Orthogonal polynomials play an important role in numerical ana ysis and applied mathematics. They satisfy **orthogonality rel tions**[23] of the form

$$\int_A^B dx\, w(x) p_n(x)\, p_m(x) = \delta_{nm} \equiv \begin{cases} 1, & m=n \\ 0, & m \neq n \end{cases} \qquad (40)$$

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

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

Denote the integral in Eq. 40 by $(p_n\,, p_m\,)$ to save having to writ 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 recursio relation

$$p_{n+1}(x) = \left(a_n + x b_n \right) p_n(x) \; + \; c_n\, p_{n-1}(x) \qquad (41)$$

Now apply Eq. 41: assume we have calculated p_n and p_{n-1} an want to calculate p_{n+1}. Clearly, the orthogonality property give

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

and the assumed normalization gives

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

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

---

23. 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) direc- tions in ordinary 3-dimensional space, and so are called **orthogonal** by analogy.

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

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

We express a_n and c_n in terms of b_n to get

$$p_{n+1}(x) = b_n \Big[\Big(x - (p_n , x p_n) \Big) p_n(x) - (p_n , x p_{n-1}) p_{n-1}(x) \Big] \tag{42}$$

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. 42; then we calculate

$$b_n = (\bar{p}_{n+1} , \bar{p}_{n+1})^{-1/2}, \tag{43}$$

multiply the (un-normalized) $\bar{p}_{n+1}$ 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

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

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

$$(x - \alpha_n) p_{n-2} \equiv \sum_{k=0}^{n-1} \mu_k p_k(x) \tag{44}$$

and (by hypothesis) $p_n \perp$ every term of the rhs of Eq. 44, hence it follows (by mathematical induction) that

$$p_{n+1} \perp \{ p_{n-2} , p_{n-3} , \cdots \} .$$

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. 40, we might also define orthogonality in terms of a **sum**, as in Eq. 39a. That is, suppose we define the polynomials such that

$$\sum_{k=0}^{M-1} p_n(x_k)\, p_m(x_k)\, \frac{1}{\sigma_k^2} \equiv \delta_{nm} = \begin{cases} 1, & m=n \\ 0, & m \neq n \end{cases} \qquad (45)$$

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 45 by construction, the coefficients γ_n in our fitting polynomial are simply

$$\gamma_n = \sum_{k=0}^{M-1} p_n(x_k)\, f_k\, \frac{1}{\sigma_k^2}; \qquad (46)$$

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

The algorithm for fitting data with Gram polynomials may be expressed in flow-diagram form:

Read in points f_k, x_k and $w_k \equiv 1/\sigma_k^2$.

DO n = 1 to N-1 (outer loop)
Construct a_n , c_n :

DO k = 0 to M-1 (inner loop)
$p_{n+1}(x_k) = (x_k - a_n)p_n(x_k) - c_n p_{n-1}(x_k)$

$sum = sum + \left(p_{n+1}(x_k)\right)^2 w_k$

$c_{n+1} = c_{n+1} + f_k\, p_{n+1}(x_k)\, w_k$
LOOP (end inner loop)
$c_{n+1} = c_{n+1}\ /sum$

DO k = 0 to M-1 (normalize)
$p_{n+1}(x_k) = p_{n+1}(x_k) / \sqrt{sum}$
LOOP

LOOP

Fig. 8-5 *Construction of Gram polynomials*

The required storage is 5 vectors of length M to hold x_k, $p_n(x_k)$, $p_{n-1}(x_k)$, f_k and $w_k \equiv 1/\sigma_k^2$. We also need to store the coefficients a_n, c_n and the normalizations b_n —that is, 3 vectors of length $N<<M$— in case they should be needed to interpolate. The time involved is approximately 7M multiplications and additions for each n, giving 7NM. Since N can be no greater than M-1 (M data determine at most a polynomial of degree M-1), the maximum possible running time is $7M^2$, which is much less than the time to solve M linear equations.

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 $\chi^2/(degree\ of\ freedom)$[24]: With M data points and an N'th order polynomial, there are M-N-1 degrees of freedom. That is, we evaluate Eq. 38 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:

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

$$\equiv \sum_{k=0}^{M-1} (f_k)^2 - \sum_{n=0}^{N} (\gamma_n)^2 \qquad (47)$$

The first term after ≡ in Eq. 47 is independent of N, and the second term is computed as we go. Thus we could turn the outer loop (over N) into a **BEGIN ... WHILE ... REPEAT** loop, in which N is incremented as long as $\sigma^2_{M,N}$ is larger than $\sigma^2_{M,N+1}$. (Incidentally, Eq. 47 guarantees that as we increase N the fitted curve deviates less and less, on the average, from the measured points. When N = M-1, in fact, the curve goes through the points. But as explained above, this is a meaningless fit, since all data contain measurement errors. A fitted curve that passes closer than σ_k to more than about ⅓ of the points is suspect.)

The code for Gram polynomials is relatively easy to write using the techniques developed in Ch. 5. The program is displayed in full in Appendix 8.4.

---

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

§§3 Simplex algorithm

Sometimes we must fit data by a function that depends on parameters in a **nonlinear** manner. An example is

$$f_k \approx \frac{F}{1 + e^{\alpha (x_k - X)}} \qquad (48)$$

Although the dependence on the parameter F is linear, that on the parameters α and X is decidedly nonlinear.

One way to handle a problem like fitting Eq. 48 might be to transform the data, to make the dependence on the parameters linear. In some cases this is possible, but in 48 no transformation will render linear the dependence on all three parameters at once.

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 **R**. Then we want to find those values of $\{\theta\} \subset \mathbf{R}$ that minimize a positive function:

$$\chi^2 \left(\bar{\theta}_0, \ldots, \bar{\theta}_{N-1}\right) = \min_{\{\theta\} \subset \mathbf{R}} \chi^2 \left(\theta_0, \ldots \theta_{N-1}\right) \qquad (49)$$

One way to accomplish the minimization is *via* calculus, using a method known as **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 θ'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[25].

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* minimum – if a function has several local

---

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

minima, steepest descents will not necessarily find the smallest. 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 a meandering river's channel. Steepest descents 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? A standard way to do this is called the **simplex method**. The idea is to construct a **simplex** – a set of N + 1 distinct and **non-degenerate** vertices in the N-dimensional θ-space ("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.)

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 by the a strategy, expressed as the flow diagram, Fig. 8-8 on page 203 below, that in action somewhat resembles the behavior of an amoeba seeking its food. The key word **)MINIMIZE** that implements the complex decision tree in Fig. 8-8 (given here in pseudocode) is

```
: )MINIMIZE  ( n.iter - - 87: rel.error - - )
    INITIALIZE
    BEGIN  done? NOT  N N.max < AND  .
    WHILE
      REFLECT  r> =best?
      IF  r> =2worst?
        IF  r<worst?  IF STORE.X THEN
            HALVE  r<worst?
            IF STORE.X ELSE SHRINK THEN
        ELSE  STORE.X  THEN
      ELSE DOUBLE    r> =best?
       IF STORE.XP ELSE STORE.X THEN
      THEN
      N 1+  IS N  SORT
    REPEAT  ;
```

used in the format

```
USE( f.name 20 % 1.E-4 )MINIMIZE
```

Fleshing out the details is a —by now— familiar process, so we leave the program *per se* to Appendix 8.5. We also include there a FORTRAN subroutine for the simplex algorithm, taken from

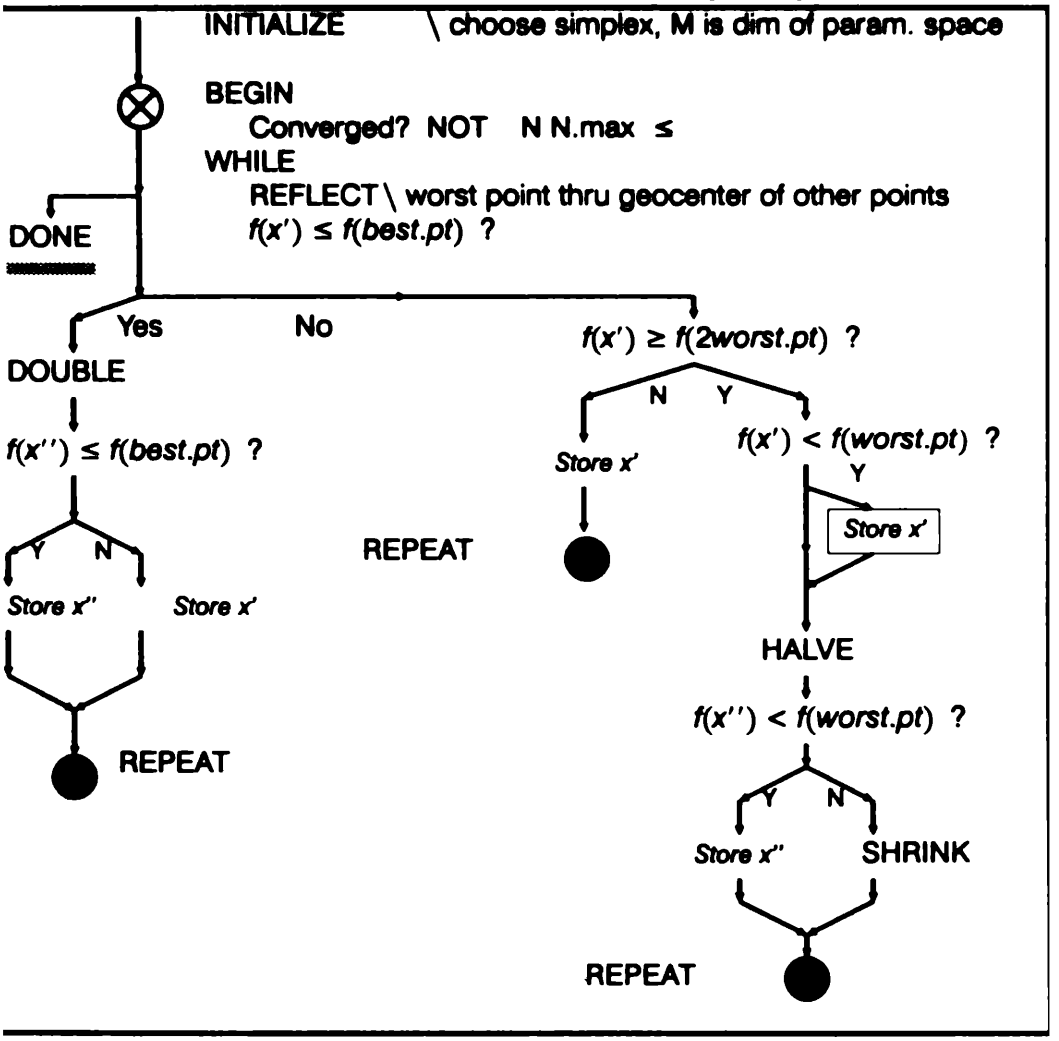

Fig. 8-8 *Flow diagram of the simplex algorithm*

Numerical Recipes [26], as an example of just how indecipherable traditional languages can be.

§3 Appendices

§§1 Gaussian quadrature

Gaussian quadrature formulae are based on the following idea: if we let the points ξ_n and weights w_n be 2N free parameters (n runs from 1 to N), what values of them most accurately represent an integral by the formula

$$I = \int_A^B dx\, \sigma(x) f(x) \approx \sum_{n=1}^{N} w_n f(\xi_n) \quad ? \qquad (50)$$

In Eq. 50 $\sigma(x)$ is a (known) positive function and $f(x)$ is the function we want to integrate. This problem can actually be solved, and leads to tables of points ξ_n and weight coefficients w_n specific to a particular interval [A,B] and weight function $\sigma(x)$. **Gauss-Legendre** integration pertains to $[-1,+1]$ and $\sigma(x)=1$. (Note any interval can be transformed into $[-1,+1]$.)

The interval $[0,\infty)$ and $\sigma(x) = e^{-x}$ leads to **Gauss-Laguerre** formulae, whereas the interval $(-\infty,+\infty)$ and $\sigma(x) = e^{-x^2}$ leads to **Gauss-Hermite** formulae.

Finally, we note that the more common integration formulae such as Simpson's rule or the trapezoidal rule can be derived on the same basis as the Gauss methods, except that the points are specified in advance to be equally spaced and to include the endpoints of the interval. Only the weights w_n can be determined as free fitting parameters that give the best approximation to the integral.

For given N Gaussian formulae can be more accurate than equally-spaced rules, as they have twice as many parameters to play with.

---

26. Press, *et al.*, *Numerical Recipes*, *ibid.*, p. 289ff.

Some FORTH words for 5-point Gauss-Legendre integration:

```
% 0.906179845938664   FCONSTANT  x2
% 0.538469310105683   FCONSTANT  x1
% 0.568888888888889   FCONSTANT  w0
% 0.478628670499366   FCONSTANT  w1
% 0.236926885056189   FCONSTANT  w2

: scale  ( 87: A B - - [A+B]/2 [B-A]/2 )
        FOVER F- F2/ FUNDER F+ ;
: rescale        ( 87: a b x - - a+b*x) F* F+ ;
: F3R+ ( 87: a b c x - - a+x b c) F3R F+ F-ROT ;
```

```
: }Integral       ( 87: A B - - I)
        scale  ( 87: A B - - [A+B]/2 [B-A]/2 )
        FOVER F(X) w0 F* F-ROT  ( 87: - - I a b)
        XDUP  x1 rescale  F(X)  w1 F*  F3R+
        XDUP  x1 FNEGATE rescale
        F(X)  w1 F*  F3R+
        XDUP  x2 rescale  F(X)  w2 F*  F3R+
        XDUP  x2 FNEGATE rescale
        F(X)  w2 F*  F3R+
```

§§2 The trapezoidal rule

Recall (Ch. 8, §1.1) how we approximated the area under a curve by capturing it between rectangles consistently higher- and lower than the curve; and calculating the areas of the two sets of rectangles. In practice we use a better approximation: we average the rectangular upper and lower bounds. The errors tend to cancel, resulting in[27]

$$\frac{w}{2}\sum_{n=0}^{(B-A)/w}\Big(f(A+nw)+f(A+nw+w)\Big) \tag{51}$$

$$\approx \int_A^B dx\, f(x) + \frac{1}{12}\left(\frac{B-A}{w}\right) w^3 \max_{A<x<B}\left|f''(x)\right|$$

Now the error is much smaller[28] – of order w^2 – so if we double the number of points, we decrease the error four-fold. Yet this so-called **trapezoidal rule**[29] requires no more effort (in terms of the number of function evaluations) than the rectangle rule.

---

27. See, *e.g.*, Abramowitz and Stegun, *HMF*, p. 885.
28. $f''(x)$ is the second derivative of $f(x)$, *i.e.* the first derivative of $f'(x)$.
29. Called so because the curve $f(x)$ is approximated by straight line segments between successive points $x, x+w$. Thus we evaluate the areas of trapezoids rather than rectangles.

§§3 Richardson extrapolation

In the program)**INTEGRAL** in Ch.8 §1§§5.3 the word **INTERPOLATE** performs Richardson extrapolation for the trapezoidal rule. The idea is this: If we use a given rule, accurate to order w^n, to calculate the integral on an interval [a,b], then presumably the error of using the formula on each half of the interval and adding the results, will be smaller by 2^{-n}. For the trapezoidal rule, n = 2, hence we expect the error from summing two half-intervals to be 4× smaller than that from the whole interval.

Thus, we can write $\left(I_0 = \int_a^b \ , \ I'_0 = \int_a^{(a+b)/2} \ , \ I_1 = \int_{(a+b)/2}^b \right)$

$$I_0 = I_{exact} + R \tag{52a}$$

$$I'_0 + I_1 = I_{exact} + \frac{R}{4} \tag{52b}$$

Equation 52b is only an approximation because the R that appears in it is not exactly the same as R in Eq. 52a. We will pretend the two R's are equal, however, and eliminate R from the two equations (8.52a,b) ending with an expression for I_{exact}:

$$I_{exact} \approx \frac{4}{3}\left(I'_0 + I_1 - I_0\right) \tag{53}$$

Equation 53 is exactly what appears in **INTERPOLATE**.

§§4 Linear least-squares: Gram polynomials

Here is a FORTH program for implementing the algorithm derived in §2§§2 above.

```
ASK GRAM1

      CODEed words to simplify fstack management
CODE G(N+1) 4 FMUL. FCHS. 2 FLD. 6 FSUB.
    2 FMUL. 1 FADDP.
    DX DS MOV. DS POP.  R64 DS: [BX] FST.
    DS DX MOV. BX POP.  END-CODE
( sag off -- 87: s a b w x g[n] g[n-1] -- s a b w x g[n] g[n+1] )

CODE B(N+1) 2 FXCH. 2 FMUL. 3 FMUL.
    1 FXCH. 1 FMUL. DX DS MOV. DS POP.
    R64 DS: [BX] FADD. DS: [BX] FSTP.
    DS DX MOV. BX POP. END-CODE
( sag off -)
 87: s a b w x g[n] g[n+1] -- s a b w g[n+1] w x g[n+1] )

CODE A(N+1) 1 FMUL. DX DS MOV. DS POP.
    R64 DS: [BX] FADD.
    DS: [BX] FSTP.  DS DX MOV. BX POP. END-CODE
( sag off -- 87: s a b w g[n+1] wxg[n+1] -- s a b w g[n+1] )

CODE C(N+1) 1 FXCH. 2 FMUL*. 2 FMUL.
    2 FXCH. 1 FMULP.
    DX DS MOV. DS POP.
    R64 DS: [BX] FADD. DS: [BX] FSTP.
    DS DX MOV. BX POP. 3 FADDP. END-CODE
( sag off -- 87: s a b w g[n+1] f -- s=s+wg[n+1]**2 a b )

-------------------------Gram polynomial coding
REAL SCALAR DELTA
VAR Nmax
usage: A{ B{ C{ G{{ Nmax }FIT

: FIRST.AB's  F=0 a{ 0 } G!  F=0 b{ 0 } G! ;

: INIT.DELTA  (87: -- g{{ I I }} )  F=0 F=0
    M 0 DO w{ I } G@ y{ I } G@  F**2 FOVER F*
    (87: s s' -- s s' w w*I^2)
    FROT F+  F-ROT F+  FSWAP
    (87: -- s=s+w s'=s'+w*I^2)
  LOOP  DELTA G! FSQRT 1/F ;

: FIRST.G's (87: g{{ 1 I }} -- g{{ 1 I }} )
  M 0 DO F=0 g{{ 0 I }} G! FDUP g{{ 1 I }} G! LOOP ;

: SECOND.AB's  (87: g{{ 1 I }} -- g{{ 1 I }} )
  F=0 b{ 1 0 } G!  FDUP F**2
  F=0 M 0 DO w{ I 0 } G@ x{ I 0 } G@ F* F+ LOOP
  F* a{ 1 0 } G! ;

: FIRST.& SECOND.C's (87: g{{ 1 I }} -- )
  F=0 c{ 0 0 } G! F=0
  M 0 DO w{ I 0 } G@ y{ I 0 } G@ F* F+ LOOP
  F* c{ 1 0 } G! ;

: INITIALIZE  FINIT  IS Nmax  IS g{{  IS c{  IS b{  IS a{
  FIRST.AB's  INIT.DELTA  FIRST.G's  SECOND.AB's
  FIRST.& SECOND.C's ;

0 VAR N  0 VAR N+1
: inc.N  N+1 DUP IS N  1+  IS N+1 ;
: DISPOSE  R DDUP R@ G@  R ;
: inc.OFF  #BYTES DROP + ;
: ZERO.L  DDUP F=0 R64!L ;

: START.Next.G
( -- [c{n+1}] [a{n+1}] [b{n+1}]  87: -- s=0 a{n} b{n} )
    FINIT  F=0  c{ N+1 0 } DROP  ZERO.L
    a{ N 0 } DISPOSE  inc.OFF  ZERO.L
    b{ N 0 } DISPOSE  inc.OFF  ZERO.L ;


: SET.FSTACK    w{ I' 0 } G@  x{ I' 0 } G@  g{{ N I' }} G@
     g{{ N 1- I' }} G@ ;

: }@*I  (adr n -- 87: x -- x)  FDUP 0} DISPOSE F* G! ;

: NORMALIZE  (87: sum -- )
     1/F a{ N+1 }@*I  FSQRT
     b{ N+1 }@*I  c{ N+1 }@*I
     M 0 DO FDUP g{{ N+1 I }} DISPOSE F* G! LOOP
     FDROP ;
CODE 6DUP  OPT" 6 PICK 6 PICK 6 PICK
          6 PICK 6 PICK 6 PICK " END-CODE
CODE 6DROP OPT" DDROP DDROP DDROP " END-CODE
```

```
\ GRAM POLYNOMIAL LEAST-QUARES (CONT'D)

Next.G  START.Next.G
   M 0 DO  6DUP  SET.FSTACK
       g{{ N+1 I }} DROP  G(N+1)  B(N+1)  A(N+1)
       y{ I 0} G@  C(N+1)
   LOOP  6DROP  FDROP FDROP  NORMALIZE ;

: New.DELTA  (:: – old.delta new.delta)
     DELTA G@  FDUP  c{ N 0} G@  F**2  F-  FDUP
      DELTA G! ;

: NOT.ENUF.G's?  New.DELTA  M N+1 -  S-F
     FDUP  F=1 F-
( CR  .FS  ." NEXT ITERATION?" ?YN  0= IF ABORT THEN )
     (:: – d d' m-n-1  m-n-2 )
     FROT  F\  F-ROT  F/  (:: – d'/[m-n-2]  d/[m-n-1] )
     FOVER  F0  IF  F  ELSE  FDROP FDROP 0 THEN ;

: }FIT  ( X{ Y{ S{ Nmax A{ B{ C{ G{{ – )
    INITIALIZE   1 IS N  2 IS N+1
    BEGIN    NOT.ENUF.G's?   N Nmax  AND
    WHILE    Next.G  inc.N
    REPEAT  FINIT ;
\---------------------------------- end of code

: RECONSTRUCT  M 0  DO  CR  x{ I 0} G@ F.  y{ I 0} G@ F.
         F=0  N+1 1+  1 DO
            c{ I 0} G@  g{{ I J }} G@  F* F+
           LOOP  F.
        LOOP  ;
```

§§5 Non-linear least squares: simplex method

A FORTRAN program for the simplex method is given bel
on page 209. The FORTH version, as discussed in §2§§3
given on pages 210 and 211.

```
      SUBROUTINE AMOEBA(P,Y,MP,NP,NDIM,FTOL,FUNK,ITER)
      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
      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
1     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.
2     CONTINUE
      DO 14 I=1,MPTS
        IF(I.NE.IHI)THEN
          DO 13 J=1,NDIM
            PBAR(J)=PBAR(J)+P(I,J)
3         CONTINUE
        ENDIF
4     CONTINUE
      DO 15 J=1,NDIM
        PBAR(J)=PBAR(J)/NDIM
        PR(J)=(1.+ALPHA)*PBAR(J)-ALPHA*P(IHI,J)
5     CONTINUE
      YPR=FUNK(PR)
      IF(YPR.LE.Y(ILO))THEN
        DO 16 J=1,NDIM
          PRR(J)=GAMMA*PR(J)+(1.-GAMMA)*PBAR(J)
6       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
```

```
\ MINIMIZATION BY THE SIMPLEX METHOD
\ VERSION OF 20:51:19 4/30/1991

TASK AMOEBA

\ ---------------------- FUNCTION NOTATION
  VARIABLE <F>
  : USE( [COMPILE] ' CFA <F> ! ;
  : F(X) EXECUTE@ ;
  BEHEAD' <F>
\ ------------------- END FUNCTION NOTATION

\ ------------------------ DATA STRUCTURES
3 VAR Ndim
0 VAR N
0 VAR N.max

CREATE SIMPLEX{{ Ndim 4 (bytes) * Ndim 1+ (# points)
    * ALLOT
CREATE F{ Ndim 1+ 4 (bytes) * ALLOT \ residuals
CREATE index Ndim 1+ 2* ALLOT \ array for scrambled indices

: >index (I - I') 2* index + @ ;
DVARIABLE Residual
DVARIABLE Residual'
DVARIABLE Epsilon
CREATE X{ Ndim 4 (bytes) * ALLOT \ trial point
CREATE XP{ Ndim 4 (bytes) * ALLOT \ 2nd trial point
CREATE Y{ Ndim 4 (bytes) * ALLOT \ geocenter

: } (adr n - adr + 4n) 4* + ; \ part of array notation
: }} (adr m n - adr + [m*Ndim + n]*4) SWAP Ndim * + } ;
\ ---------------- ---- END DATA STRUCTURES
\ ---------------------------- ACTION WORDS
: RESIDUALS
    Ndim 1+ 0 DO SIMPLEX{{ I 0 }} F(X) F{ I } R32!
    LOOP ;
: <index> Ndim 1+ 0 DO I index I 2* + ! LOOP ; \ fill index

: ORDER <index>
    Ndim 1+ 0 DO F{ I >index } R32@
    I 1+ BEGIN Ndim 1+ OVER
    WHILE F{ OVER >index } R32@ FOVER FOVER F>
        IF FSWAP
            I >index OVER >index
            index I 2* + ! index 3 PICK 2* + !
        THEN FDROP 1+
    REPEAT FDROP DROP
    LOOP ;
CENTER ( - ) FINIT Ndim S-F (87: - Ndim)
    Ndim 0 DO F=0 \ loop over components
        Ndim 0 DO \ average over vectors

            SIMPLEX{{ I index J }} R32@ F+
        LOOP FOVER F/
        Y{ I } R32! \ put away
    LOOP FDROP ;
\ note: Worst.Point is SIMPLEX{{ Ndim index J }}
\ -- excluded!

: DONE! CR ." We're finished." ;
: TOO.MANY CR ." Too many iterations." ;

: V.MOVE (src.adr dest.adr - ) \ move vector
    Ndim 0 DO OVER I } OVER I } 2 MOVE
  LOOP DDROP ;
: STORE (adr1 adr2 -- ) 0 }
    SIMPLEX{{ Ndim index 0 }} V.MOVE
    F{ Ndim index } 2 MOVE ;
: STORE.X Residual X{ STORE ;
: STORE.XP Residual' XP{ STORE ;

: New.F X{ F(X) Residual R32! ;
: EXTRUDE (87: scale.factor -- )
\ extend pseudopod
    Ndim 0 DO DUP I } R32@ (87: -- s.f x)
    Y{ I } R32@ FUNDER F- (87: -- s.f y x-y)
    FROT FUNDER F* (87: -- y s.f [x-y]*s.f)
    FROT F+ (87: -- s.f y+[x-y]*s.f)
    X{ I } R32!
    LOOP DROP FDROP New.F ;

: F=1/2 F=1 FNEGATE F=1 FSCALE FPLUCK ;
: F=2 F=1 FDUP FSCALE FPLUCK ;

\ ---------------------- DEBUGGING CODE
0 VAR DBG
: DEBUG-ON -1 IS DBG ;
: DEBUG-OFF 0 IS DBG ;
: .V 3 SPACES Ndim 0 DO DUP I } R32@ F.
    LOOP DROP ;
: .M (--)
    Ndim 1+ 0 DO DBG CR
    IF I . 2 SPACES THEN
    SIMPLEX{{ I index 0 }} .V
    DBG IF F{ I index } R32@ F. THEN
    LOOP CR
  DBG IF CR X{ .V Residual R32@ F. THEN ;

: .F Ndim 1+ 0 DO CR F{ I index } R32@ F.
    LOOP ;
\ ----------------- END DEBUGGING CODE
: REFLECT CENTER
    F=1 FNEGATE \ scale.factor = -1
    SIMPLEX{{ Ndim index 0 }} \ worst pt.
    EXTRUDE \ calculate x, f(x)
    DBG IF CR ." REFLECTING" THEN .M ;
```

```
AMOEBA CONT'D

DOUBLE   Residual Residual' 2 MOVE
   \ save residual
     X{ XP{ V.MOVE              \ save point
     FINIT F=2 FNEGATE (?)   \ scale factor = -2
     SIMPLEX{{ Ndim index 0 }}     \ worst pt.
     EXTRUDE                  \ calculate x, f(x)
     DBG IF CR ." DOUBLING" THEN .M ;

HALVE    FINIT F=1/2              \ scale factor = 0.5
     SIMPLEX{{ Ndim index 0 }}     \ worst pt.
     EXTRUDE                  \ calculate x, f(x)
     DBG IF CR ." HALVING" THEN .M ;
SHRINK   SIMPLEX{{ 0 >index 0 }}       \ best pt.
     Y{ V.MOVE                \ save it
     Ndim 1+ 1 DO             \ by vector
       Ndim 0 DO              \ by component
         SIMPLEX{{ J >index I }} DUP
         R32@ Y{ I } R32@  FUNDER F-
         F=1/2 F* F+   R32!
       LOOP
     LOOP  RESIDUALS  ORDER
     DBG IF CR ." SHRINKING" THEN .M ;
-------------------------- END ACTION WORDS
------------------------------- TEST WORDS
test) FINIT Residual R32@
   F{ SWAP >index } R32@  F> NOT ;
>=best?  0    (test) ;

<worst?  Ndim  (test) NOT ;
>=2worst? Ndim 1- (test)  ;

done?  F{ Ndim index } R32@
   F{ 0  index } R32@
   FOVER FOVER   F- F2*
   F-ROT F+ F/ FABS  Epsilon R32@  F> ;
--------------------------- END TEST WORDS
```

```
: )MINIMIZE  ( n.iter -- 87: error -- )
   IS N.max  Epsilon R32!  0 IS N        \ initialize
   RESIDUALS                    \ compute residuals
   ORDER                        \ locate best, 2worst, worst
   BEGIN                        \ start iteration
     done?  NOT   N N.max  <  AND
   WHILE
     REFLECT  r> =best?
     IF  r> =2worst?
       IF  r<worst?  IF STORE.X THEN
         HALVE  r<worst?
         IF STORE.X ELSE SHRINK THEN
       ELSE  STORE.X  THEN
     ELSE DOUBLE
       r> =best?
       IF STORE.XP ELSE STORE.X THEN
     THEN
     N 1+ IS N   ORDER
   REPEAT  DONE! ;

\------------------------ EXAMPLE FUNCTIONS
: F1     ( adr -- 87: F ) DUP 0 } R32@ FDUP F**2
                DUP 1 } R32@ F**2  F2* F+
                  2 } R32@ F**2  F2* F2* F+
      36 S-F F- F**2 F2/ FSWAP 6 S-F F* F- ;
\ f1 = .5 * (x * x + 2 * y * y + 4 * z * z - 36) ^ 2 - 6 * x

: F2     ( adr -- 87: F ) DUP DUP 0 }
     R32@ FDUP F**2 1 } R32@ F**2
          F2* F+  36 S-F F- F**2 F2/ FSWAP 6 S-F F* F-
F**2 F* ;  DUP 1 } R32@ 0 } R32@ F/ FATAN F2* FCOS
\ f2 = [(x ^ 2 2*y ^ 2 - 36) ^ 2 - 6*x] * cos(2*atan(y/x)) ^ 2
\ Usage:  USE( MYFUNC 10 1.E-3 )MINIMIZE
FLOATS
5. SIMPLEX{{ 0 0 }} R32!      5. SIMPLEX{{ 1 0 }} R32!
-3. SIMPLEX{{ 0 1 }} R32!     3. SIMPLEX{{ 1 1 }} R32!
7. SIMPLEX{{ 0 2 }} R32!      -1.5 SIMPLEX{{ 1 2 }} R32!

-10. SIMPLEX{{ 2 0 }} R32!    5. SIMPLEX{{ 3 0 }} R32!
1. SIMPLEX{{ 2 1 }} R32!      3. SIMPLEX{{ 3 1 }} R32!
3. SIMPLEX{{ 2 2 }} R32!      3. SIMPLEX{{ 3 2 }} R32!
```