0 \tag{9.48}$$

Even without the substitution of (9.44) for $g(x, a)$, it is clear that these three equations depend on the a's in nonlinear fashions; this is what makes them hard to solve. However, we already know how to solve them! If you look back at Section 8.1.2, you will see that we derived there the method for conducting an N-dimensional Newton–Raphson search for the roots of

$$f_i(a_1, a_2, \ldots, a_N) = 0 \qquad i = 1, N \tag{9.49}$$

where we have made the change of variable $y_i \to a_i$ for the present problem. We use that same formalism for the $N = 3$ Eqs. (9.48) by writing them as

$$f_1(a_1, a_2, a_3) = \sum_{i=1}^{9} \frac{y_i - g(x_i, a)}{(x_i - a_2)^2 + a_3} = 0 \tag{9.50}$$

$$f_2(a_1, a_2, a_3) = \sum_{i=1}^{9} \frac{\{y_i - g(x_i, a)\}\,(x_i - a_2)}{[(x_i - a_2)^2 + a_3]^2} = 0 \tag{9.51}$$

$$f_3(a_1, a_2, a_3) = \sum_{i=1}^{9} \frac{y_i - g(x_i, a)}{[(x_i - a_2)^2 + a_3]^2} = 0 \tag{9.52}$$

Because $f_r \equiv a_1$ is the peak value of the cross section, $E_R \equiv a_2$ is the energy at which the peak occurs, and $\Gamma = 2\sqrt{a_3}$ is the full width of the peak at half maximum, good starting guesses for the a's can be extracted from a graph

of the data. All that is needed then to apply the Newton–Raphson matrix equations (8.16) to our problem are expressions for the derivatives of the three f's with respect to the three unknown a's. As before, it is easy to evaluate all nine derivatives by use of two nested loops over i and j, along with the forward-difference approximation for the derivative,

$$\frac{\partial f_i}{\partial a_j} \simeq \frac{f_i(a_j + \Delta a_j) - f_i(a_j)}{\Delta a_j} \tag{9.53}$$

where Δa_j corresponds to a small, say $\leq 1\%$, change in the parameter value.

9.4.5.1 Nonlinear Fit Implementation

Use the Newton–Raphson algorithm as outlined in Section 9.4.5 to conduct a nonlinear search for the best fit parameters of the Breit–Wigner theory (9.44) to the data in Tab. 9.1. Compare the deduced values of (f_r, E_R, Γ) to that obtained by inspection of the graph. The program `Newton_Jama.java` on the instructor's CD solves this problem.

9.5 Appendix: Calling LAPACK from C

Calling a Fortran-based matrix library from Fortran is less trouble than calling it from some other language. However, if you refuse to be a Fortran programmer, there are often C and Java implementations of the Fortran libraries available, such as JAMA, JLAPACK, LAPACK++, and TNTfor use from these languages.

Some of our research projects have required us to have Fortran and programs calling each other, and, after some experimentation, we have had success doing it under Unix. But care is needed in accounting for the somewhat different ways the compilers store subroutine names, for the quite different ways they store arrays with more than one subscript, and for the different data types available in the two languages.

Tab. 9.2 Matching data types in C and Fortran.

C	Fortran	C	Fortran
char	Character	unsigned int	Logical*4
signed char	Integer*1	float	Real (Real*4)
unsigned char	Logical*1	structure of 2 floats	Complex
short signed int	Integer*2	double	Real*8
short unsigned int	Logical*2	structure of 2 doubles	Complex*16
signed int (long int)	Integer*4	char[n]	Character*n

The first thing you must do is ensure that the data types of the variables in the two languages are matched. The matching data types are given in Tab. 9.2.

Note that if the data are stored in arrays, your C calling program must convert to the storage scheme used in the Fortran subroutine before you can call the Fortran subroutine (and then convert back to the C scheme after the subroutine does its job if you have overwritten the original array and intend to use it again).

When a function is called in the C language, usually the actual value of the argument is passed to the function. In contrast, Fortran passes the address in memory where the value of the argument is to be found (a *reference pass*). If you do not ensure that both languages have the same passing protocol, your program will process the numerical value of the address of a variable as if it were the actual value of the variable (we are willing to place a bet on the correctness of the result). Here are some procedures for **calling Fortran from C**:

1. Use pointers for all arguments in your C programs. Generally this is done with the address operator `&`.

2. Do not have your program make calls such as `sub(1, N)` where the actual value of the constant "`1`" is fed to the subroutine. Instead, assign the value one to a variable, and feed that variable (actually a pointer to it) to the subroutine. For example:

```
one = 1.                          // Assign value to variable
sub(one)                 // All Fortran calls are reference calls
sub(&one)                         // In C, make pointer explicit
```

   This is important because the value "`1`" in the subroutine call, in Fortran, anyway, is actually the address where the value "1" is stored. If the subroutine modifies that variable, it will modify the value of "`1`" at every place in your program! In C, it would change the first value in memory.

3. Depending on the particular operating system you are using, you may have to append an underscore _ to the called Fortran subprogram names. As a case in point, `sub(one, N)` → `sub_(one, N)`. Generally, the Fortran compiler appends an underscore automatically to the names of its subprograms, while the C compiler does not (but you will need to experiment).

4. Use lowercase letters for the names of external functions. The exception is when the Fortran subprogram being called was compiled with a `-U` option, or the equivalent, for retaining uppercase letters.

9.5.1 Calling LAPACK Fortran from C

```
// Calling LAPACK from C  to solve AX=B

#include <stdio.h>}                                    // I/O headers
#define size 100                        // Dimension of Hilbert matrix

main()  {
  int i, j , c1, c2, pivot[size], ok;
  double matrix[size][size], help[size*size], result[size];
                                        // Pointers for function call
  c1 = size;
  c2 = 1;                                          // Numbers as variables
  for (i = 0; i < c1; i++) {                      // Create Hilbert matrix
    for (j = 0; j < c1; j++)   matrix[i][j]  = 1.0/(i+j+1);
      result[i] = 1. / (i+1);                    // Create solution vector
    }
                                                      // Transform matrix
  for (i = 0; i < size; i++)
     for(j = 0; j < size; j++) help[j+size*i] = matrix[j][i];

  dgesv_(&c1, &c2, help, &c1, pivot, result, &c1, &ok);
  for (j = 0; j<size; j++) printf("%e\n", result[j]);
}
```

You will notice here that the call to the Fortran subroutine `dgesv` is made as

```
dgesv_(&c1, &c2, help, &c1, pivot, result, &c1, &ok)
```

That is, lowercase letters are used and an underscore is added to the subroutine name. In addition, we convert the matrix A,

```
for (i=0; i < size; i++)                          // Matrix transformation
  for (j=0; j < size; j++)  help[j+size*i] = matrix[j][i];
```

This changes C's row-major order to Fortran's column-major order using a scratch vector.

9.5.2 Compiling C Programs with Fortran Calls

Multilanguage programs actually get created when the compiler links the object files together. The tricky part is that while Fortran automatically includes its math library, if your final linking is done with the C compiler, you may have to explicitly include the Fortran library as well as others. Here we give some examples that have worked at one time or another, on one machine or another; you probably will need to read the User's Guide and experiment to get this to work with your system:

```
> cc -O call.c f77sub.o -lm -ldxml          DEC extended mathlibe
```

```
> cc -O call.c f77sub.o -L/usr/lang/SC0.0 -lF77          Link, SunOS
> cc -O c_fort c_fort.o area_f.o -lxlf                   Link, AIX
> gcc -Wall call.c /usr/lib/libm.a -o call               gcc explicit
> gcc -Wall call.c -lm -o call                           gcc shorthand version of above
```