Representation of functions

Chapter Contents

One of the most important applications of numerical analysis is the representation of numerical data in functional form. This includes representations of standard mathematical functions, fitting, smoothing, filtering, interpolating, *etc.* Related subjects are how to evaluate various functions efficiently, fitting data as polynomials or Fourier series, and fitting data in such a way as to minimize the average deviation.

1. Evaluating functions

A program may requirevalues of some mathematical function—sinθ, say—for arbitrary values of θ. The function may be moderately or extremely time-consuming to compute directly. According to the Intel timings for the 80x87 chip, computing sinθ takes about eight times longer than a floating point multiply. In some real-time applications this may be too slow.

There are several ways to speed up the computation of a function. They are all based on compact representations—either in tabular form or as coefficients of functions that are faster to evaluate. For example, we might represent sinθ by a simple polynomial[1]

$$\sin\theta \approx \theta\,(0.994108 - 0.147202\theta)\,, \qquad (1)$$

accurate to better than 1% over the range $-\frac{\pi}{2} \le \theta \le \frac{\pi}{2}$, that requires but two multiplications and an addition to evaluate. This would be 3-4 times faster than calculating sinθ on the 80x87 chip[2].

1. This comes from the Chebyshev polynomial representation for $\sin(x)$. See, *e.g.*, Abramowitz and Stegun, *HMF*, §4.3.104.
2. Although the 80x87 already uses a compact representation of the trigonometric functions and is thus fairly hard to beat, especially if high accuracy is demanded.

To achieve substantially greater speed requires table lookup. To locate data in an ordered table, programmers often employ binary search: that is, look at the θ-value halfway down the table and see if the desired value is greater or less than that. On the average, $\log_2(N)$ comparisons are required, where N is the length of the table. For a table with 1% precision, we might need 128 entries, *i.e.* up to seven comparisons.

Binary search can be unacceptably slow — is there a faster method? In fact, assuming an ordered table of equally-spaced abscissae the fastest way to locate the desired x-value is *hashing*, a method for *computing* the address rather than finding it using comparisons. Suppose, as before, we need 1% accuracy, *i.e.* a 128-point table with x in the range $[0,\pi/2]$. To look up a value, we multiply x by $256/\pi \cong 81.5$, truncate to an integer and quadruple it to get a (4-byte) floating point address. These operations take about 1.5-2 fp multiply times, hence the speedup is 4-fold.

The speedup factor does not seem like much, especially for a function such as $\sin\theta$ that is built in to many numeric co-processors. However, if we needed a function that is considerably slower to evaluate (for example one requiring evaluation of an integral, or solution of a differential equation) hashed table lookup with interpolation can be several orders of magnitude faster than direct evaluation.

Once we know the function values at abscissas bracketing the one we need, we must still interpolate in the table. Several forms of interpolation are commonly employed, depending on the precision desired, or on the desirability of some degree of smoothing.

Everyone is familiar with linear interpolation,

$$f(x_k + ph) \approx (1-p)\, f_k \;+\; p\, f_{k+1} + R \qquad (2)$$

where the spacing between successive points (in a uniformly spaced table) is

$$h = x_{k+1} - x_k \,,$$

the remainder is given approximately by

$$R \approx \frac{1}{8} h^2 f^{(2)}(\xi)$$

and ξ is a point in the interval $\left[x_k \,,\; x_{k+1}\right]$. To understand the origin of the remainder we compare the formula

$$f(x) \approx \left(1 - \frac{x}{h}\right) f_0 \;+\; \frac{x}{h} f_1 \;+\; R$$

with the first few terms of the Taylor's series expansion

$$f(x) \approx f_0 \;+\; x\, f'(x_0) \;+\; \frac{1}{2} x^2 f''(x_0)$$

to get

$$R \approx \frac{1}{2} x(x-h)\, f''(x_0)$$

and note that

$$|R| \le \frac{1}{2} \max_{0 \le x \le h} \Big(|x(x-h)| \;\; |f''(x)| \Big) = \frac{1}{8} h^2 \;\; |f''(\xi)| \,.$$

More generally we may represent the function in the region containing the points $x_1, \ldots, x_n$ by the Lagrange interpolation formula

$$f(x) \approx \sum_{k=1}^{n} \frac{\pi_k(x)\, f_k}{\pi_k(x_k)} \tag{3}$$

where

$$\pi_k(x) = \prod_{m \neq k} (x - x_m) .$$

We notice this formula goes through each of the points x_m .

Higher order Lagrange interpolation is rarely used—it is better to decrease the spacing of the table (and thereby its demand for storage) if higher precision is required.

A widely employed technique is cubic spline interpolation[3]. Suppose we have a table of values f_k at abscissas x_k. Then if we define

$$A_k(x) = \frac{x_{k+1} - x}{x_{k+1} - x_k} = \frac{x_{k+1} - x}{\Delta_k} , \qquad B_k(x) = 1 - A_k(x) \equiv \frac{x_k - x}{x_k - x_{k+1}} = \frac{x - x_k}{\Delta_k}$$

we may write down by inspection a cubic polynomial,

$$p_k(x) = A_k(x)\, f_k \; + \; B_k(x)\, f_{k+1} \; + \; \tfrac{1}{6}\alpha_k \left(A_k^3 - A_k\right)\Delta_k^2 \; + \; \tfrac{1}{6}\beta_k \left(B_k^3 - B_k\right)\Delta_k^2 ,$$

that is unique (up to two undetermined constants α_k and β_k) and passes through the points $\left(x_k\,,\; f_k\right), \left(x_{k+1}\,,\; f_{k+1}\right)$. The undetermined parameters may be used to require that the derivatives of the fitting polynomials match at the endpoints, *i.e.*

$$p'_k(x_{k+1}) = p'_{k+1}(x_{k+1}) \; , p''_k(x_{k+1}) = p''_{k+1}(x_{k+1}) .$$

From the second of these conditions we see that

$$\beta_k = \alpha_{k+1} \equiv f''_{k+1} ;$$

that is, the α's are the second derivatives of the function. The first condition yields the two-term recursion relation

$$\tfrac{1}{6}\alpha_k\, \Delta_k \; + \; \tfrac{1}{3}\alpha_k \left(\Delta_k + \Delta_{k+1}\right) + \; \tfrac{1}{6}\alpha_{k+2}\, \Delta_{k+1} \; = \; \frac{f_{k+2} - f_{k+1}}{\Delta_{k+1}} \; - \; \frac{f_{k+1} - f_k}{\Delta_k} .$$

---

3. Our discussion here will follow that of William H. Press, Saul A. Teukolsky and William T. Vetterling *Numerical Recipes in Fortran : The Art of Scientific Computing* (Cambridge U. Press, Cambridge, 1992).

Since k runs from 1 to N the above represent N–2 equations in N unknowns, hence two additional conditions must be imposed in order to get a unique fit. These are usually impsed at the endpoints and consist either of setting $\alpha_1 = \alpha_N = 0$ ("natural" spline fit) or of giving the first derivatives specific values at the endpoints (the latter is usually the case in CAD programs). The linear equations can then be solved (in $\mathcal{O}(N)$ time) using a standard method suited to tridiagonal matrices. A FORTRAN program for cubic spline fitting is given below:

```
      SUBROUTINE SPLINE(X,Y,N,YP1,YPN,Y2)
      PARAMETER (NMAX=100)
      DIMENSION X(N),Y(N),Y2(N),U(NMAX)
      IF (YP1.GT..99E30) THEN
        Y2(1)=0.
        U(1)=0.
      ELSE
        Y2(1)=-0.5
        U(1)=(3./(X(2)-X(1)))*((Y(2)-Y(1))/(X(2)-X(1))-YP1)
      ENDIF

      DO 11 I=2,N-1
        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)=(6.*((Y(I+1)-Y(I))/(X(I+1)-X(I))-(Y(I)-Y(I-1))
     *      /(X(I)-X(I-1)))/(X(I+1)-X(I-1))-SIG*U(I-1))/P
11    CONTINUE
      IF (YPN.GT..99E30) THEN
        QN=0.
        UN=0.
      ELSE
        QN=0.5
        UN=(3./(X(N)-X(N-1)))*(YPN-(Y(N)-Y(N-1))/(X(N)-X(N-1)))
      ENDIF
      Y2(N)=(UN-QN*U(N-1))/(QN*Y2(N-1)+1.)
      DO 12 K=N-1,1,-1
        Y2(K)=Y2(K)*Y2(K+1)+U(K)
12    CONTINUE
      RETURN
      END
```

2. Representing measurements

We now consider how to represent data by mathematical functions. This can be useful in several contexts:

- The theoretical form of the function, but with unknown parameters, may be known. One might like to *determine* the parameters from the data. For example, one might have a lot of data on pendulums: their periods, masses, dimensions, *etc*. The period of a pendulum is given, theoretically, by

$$\tau = \left(\frac{2\pi L}{g}\right)^{1/2} f\left(\frac{L}{r}, \frac{m_{bob}}{m_{string}}, \ldots\right) \tag{4}$$

  where L is the length of the string, g the acceleration of gravity, and f is some function of ratios of typical lengths, masses and other factors in the problem. In order to determine g accurately, one generally fits a function of all the measured factors, and tries to minimize its deviation from the measured periods. That is, one might try

$$\tau_n = \left(\frac{2\pi L_n}{g}\right)^{1/2} \left[1 + \alpha\frac{r_n}{L_n} + \beta\left(\frac{m_{bob}}{m_{string}}\right)_n + \ldots\right] \tag{5}$$

  for the nth set of observations, with g, α, β, ... the unknown parameters to be determined.

- Sometimes one knows that a phenomenon is basically smoothly varying; so that the wiggles and deviations in observations are noise or otherwise uninteresting. How can we filter out the noise without losing the significant part of the data? Several methods have been developed for this purpose, based on the same principle: the data are represented as a sum of functions from a **complete** set of functions, with unknown coefficients. That is, if $\varphi_m(x)$ are the functions, we say (y_n are the data)

$$y_n = \sum_{m=0}^{\infty} c_m \varphi_m(x_n) \tag{6}$$

  Such representations are theoretically possible under general conditions. Then to filter we keep only a finite sum, retaining the first N (usually simplest and smoothest) functions from the set. An example of a complete set is *monomials*, $\varphi_m(x) = x^m$. Another is sinusoidal (trigonometric) functions,

$$\sin(2\pi mx), \quad \cos(2\pi mx), \quad 0 \le x \le 1,$$

  used in Fourier-series representation. *Gram polynomials*, discussed below, comprise a third useful complete set.

The representation in Eq. 6 is called *linear* because the unknown coefficients c_m appear to their first power. Thus, if all the data were to double, we see immediately that the c_m's would have to be multiplied by the same factor, 2.

Sometimes, as in the example of the measurement of g above, the unknown parameters appear in more complicated fashion. The problem of fitting with these more general functional forms is called *nonlinear* for obvious reasons. The *simplex algorithm* is an example of a nonlinear fitting procedure.

We are now going to discuss fitting both linear and nonlinear functions to data. The first and conceptually simplest of these is the Fourier transform, namely representing a function as a sum of sines and cosines. Such a representation can be made the basis of *digital filter* routines.

Fast Fourier transform

What is a Fourier transform? Suppose we have a function that is *periodic* on the interval $0 \le x \le 2\pi$:

$$f(x + 2\pi) = f(x) ;$$

Then under fairly general conditions the function can be expressed in the form

$$f(x) = a_0 + \sum_{n=1}^{\infty} \left(a_n \cos(nx) + b_n \sin(nx) \right) \tag{7}$$

Another way to write Eq. 7 is

$$f(x) = \sum_{-\infty}^{+\infty} c_n e^{inx}. \tag{8}$$

In either way of writing, the c_n are called Fourier coefficients of the function $f(x)$. Looking at Eq. 8, we see that the orthogonality of the sinusoidal functions leads to the expression

$$c_n = \frac{1}{2\pi} \int_0^{2\pi} f(x)\, e^{-inx}\, dx \, . \tag{9}$$

Evaluating Eq. 9 numerically requires—for given n—at least $2n$ points[4]. Naively, for each n=0 to N–1 we have to do a sum

$$c_n \approx \sum_{k=1}^{2N} f_k\, e^{-2\pi i n k/N}$$

which means carrying out $2N^2$ complex multiplications.

The fast Fourier transform (FFT) was discovered by Runge and König, rediscovered by Danielson and Lanczos and re-rediscovered by Cooley and Tukey[5]. The FFT algorithm can be expressed as three steps:

To evaluate rapidly the polynomial

4. to prevent *aliasing*.
5. See, *e.g.*, D.E. Knuth, *The Art of Computer Programming*, v. 2 (Addison-Wesley Publishing Co., Reading, MA, 1981) p. 642.

$$c_n = \mathrm{P_N}\,(w_n) \equiv \sum_{k=0}^{N-1} f_k\,(w_n)^k$$

we divide it into two polynomials of order $N/2$, dividing each of those in two, *etc*. This procedure is efficient only for $N = 2^\nu$, with ν an integer, so this is the case we attack.

How does dividing a polynomial in two help us? If we segregate the odd from the even powers, we have, symbolically,

$$P_N(w) = E_{N/2}(w^2) + w\,O_{N/2}(w^2)\ . \tag{10}$$

Suppose the time to evaluate $\mathrm{P_N}(w)$ is T_N. Then, clearly,

$$T_N = \lambda + 2T_{N/2} \tag{11}$$

where λ is the time to segregate the coefficients into odd and even, plus the time for 2 multiplications and a division. The solution of Eq. 11 is $\lambda(N-1)$. That is, it takes $O(N)$ time to evaluate a polynomial.

However, the discreteness of the Fourier transform helps us here. The reason is this: to evaluate the transform, we have to evaluate $P_N\,(w_n)$ for N values of w_n . But w_n^2 takes on only $N/2$ values as n takes on N values. Thus to evaluate the Fourier transform for all N values of n, we can evaluate the two polynomials of order $N/2$ for half as many points.

Suppose we evaluated the polynomials the old-fashioned way: it would take $2(N/2) \equiv N$ multiplications to do both, but we need do this only $N/2$ times, and N more (to combine them) so we have $\frac{N^2}{2} + N$ rather than N^2. We have gained a factor 2. Obviously it pays to repeat the procedure, dividing each of the sub-polynomials in two again, until only monomials are left.

Symbolically, the number of multiplications needed to evaluate a polynomial for N (discrete) values of w is

$$\tau_N = N\lambda + 2\ \tau_{N/2} \tag{12}$$

whose solution is

$$\tau_N = \lambda N\ \log_2(N)\ . \tag{13}$$

Although the FFT algorithm can be programmed recursively, it almost never is. To see why, imagine how the coefficients would be re-shuffled by Eq. 10: we work out the case for 16 coefficients, exhibiting them in the table below, writing only the indices:

Bit-reversal for re-ordering discrete data prior to FFT					
Start	Step 1	Step 2	Step 3	Binary_0	Binary_3
0	0	0	0	0000	0000
1	2	4	8	0001	1000
2	4	8	4	0010	0100
3	6	12	12	0011	1100
4	8	2	2	0100	0010
5	10	6	10	0101	1010
6	12	10	6	0110	0110
7	14	14	14	0111	1110
8	1	1	1	1000	0001
9	3	5	9	1001	1001
10	5	9	5	1010	0101
11	7	13	13	1011	1101
12	9	3	3	1100	0011
13	11	7	11	1101	1011
14	13	11	7	1110	0111
15	15	15	15	1111	1111

The crucial columns are "Start" and "Step 3". Unfortunately, they are written in decimal notation, which conceals a fact that becomes glaringly obvious in binary notation. So we re-write them in binary in the columns Binary_0 and Binary_3—and see that the final order can be obtained from the initial order simply by reversing the order of the bits, from left to right!

Now, how do we go about evaluating the sub-polynomials to get the answer? First, let us write the polynomials (for our case $N=16$) corresponding to taking the (bit-reversed) addresses off the stack in succession, as shown below.

$$
\left.\begin{array}{l}
\left.\begin{array}{l}
\left.\begin{array}{l} w^7(f_7 + w^8 f_{15}) \\ w^3(f_3 + w^8 f_{11}) \end{array}\right\} w^3(a_3 + w^4 a_7) \\
\left.\begin{array}{l} w^5(f_5 + w^8 f_{13}) \\ w^1(f_1 + w^8 f_9) \end{array}\right\} w^1(a_1 + w^4 a_5)
\end{array}\right\} w^1(b_1 + w^2 b_3) \\
\left.\begin{array}{l}
\left.\begin{array}{l} w^6(f_6 + w^8 f_{14}) \\ w^2(f_2 + w^8 f_{10}) \end{array}\right\} w^2(a_2 + w^4 a_6) \\
\left.\begin{array}{l} w^4(f_4 + w^8 f_{12}) \\ w^0(f_0 + w^8 f_8) \end{array}\right\} w^0(a_0 + w^4 a_4)
\end{array}\right\} w^0(b_0 + w^2 b_2)
\end{array}\right\} c_0 + w c_1 \qquad (14)
$$

We see that w_n^8 (for $N=16$) has only two possible values, ± 1. Thus we must evaluate not 16×8 terms like $f_i + w^8 f_{i+8}$, but only 2×8. Similarly, we do not need to evaluate 16×4 terms of form $f_i + w^4 f_{i+4}$, but only 4×4, since there are only 4 possible values of w_n^4. Thus the total number of multiplications is

$$2\times 8 + 4\times 4 + 8\times 2 + 16\times 1 = 64 \equiv 16 \log_2 16,$$

as advertised. This is far fewer than 16×16=256, and the ratio improves with N — for example a 1024 point FFT is 100 times faster than a slow FT.

Gram polynomials

Gram polynomials are useful in fitting data by the linear least- squares method. The usual method is based on the following question: What is the "best" polynomial,

$$P_N(x) = \sum_{n=0}^{N} \gamma_n x^n, \tag{15}$$

(of order N) that we can use to fit some set of M pairs of data points,

$$\begin{Bmatrix} x_k \\ f_k \end{Bmatrix}, \quad k=0, 1, \ldots, M-1$$

(with $M > N$) where $f(x)$ is measured at M distinct values of the independent variable x ?

The usual answer, found by Gauss, is to minimize the squares of the deviations (at the points x_k) of the fitting function $P_N(x)$ from the data —possibly weighted by the uncertainties of the data. That is, we want to minimize the statistic

$$\chi^2 = \sum_{k=0}^{M-1} \left(f_k - \sum_{n=0}^{N} \gamma_n (x_k)^n \right)^2 \frac{1}{\sigma_k^2} \tag{16}$$

with repect to the $N+1$ parameters γ_n .

From the differential calculus we know that a function's first derivative vanishes at a minimum, hence we differentiate χ^2 with respect to each γ_n independently, and set the results equal to zero. This yields $N+1$ linear equations in $N+1$ unknowns:

$$\sum_m A_{nm} \gamma_m \overset{df}{=} \beta_n, \quad n=0, 1, \ldots, N \tag{17}$$

where the symbol $\overset{df}{=}$ means "is defined by".

We shall develop methods for solving linear equations. Unfortunately, they cannot be applied to Eq. 17 for $N \gtrsim 9$ because the matrix A_{nm} approximates a Hilbert matrix,

$$H_{nm} = \frac{const.}{n+m+1},$$

a particularly virulent example of an exponentially ill-conditioned matrix. That is, the roundoff error in solving Eq. 17 grows exponentially with N, and is generally unacceptable.

We can avoid roundoff problems by expanding in polynomials rather than monomials:

$$\chi^2 = \sum_{k=0}^{M-1} \left(f_k - \sum_{n=0}^{N} \gamma_n p_n(x_k) \right)^2 \frac{1}{\sigma_k^2}. \tag{18}$$

The matrix we must invert then becomes

$$A_{nm} = \sum_{k=0}^{M-1} p_n(x_k)\, p_m(x_k)\, \frac{1}{\sigma_k^2} \tag{19a}$$

and the inhomogeneous term is now

$$\beta_n = \sum_{k=0}^{M-1} p_n(x_k)\, f_k\, \frac{1}{\sigma_k^2} \tag{19b}$$

Is there any choice of the polynomials $p_n(x)$ that will eliminate the ill-conditioning problem (*i.e.* roundoff error)? The best kinds of linear equations are those with nearly diagonal matrices. We note the sum in Eq. 19a is nearly an integral, if M is large. If we choose the polynomials so they are orthogonal with respect to the weight function

$$w(x) = \frac{1}{\sigma_k^2}\, \theta(x_k - x)\, \theta(x - x_{k-1}),$$

where

$$\theta(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

then A_{nm} will be nearly diagonal, and well-conditioned.

Orthogonal polynomials play an important role in numerical analysis and applied mathematics. They satisfy orthogonality relations[6] of the form

6. Polynomials can be thought of as vectors in a space of infinitely many dimensions ("Hilbert" space). Certain polynomials are like the vectors that point in the (mutually orthogonal) directions in ordinary 3-dimensional space, and so are called **orthogonal** by analogy.

$$\int_A^B dx\, w(x) p_n(x)\, p_m(x) = \delta_{nm} \equiv \begin{cases} 1, & m=n \\ 0, & m\neq n \end{cases} \tag{20}$$

where the weight function $w(x)$ is positive.

For a given $w(x)$ and interval [A,B], we can construct orthogonal polynomials using the Gram-Schmidt orthogonalization process.

Denote the integral in Eq. 20 by $(p_n\ ,\ p_m\ )$ to save having to write it many times. We start with

$$p_{-1} = 0\ ,$$
$$p_0(x) = \left(\int_A^B dx\ w(x) \right)^{-1/2} = \text{const.}\ ,$$

and assume the polynomials satisfy the 2-term upward recursion relation

$$p_{n+1}(x) = \Big(a_n + x b_n \Big) p_n(x)\ +\ c_n\, p_{n-1}(x) \tag{21}$$

Now apply Eq. 21: assume we have calculated p_n and p_{n-1} and want to calculate p_{n+1}. Clearly, the orthogonality property gives

$$(p_{n+1}\ ,\ p_n\ ) = (p_{n+1}\ ,\ p_{n-1}\ ) = (p_n\ ,\ p_{n-1}\ ) = 0\ ,$$

and the assumed normalization gives

$$(p_n\ ,\ p_n\ ) = 1\ .$$

These relations yields two equations for the three unknowns, a_n, b_n and c_n:

$$a_n + b_n\, (p_n\ ,\ x p_n) = 0$$

$$c_n + b_n\, (p_n\ ,\ x\, p_{n-1}) = 0$$

We express a_n and c_n in terms of b_n to get

$$p_{n+1}(x) = b_n \Big[(x - (p_n\ ,\ x\, p_n\ ))\ p_n(x)\ -\ (p_n\ ,\ x\, p_{n-1}\ )\, p_{n-1}(x) \Big] \tag{22}$$

We determine the remaining parameter b_n by again using the normalization condition:

$$(p_{n+1}\ ,\ p_{n+1}\ ) = 1.$$

In practice, we pretend $b_n = 1$ and evaluate Eq. 22; then we calculate

$$b_n = (\overline{p}_{n+1}\ ,\ \overline{p}_{n+1}\ )^{-1/2}, \tag{23}$$

multiply the (un-normalized) $\overline{p}_{n+1}$ by b_n , and continue.

The process of successive orthogonalization guarantees that p_n is orthogonal to all polynomials of lesser degree in the set. Why is this so? By construction, $p_{n+1} \perp p_n$ and p_{n+1}. Is it $\perp p_{n-2}$? We need to ask whether

$$\left(p_n ,\ (x - \alpha_n)\, p_{n-2} \right) = 0 .$$

But we know that any polynomial of degree N-1 can be expressed as a linear combination of independent polynomials of degrees 0, 1, … , N-1. Thus

$$(x - \alpha_n)\, p_{n-2} \equiv \sum_{k=0}^{n-1} \mu_k\, p_k(x) \qquad (24)$$

and (by hypothesis) $p_n \perp$ every term of the rhs of Eq. 24, hence it follows (by mathematical induction) that

$$p_{n+1} \perp \left\{ p_{n-2} , p_{n-3} , \ldots \right\} .$$

Let us illustrate the process for Legendre polynomials, defined by weight $w(x) = 1$, interval [-1,1]:

$$p_0 = \left(\frac{1}{2} \right)^{1/2} ,$$

$$p_1 = \left(\frac{3}{2} \right)^{1/2} x ,$$

$$p_2 = \left(\frac{5}{2} \right)^{1/2} \left(\frac{3}{2} x^2 - \frac{1}{2} \right),$$

……

These are in fact the first three (normalized) Legendre polynomials, as any standard reference will confirm.

Now we can discuss Gram polynomials. While orthogonal polynomials are usually defined with respect to an integral as in Eq. 20, we might also define orthogonality in terms of a sum, as in Eq. 19a. That is, suppose we define the polynomials such that

$$\sum_{k=0}^{M-1} p_n(x_k)\, p_m(x_k)\, \frac{1}{\sigma_k^2} \equiv \delta_{nm} = \begin{cases} 1, & m = n \\ 0, & m \neq n \end{cases} \qquad (25)$$

Then we can construct the Gram polynomials, calculating the coefficients by the algebraic steps of the Gram-Schmidt process, except now we evaluate sums rather than integrals. Since $p_n(x)$ satisfies Eq. 25 by construction, the coefficients γ_n in our fitting polynomial are simply

$$\gamma_n = \sum_{k=0}^{M-1} p_n(x_k)\, f_k \frac{1}{\sigma_k^2} \ ; \tag{26}$$

they can be evaluated without solving *any* coupled linear equations, ill-conditioned or otherwise. Roundoff error thus becomes irrelevant.

In practice, we would never wish to fit a polynomial of order comparable to the number of data, since this would include the noise as well as the significant information.

We therefore calculate a statistic called χ^2/(*degree of freedom*)[7]: With M data points and an N'th order polynomial, there are M-N-1 degrees of freedom. That is, we evaluate Eq. 18 for fixed N, and divide by M-N-1. We then increase N by 1 and do it again. The value of N to stop at is the one where

$$\sigma^2_{M,N} = \frac{\chi^2_{M,N}}{M-N-1}$$

stops decreasing (with N) and begins to increase.

The best thing about the $\chi^2_{M,N}$ statistic is we can increase N without having to do any extra work:

$$\chi^2_{M,N} = \sum_{k=0}^{M-1} \left(f_k - \sum_n \gamma_n\, p_n(x_k) \right)^2 w_k \equiv \sum_{k=0}^{M-1} (f_k)^2 - \sum_{n=0}^{N} (\gamma_n)^2 \tag{27}$$

7. That is, "chi-squared per degree of freedom".

3. Chebyshev approximation

The Chebyshev polynomial of degree n is given by

$$T_n(x) \overset{df}{=} \cos\left(n \cos^{-1} x\right) \tag{28}$$

and satisfies the recurrence relation

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x), \quad n \geq 1 .$$

Manifestly,

$$T_0 = 1$$
$$T_1 = x$$
$$T_2 = 2x^2 - 1$$
$$\cdots$$

From Eq. 28 it is obvious that $\left|T_n(x)\right| \leq 1$.

The polynomial $T_N(x)$ has N zeros in the interval $(-1, +1)$, located at

$$x_k^{(N)} = \cos\left(\frac{\pi\,(k - \frac{1}{2})}{N}\right), \quad k = 1, 2, \ldots, n .$$

Since the Chebyshev polynomials satisfy the discrete orthogonality relation

$$\sum_{k=1}^{N} T_m(x_k)\, T_n(x_k) = \begin{cases} 0, & m \neq n \\ N/2, & m = n \neq 0 \\ N, & m = n = 0 \end{cases} \tag{29}$$

(here $m, n \leq N$), the Chebyshev polynomials are in fact the Gram polynomials of the preceding Section, for the case of equally spaced abscissas x_k and equal weights $1/\sigma_k^2$.

An arbitrary function $f(x)$ may be expanded in the form

$$f(x) \approx \frac{1}{2}a_0 + \sum_{k=1}^{N-1} a_k\, T_k(x) \tag{30}$$

where

$$a_n = \frac{2}{N} \sum_{k=0}^{N-1} f\left[\cos\left(\frac{\pi\,(k+\frac{1}{2})}{N}\right)\right] T_n\left[\cos\left(\frac{\pi\, n\,(k+\frac{1}{2})}{N}\right)\right].$$

With the above definitions, the approximation is *exact* at the zeros

$$x_k = \cos\left(\frac{\pi\,(k+\frac{1}{2})}{N}\right), \quad k = 0, 1, \ldots, N-1$$

of T_N.

Now for many functions the a_n's decrease rapidly with n. Thus if we truncate the sum in Eq. 30 at some $m < N-1$, the error will be dominated by the coefficient a_{m+1} of the first neglected term. This is not the average error (in the sense of fitting to minimize χ^2) but rather the *maximum* error since $\left| T_{m+1}(x) \right| \leq 1$

The virtue of fitting with Chebyshev polynomials is that the error can be distributed over the entire interval rather than being concentrated at one or another end—as would be the case with a truncated series expansion[8].

As an example, consider[9] fitting $\cos x$ on the interval $\left[\frac{-\pi}{2}, \frac{\pi}{2}\right]$. We note that the power series expansion is

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots - \ldots$$

The graph below plots the cosine function, the first four terms of the series expansion, the first neglected term ($x^8/8!$ —multiplied by 10^3 to fit it on the same scale), and the error function of a 3-term Chebyshev fit (see discussion below), on the interval $[0, \pi/2]$.

We note that all the error (represented by the first neglected term, $x^8/8!$) appears at the end of the interval and is bounded by 9.2×10^{-4}. That is, we must evaluate a cubic polynomial (in x^2) to get a precision of about one part in 1000.

There are two ways we might apply the Chebyshev approximation. If the labor of evaluating a cubic polynomial is acceptable, we can find a better cubic with much smaller error by approximating $x^8/8!$ as a sum of Chebyshev polynomials

$$x^8 = \left(\frac{\pi}{2}\right)^8 \sum_{n=0}^{4} c_n T_{2n}\left(\frac{2x}{\pi}\right),$$

then neglecting the T_8 term. Since the latter's coefficient is $\frac{1}{128\times 8!}\left(\frac{\pi}{2}\right)^8 = 7.18\ldots\times10^{-6}$, the improved cubic has error two orders of magnitude tahn the truncated power series.

Alternatively one might represent the term $x^6/6!$ by Chebyshev polynomials, discarding the T_6 term and achieving thereby the fourth-order representation

$$\cos x \approx 0.99935 - 0.49524x^2 + 0.03653\,x^4 \qquad (31)$$

with precision roughly 10^{-3} as before—but somewhat faster to calculate.

---

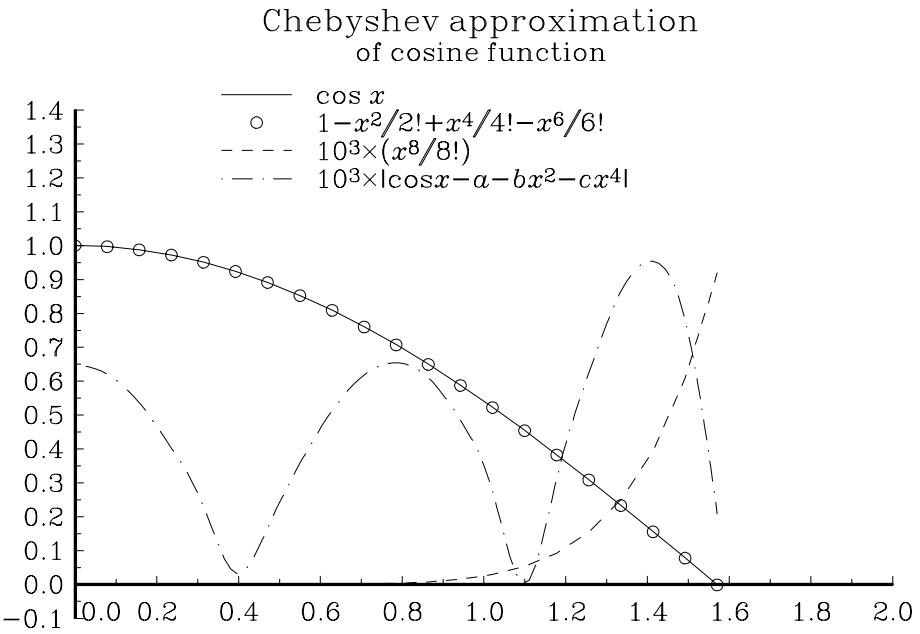
8. See, *e.g.*, C. Hastings, Jr., *Approximations for Digital Computers* (Princeton U. Press, Princeton, 1955.)
9. see, *e.g.*, Abramowitz & Stegun, *op. cit.*, p. 76.

To illustrate what has been achieved, the dot-dashed line in the preceding figure represents the error

$$E(x) \overset{df}{=} \left|\cos x - \left(0.99935 - 0.49524x^2 + 0.03653\,x^4\right)\right|$$

(multiplied by 1000 to display it on the same scale). This might appear at first blush like magic—we have found an approximation that is no less precise using a lower order polynomial! But of course, despite appearances, we have not actually gotten something for nothing—the error is now distributed uniformly over the interval, whereas previously it was concentrated at one end.

4. Function minimization



Sometimes we must fit data by a function that depends nonlinearly on its parameters. Consider

$$f_k \approx F\left[1 + e^{\alpha(x_k - X)}\right]^{-1}. \tag{32}$$

Although the dependence on the parameter F is linear, that on the parameters α and X is decidedly *non*linear. Several strategies can be used in such cases. One way to handle a problem like fitting Eq. 32 might be to transform the data, to make the dependence on the parameters linear. That is, we re-express Eq. 32 in the form

$$\log\left(\frac{F}{f_k} - 1\right) \approx \alpha\,(x_k - X)\,. \tag{33}$$

In some cases (for example if the value of F were known in advance rather than having to be determined from the data themselves) this might be possible, but in Eq. 32 no transformation will render linear the dependence on all three parameters at once. Nevertheless putting the problem in the form Eq. 33 might simplify the labor of determining the three parameters, so this avenue ought to be explored.

Thus we are frequently confronted with having to minimize numerically a complicated function of several parameters. Let us denote these by θ_0, θ_1, … , θ_{N-1}, and denote their possible range of variation by $\mathbf{R}$. Then we want to find those values of $\{\theta\} \subset \mathbf{R}$ that minimize a positive function:

$$\chi^2\left(\bar{\theta}_0, \ldots, \bar{\theta}_{N-1}\right) = \min_{\{\theta\}\subset \mathbf{R}} \chi^2\left(\theta_0, \ldots \theta_{N-1}\right). \tag{34}$$

One way to accomplish the minimization uses calculus, *via* the method of *steepest descents*. The idea is to differentiate the function χ^2 with respect to each θ_k, and to set the resulting N equations equal to zero, solving for the N $\bar{\theta}$'s. This is generally a pretty tall order, hence various approximate, iterative techniques have been developed. The simplest just steps along in θ-space, along the direction of the local downhill gradient $-\nabla\chi^2$, until a minimum is found. Then a new gradient is computed, and a new minimum sought[10].

Aside from the labor of computing $-\nabla\chi^2$, steepest descents has two main drawbacks: first, it only guarantees to find *a* minimum, not necessarily the absolute minimum—if a function has several local minima, steepest descents will not necessarily find the lowest. Worse, consider a function that has a minimum in the form of a steep-sided gulley that winds slowly downhill to a declivity—somewhat like the channel of a meandering river. A naive steepest descents routine will then spend all its time bouncing up and down the banks of the gulley, rather than proceeding along its bottom, since the steepest gradient is always nearly perpendicular to the line of the channel.

Sometimes the function χ^2 is so complex that its gradient is too expensive to compute. Can we find a minimum *without* evaluating partial derivatives? Several algorithms that do this have been devised. Here we explore two of them: the *simplex method* and *simulated annealing*.

10. This is not by itself very useful. Useful modifications can be found in Press, *et al*., *Numerical Recipes*, *ibid*., p. 301ff.

The simplex method

The idea behind the simplex method is to construct a simplex—a set of $N+1$ distinct and non-degenerate[11] vertices in the N-dimensional θ-space. We evaluate the function to be minimized at each of the vertices, and sort the table of vertices by the size of χ^2 at each vertex, the best (smallest χ^2) on top, the worst at the bottom. The simplex algorithm then chooses a new point in θ-space using a strategy that in action somewhat resembles the behavior of an amoeba seeking its food.

A standard FORTRAN subroutine for the simplex method has the disadvantages of being more than a page long and containing deeply nested control structures. It is thus hard to decipher—the

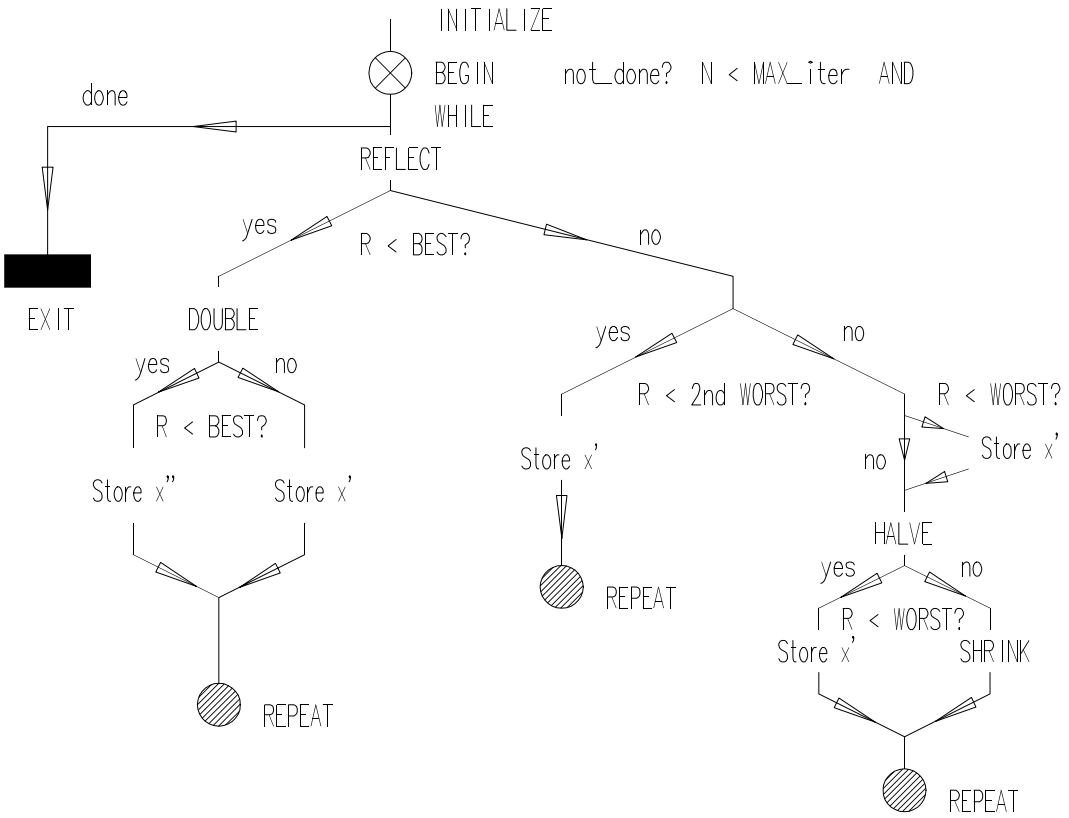
```
      SUBROUTINE AMOEBA(P,Y,MP,NP,NDIM,FTOL,FUNK,ITER)
    C from Press, et al., "Numerical Recipes", p. 292.
      PARAMETER (NMAX=20,ALPHA=1.0,BETA=0.5,GAMMA=2.0,ITMAX=500)
      DIMENSION P(MP,NP),Y(MP),PR(NMAX),PRR(NMAX),PBAR(NMAX)
      MPTS=NDIM+1
      ITER=0
 1    ILO=1
      IF(Y(1).GT.Y(2))THEN
        IHI=1
        INHI=2
      ELSE
        IHI=2
        INHI=1
      ENDIF
      DO 11 I=1,MPTS
        IF(Y(I).LT.Y(ILO)) ILO=I
        IF(Y(I).GT.Y(IHI))THEN
          INHI=IHI
          IHI=I
        ELSE IF(Y(I).GT.Y(INHI))THEN
          IF(I.NE.IHI) INHI=I
        ENDIF
 11    CONTINUE
      RTOL=2.*ABS(Y(IHI)-Y(ILO))/(ABS(Y(IHI))+ABS(Y(ILO)))
      IF(RTOL.LT.FTOL)RETURN
      IF(ITER.EQ.ITMAX) PAUSE 'Amoeba exceeding maximum iterations.'
      ITER=ITER+1
      DO 12 J=1,NDIM
        PBAR(J)=0.
 12    CONTINUE
      DO 14 I=1,MPTS
        IF(I.NE.IHI)THEN
          DO 13 J=1,NDIM
            PBAR(J)=PBAR(J)+P(I,J)
 13        CONTINUE
        ENDIF
```

corresponding flow chart (see below) took some time to construct.

11. "Non-degenerate" means the geometrical object, formed by connecting the $N+1$ vertices with straight lines, has non-zero N-dimensional volume; for example, if $N=2$, the simplex is a triangle.

```
14    CONTINUE
     DO 15 J=1,NDIM
       PBAR(J)=PBAR(J)/NDIM
       PR(J)=(1.+ALPHA)*PBAR(J)-ALPHA*P(IHI,J)
15    CONTINUE
     YPR=FUNK(PR)
     IF(YPR.LE.Y(ILO))THEN
       DO 16 J=1,NDIM
         PRR(J)=GAMMA*PR(J)+(1.-GAMMA)*PBAR(J)
16      CONTINUE
       YPRR=FUNK(PRR)
       IF(YPRR.LT.Y(ILO))THEN
         DO 17 J=1,NDIM
           P(IHI,J)=PRR(J)
17        CONTINUE
         Y(IHI)=YPRR
       ELSE
         DO 18 J=1,NDIM
           P(IHI,J)=PR(J)
18        CONTINUE
         Y(IHI)=YPR
       ENDIF
     ELSE IF(YPR.GE.Y(INHI))THEN
       IF(YPR.LT.Y(IHI))THEN
         DO 19 J=1,NDIM
           P(IHI,J)=PR(J)
19        CONTINUE
         Y(IHI)=YPR
       ENDIF
       DO 21 J=1,NDIM
         PRR(J)=BETA*P(IHI,J)+(1.-BETA)*PBAR(J)
21      CONTINUE
       YPRR=FUNK(PRR)
       IF(YPRR.LT.Y(IHI))THEN
         DO 22 J=1,NDIM
           P(IHI,J)=PRR(J)
22        CONTINUE
         Y(IHI)=YPRR
       ELSE
         DO 24 I=1,MPTS
           IF(I.NE.ILO)THEN
             DO 23 J=1,NDIM
               PR(J)=0.5*(P(I,J)+P(ILO,J))
               P(I,J)=PR(J)
23           CONTINUE
             Y(I)=FUNK(PR)
           ENDIF
24        CONTINUE
       ENDIF
     ELSE
       DO 25 J=1,NDIM
         P(IHI,J)=PR(J)
25      CONTINUE
       Y(IHI)=YPR
     ENDIF
     GO TO 1
     END
```

If the FORTRAN is translated directly to a more structured form in, say, Forth (with the various operations on the simplex factored out into subroutines) the indefinite outer loop (simulated in the FORTRAN routine by the penultimate GOTO 1 statement) appears as a BEGIN… WHILE…RE-



PEAT loop that is not nearly so long. Nevertheless, the triply nested IF… ELSE…THENs make the control logic hard to follow.

```
: )MINIMIZE     INITIALIZE
   BEGIN   not_done?  N max_iter <  AND
   WHILE  REFLECT ( worst point thru geocenter of the rest to get x')
          x' Best_point   better?
          IF  DOUBLE  ( find a point x'' twice as far from geocenter)
              x'' Best_point   better?
              IF store x'' ELSE   store x'  ENDIF
          ELSE
              x'  2nd_Worst_point   better?
              IF     store x'
              ELSE   x' Worst_point   better?
                  IF  store x'  ENDIF
                  HALVE  ( find x'' 0.5 as far from geocenter as REFLECTed pt)
                  x'' Worst_point   better?
                  IF     store x''
                  ELSE   SHRINK  ( uniformly shrink all points toward Best_point)
                  ENDIF
              ENDIF
          ENDIF
   REPEAT ;
```

The simplex algorithm attempts to find a point closer to the minimum than any of the current vertices of the simplex by moving away from the worst vertex (that with the highest value of the function). First a new trial point is found by reflecting the worst vertex through the geometrical center of the other vertices (REFLECT). If that point is better than the best vertex so far, then the algorithm looks twice as far in the same direction (DOUBLE), the better of the two trial points replacing the former worst vertex. On the other hand, if the trial point found after REFLECTion is not better than the best, the algorithm inquires whether it is good enough to keep—that is, is it lower than the second-highest vertex. If so it replaces the worst vertex. What if it is not better than the second-worst vertex? Then the routine tests whether it is better than the worst. If so it replaces the worst, but before ending the outer loop, a new operation is performed: a trial point is found halfway between the old trial point and the geocenter of the others (HALVE). If this new point is an improvement over the worst point, it is stored; otherwise a last, desperate attempt to improve things is made: the lowest vertex is held fixed and the rest of the simplex is shrunk uniformly toward it by some scale factor.

A complex sequence of operations requiring multiple decisions often may be more clearly represented by a *finite state machine* than by a multiply branching binary logic tree. A state machine is the software analog of certain electromechanical devices[12] (such as the device that accepts coins in a vending machine, dispensing goods, making change, and rejecting slugs as necessary). State machines are often represented by graphs, but for programming purposes a tabular representation is clearer.

The simplex algorithm requires us to determine whether the trial vertex is better than the best; between the best and the second-worst; between the second-worst and the worst; or worse than the

12. Zvi Kohavi, *Switching and Finite Automata Theory*, 2nd ed. (McGraw-Hill Publishing Co, New York, 1978).

worst? That is, there are four possibilities that determine four courses of action. These can be represented as in the table below.

state \ input	R < best?	best<R<2worst?	2worst<R<worst?	worst<R?
reflected	store x′ DOUBLE → ×2	store x′ → exit	store x′ HALVE → ½	HALVE → ½
× 2	store x′′ → exit	noop → exit	noop → exit	noop → exit
½	noop → exit	noop → exit	store x′ → exit	SHRINK → exit
exit				

Each cell contains an action (or set of actions) and a state transition to be made after the action is taken. The column labels are inputs that must exhaust all possibilities, and the row labels represent the state the machine is in when presented with the input represented by the column label. The state labelled "exit" is a terminal state, hence its cells contain neither actions nor transitions. The noop ("do nothing") action could, if one wished, be replaced by an error handler that would inform us if—say by hardware failure—the (software) finite state machine has landed in a cell that should be impossible to reach (these cells are indicated by being tinted).

The chief virtue of the state machine representation of complex logic is the impossibility of producing "dead" code that is impossible to reach. Moreover the state transition table shows explicitly which input leads to which action. With such a state machine (called slither below), the)MINIMIZE subroutine becomes

```
: )MINIMIZE      INITIALIZE
   BEGIN   not_done?  N max_iter <  AND
   WHILE  REFLECT   ( worst point thru geocenter of the rest to get x′)
      reflected  >state slither  ( initialize FSM)
      BEGIN     test_x′          ( result is a column number, 0-3)
                slither          ( operate FSM)
                state: slither   ( get current state)
                exit =           ( test for exit state)
      UNTIL
   REPEAT  ;
```

which, when accompanied by the code for slither in tabular format, is much easier to follow than the previous version. Precisely how the state machine is implemented is up to the programmer. For example, a CASE… ENDCASE control structure would suit the bill. Forth is so versatile it has been possible to devise a method of compiling the tabular representation above into a subroutine[13]. This latter method is essentially self-documenting.

Simulated annealing

This method has become important in "solving" certain *NP-complete* programming tasks (this term means that the problem's running time scales like $e^{\lambda N}$ with the size of the problem). Although it might require a very long time to get an exact solution—for example the absolute minimum distance a travelling salesman must go, to visit N cities in a round trip, or the absolute minimum amount of wiring to connect up circuit elements on a printed circuit board—it is possible to come close to the true minimum in a much shorter time. Thus if one can be satisfied with an answer that is within—say—5% of optimum, the computing time can be quite brief.

Simulated annealing takes its name from thermodynamics. A chunk of glass might have many internal cracks and dislocations, and thus be in a state of greater energy than a similar chunk that has no cracks. Typically we heat the glass and slowly cool it to remove the cracks and internal strains.

From a thermodynamic point of view, the probability for the system to be in a state of energy E at temperature $\Theta = kT_{absolute}$ is

$$P \sim e^{-E/\Theta} .$$

The initial (cracked) state of the glass is a local minimum, although not the absolute minimum, of the energy. To get into another state—perhaps of lower energy—the system must pass through a "barrier" or intermediate stae of higher energy. This is very unlikely when the temperature is low. However, at higher temperatures the glass can with reasonable probability pass through many intermediate states of higher energy. Its most likely state is, of course, the lowest-energy one. Thus if the temperature is first raised (and the system allowed to equilibrate) then slowly lowered again, there is a good chance the sysem will be trapped in a state of lower energy, one in which the cracks and internal strains are virtually nonexistent.

In other words, to minimize a function $f\left(\alpha_1, \ldots, \alpha_n\right)$ we compute (randomly) a trial value of the vector $\vec{\alpha} = (\alpha_1, \ldots, \alpha_n)$ and compare $f(\vec{\alpha}_{new})$ with $f(\vec{\alpha}_{old})$. If the new value is smaller, then it replaces the old one. If it is larger, then a random number lying in the interval [0,1] is computed. If it is less than or equal to the transition probability

$$P = \exp\left[-(f_{new} - f_{old})/\Theta\right]$$

the new state is accepted, otherwise it is rejected.

This procedure is repeated many times, and at the same time the "temperature" Θ is gradually reduced. At some point very few transitions are taking place, at which point we decide the process has converged. To some extent it is useful to provide the feedback and interaction of a human programmer, since the best schedule for varying the "temperature" is by no means obvious and it is often a good idea to experiment. One should not feel a great deal of confidence in the result of a

---

13. See, *e.g.*, J.V. Noble, "Avoid Decisions", *Computers in Physics* **5**, #4 (1991) 386; J.V. Noble, *Finite State Machines in Forth* (http://www.jfar.org/article001.html).

single run. It is usually valuable to run the program several times with different initial conditions to see whether it really finds minima close to the optimum.

Simulated annealing, because of its employment of random processes, belongs to the class of numerical techniques called Monte Carlo methods (after the famous gambling casinos of the Principality of Monaco). We discuss other Monte Carlo methods in Chapter 4.