```
  c[2] := numc/denc;
  for j := 1 step 1 until jmax do
    q[2, j] := y[j] - c[2];
  for m := 3 step 1 until mmax do begin
    numc := denc := dend := 0.0;
    for j := 1 step 1 until jmax do begin
      numc := numc + w[j] × y[j] × q[m-1, j] ↑ 2;
      denc := denc + w[j] × q[m-1, j] ↑ 2;
      dend := dend + w[j] × q[m-2, j] ↑ 2 end;
    c[m] := numc/denc; d[m] := denc/dend;
    for j := 1 step 1 until jmax do
      q[m, j] := (y[j]-c[m]) × q[m-1, j] - d[m] × q[m-2, j] end;
comment evaluate contribution of each orthogonal polynomial
  to the minimization of the residuals;
for n := 1 step 1 until nmax do begin
  denpa[n] := 0.0;
    for i := 1 step 1 until imax do
    denpa[n] := denpa[n] + u[i] × p[n, i] ↑ 2 end;
  for m := 1 step 1 until mmax do begin
    denqa[m] := 0.0;
    for j := 1 step 1 until jmax do
      denqa[m] := denqa[m] + w[j] × q[m, j] ↑ 2 end;
  for n := 1 step 1 until nmax do begin
    for m := 1 step 1 until mmax do begin
      alph := 0.0;
      for i := 1 step 1 until imax do begin
        for j := 1 step 1 until jmax do
          alph := alph + u[i] × w[j] × z[i, j] × p[n, i] × q[m, j]
        end;
    alpha[n, m] := alph/(denpa[n]×denqa[m]);
    beta[n, m] := alpha[n, m] × alph; end end;
comment application of Gauss' criterion to determine the de-
  gree polynomial which yields the closest fit to the given data.
  Gauss' criterion is, strictly speaking, applicable only to cases
  where the weights u[i] and w[j] are unity;
sumzsq := 0.0;
for i := 1 step 1 until imax do begin
  for j := 1 step 1 until jmax do
    sumzsq := sumzsq + u[i] × w[j] × z[i, j] ↑ 2 end;
s := t := 1;
for n := 1 step 1 until nmax do begin
  betasum := 0.0;
  for m := 1 step 1 until mmax do begin
    for r := 1 step 1 until n do
      betasum := betasum + beta[r, m];
    if betasum > sumzsq then trialgausscrit := 0.0
    else
    trialgausscrit := (sumzsq-betasum)/(imax×jmax-n×m);
    if n = 1 ∧ m = 1 then gausscrit := trialgausscrit;
    if gausscrit = trialgausscrit then begin
      if n × m < s × t then begin
        s := n;
        t := m end end;
    if gausscrit > trialgausscrit then begin
      gausscrit := trialgausscrit;
      s := n;
      t := m end end end;
nmax := s;
mmax := t;
minsqd := (gausscrit×(imax×jmax-nmax×mmax)/(imax×jmax))
  ↑ ½;
comment evaluation of orthogonal polynomial coefficients;
for n := 1 step 1 until nmax do begin
  pc[n, m] := 1.0;
  for s := 1 step 1 until n - 1 do begin
    pc[n, s] := -a[n] × pc[n-1, s];
    if s ≠ 1 then pc[n, s] := pc[n, s] + pc[n-1, s-1];
    if s ≠ n - 1 then pc[n, s] := pc[n, s] - b[n] × pc[n-2, s]
    end end;
for m := 1 step 1 until mmax do begin
  qc[m, m] := 1.0;
  for t := 1 step 1 until m - 1 do begin
    qc[m, t] := -c[m] × qc[m-1, t];
    if t ≠ then qc[m, t] := qc[m, t] + qc[m-1, t-1];
    if t ≠ m - 1 then qc[m, t] := qc[m, t] - d[m] × qc[m-2, t]
    end end;
comment evaluation of approximating polynomial coefficients;
for s := 1 step 1 until nmax do begin
  for t := 1 step 1 until mmax do begin
    phi[s, t] := 0.0;
    for n := s step 1 until nmax do begin
      for m := t step 1 until mmax do
        phi[s, t] := phi[s, t] + alpha[n, m] × pc[n, s] × qc[m, t]
        end end end;
comment evaluation of dependent variables using the approxi-
  mating polynomial;
minsqdcomp := sumdifcomp := maxidifcomp := 0.0;
for i := 1 step 1 until imax do begin
  for j := 1 step 1 until jmax do begin
    zcomp[i, j] := 0.0;
    for s := nmax step - 1 until 1 do begin
      poly := phi[s, mmax];
      for t := mmax - 1 step 1 until 1 do
        poly := poly × y[j] + phi[s, t];
      zcomp[i, j] := zcomp[i, j] × x[i] + poly end;
    rescomp := z[i, j] - zcomp[i, j];
    zcomp[i, j] := zcomp[i, j] + meanz;
    minsqdcomp := minsqdcomp + u[i] × w[j] × rescomp ↑ 2;
    sumdifcomp := sumdifcomp + abs(rescomp);
    if abs (rescomp) > maxdifcomp then
    maxdifcomp := abs(rescomp) end end;
minsqdcomp := (minsqdcomp/(imax × jmax)) ↑ ½;
sumdifcomp := sumdifcomp/(imax × jmax);
end surfacefit
```

ALGORITHM 165
COMPLETE ELLIPTIC INTEGRALS

HENRY C. THACHER, JR.*
Reactor Eng. Div., Argonne National Lab., Argonne, Ill.

* Work supported by the U.S. Atomic Energy Commission.

```
procedure KANDE(ml, K, E, tol, alarm);
value ml, tol;
real ml, K, E, tol;
label alarm;
comment this procedure computes the complete elliptic inte-
  grals K(ml) = ∫₀^{π/2} (1 - (1 - ml) sin²ν)^{-1/2} dν and E(ml) =
  ∫₀^{π/2} (1 - (1 - ml) sin²ν)^{1/2} dν by the arithmetic-geometric-
  mean process. The accuracy is limited only by the accuracy of
  the arithmetic.
```

Except for the provision of tests for pathological values of the parameter, the calculation of K is only a slight modification of the second procedure of Algorithm No. 149 (*Comm. ACM.* **5** (Dec. 1962), 605). These integrals may also be approximated to limited (6D) accuracy by Algorithms 55 and 56 (Comm. ACM. **4** (Apr. 1961), 180). Unless the square-root is exceptionally fast, the latter algorithms are probably more efficient for 6D-accuracy.

The complementary parameter, $m1$, is chosen as the independent variable, rather than the parameter, m, the modulus, k or the modular angle α, because of the possibility of serious loss of significance in generating $m1$ from the other possible independent variables when $m1$ is small and $dK/dm1$ is very large. These variables are related by $m1 = 1 - m = 1 - k^2 = \cos^2\alpha$.

```
    The formal parameter, tol, determines the relative accuracy
  of the result. To prevent entering a nonterminating loop, tol
  should not be less than twice the relative error in the square
  root routine. If m1 ≦ 0 or if m1 > 1, the procedure exits to
  alarm. K(0) = ∞ while E(0) = 1.00000000.
    The body of this procedure has been tested using the Dart-
  mouth SCALP processor for the LGP-30. With tol = 5₁₀ − 7,
  results agreed with tabulated values to within 3 in the seventh
  significant digit;
begin real a, b, c, sum, temp;
  integer fact;
  if m 1 > 1 ∨ m1 ≦ 0 then go to alarm;
  a := fact := 1;
  b := sqrt(m1);
  temp := 1 − m1;
  sum := 0;
iter: sum := sum + temp;
  c := (a − b)/2;
  fact := fact + fact;
  temp := (a + b)/2;
  b := sqrt (a × b);
  a := temp;
  temp := fact × c × c;
  if abs(c) ≧ tol × a ∨ temp > tol × sum then go to iter;
  sum := sum + temp;
  K := 3.141592654/(a + b);
  comment pi must be given to the full accuracy desired;
  E := K × (1 − sum/2)
end
```

ALGORITHM 166
MONTECARLO

R. D. RODMAN
Burroughs Corp., Pasadena, Calif.

```
procedure montecarlo (n, a, row, tol, mxm, inv, test, count);
  value n, row, tol, mxm; integer n, row, mxm, count;
  real tol; real array a, inv, test;
  comment this procedure will compute a single row of the
    inverse of a given matrix using a monte carlo technique.
    n is the size of the matrix, array a is the matrix, row indicates
    which inverse row is to be computed, tol is a tolerance factor
    and thus a criterion for terminating the process, mxm is 1000
    times the maximum number of random walks to be taken,
    after which the process is terminated, array inv contains the
    inverse row, array test contains the innerproduct of inv with
    the rowth column of a, count is the number of random walks
    executed upon termination. real procedure RANDOM must
    be declared in the blockhead of procedure MONTE CARLO
    and generates a single random number between 0 and 1. If
    a is the matrix to be inverted, the absolute value of the largest
    eigenvalue of the matrix I − a (I is the unit matrix) must be
    less than one to assure convergence. This procedure is easily
    adapted to finding a single unknown from a set of simultaneous
    linear equations;
  begin integer i, j, k, nwk, lastwalk, walk; real res, p, g;
  real array sum[1:n], v[1:n, 1:n];
start: p := (n−1)/n × n;
       for i := 1 step 1 until n do for j := 1 step 1 until n do
       v[i,j] := if i ≠ j then −a[i,j]/p else (1−a[i,j])/p;
       nwk := 1000;
       count := res := 0;
       for k := 1 step 1 until n do test [k] := sum [k] := 0;
start1: lastwalk := row; g := 1;
start2: walk := (RANDOM/p) + 1;
       if walk > n then go to stop;
       g := v[lastwalk,walk] × g; lastwalk := walk;
       go to start2;
stop:  count := count + 1; sum[lastwalk] := sum[lastwalk] + g;
       if count < nwk then go to start1;
       for k := 1 step 1 until n do inv[k] := n × sum[k]/count;
       for i := 1 step 1 until n do for k := 1 step 1 until n do
       test[i] := inv[k] × a[k, i] + test[i];
       for i := 1 step 1 until row−1, row+1 step 1 until n do
       res := abs(test[i]) + res; res := abs(test[row]−1) + res;
       if res < tol then go to exit;
       if count ≧ 1000 × mxm then go to exit;
       nwk := nwk + 1000; res := 0;
       for k := 1 step 1 until n do test [k] := 0;
       go to start1;
exit:  end of monte carlo inversion procedure
```

ALGORITHM 167
CALCULATION OF CONFLUENT DIVIDED DIFFERENCES

W. KAHAN AND I. FARKAS
Institute of Computer Science, University of Toronto, Canada

```
real procedure DVDFC(n, X, V, B, W); integer n;
  real array X, V, B, W;
comment DVDFC ca.culates the forward divided difference
  Δf(X₁ , X₂ , ··· Xₙ). n is an integer which takes the values
  n = 1, 2, 3, ··· in turn. X is a real array of dimension at least
  n in which X[i] = Xᵢ for i = 1, 2, ··· , n. The values Xᵢ need
  not be distinct nor in any special order, but once the array X
  is chosen it will fix the interpretation of the arrays B and V.
  If X[1], X[2], ··· , X[n] are in monotonic order, then the effect
  of roundoff upon any nth divided difference is no more than
  would be caused by perturbing each f(X[i]) by n units at most
  in its last significant place. But if the X's are not in mono-
  tonic order, the error can be catastrophic if some of the divided
  differences are relatively large. V is a real array of dimension
  at least n containing the values of the function f(X) and per-
  haps its derivatives at the point Xᵢ. V[i] = f^m(Xᵢ)/m! and
  m = mᵢ for i = 1, 2, 3, ··· , n. mᵢ is the number of times that
  the value of Xᵢ has previously appeared in the array X. B is
  a real array of dimension at least n containing backward divided
  differences. Before a reference to DVDFC is executed one should
  have B[i] = Δf(Xᵢ , Xᵢ₊₁ , ··· , Xₙ₋₁) for i = 1, 2, ··· , n−1.
  After that reference to DVDFC is executed one will find B[i] =
  Δf(Xᵢ , Xᵢ₊₁ , ··· , Xₙ₋₁ , Xₙ) for i = 1, 2, ··· , n−1, n. When
  n = 1 the initial state of B is irrelevant. W is a real array of
  dimension (2 + m̄) at least, where m̄ is the maximum value of
  mᵢ for i = 1, 2, ··· , n. W is used for work space;
    begin real DENOM; integer i, j, NK, NIN;
    if n = 1 then go to L1;
    NK := 1;
    for i := 1 step 1 until n do
    begin
    if X[i] = X[n] then begin NK := NK + 1;
      W[NK] := V[i] end
    end i;
    for i := n step −1 until 2 do
    begin W[1] := B[i − 1]; B[i] := W[2];
    NIN := if n − i + 2 < NK then n − i + 2 else NK;
    for j := NIN step −1 until 2 do
    begin
    DENOM := X[n] − X[i + j − 3];
    if DENOM ≠ 0 then go to L2;
```