Algorithms

J. G. HERRIOT, Editor

ALGORITHM 338
ALGOL PROCEDURES FOR THE FAST FOURIER TRANSFORM [C6]

RICHARD C. SINGLETON*
(Recd. 21 Nov. 1966, 2 Aug. 1967 and 18 July 1968)
Stanford Research Institute, Menlo Park, CA 94025

KEY WORDS AND PHRASES: fast Fourier transform, complex Fourier transform, multivariate Fourier transform, Fourier series, harmonic analysis, spectral analysis, orthogonal polynomials, orthogonal transformation, virtual core memory, permutation

CR CATEGORIES: 3.15, 3.83, 5.12, 5.14

The following procedures are based on the Cooley-Tukey algorithm [1] for computing the finite Fourier transform of a complex data vector; the dimension of the data vector is assumed here to be a power of two. Procedure *COMPLEXTRANSFORM* computes either the complex Fourier transform or its inverse. Procedure *REALTRANSFORM* computes either the Fourier coefficients of a sequence of real data points or evaluates a Fourier series with given cosine and sine coefficients. The number of arithmetic operations for either procedure is proportional to $n \log_2 n$, where n is the number of data points.

Procedures *FFT2*, *REVFFT2*, *REORDER*, and *REALTRAN* are building blocks, and are used in the two complete procedures mentioned above. The fast transform can be computed in a number of different ways, and these building block procedures were written so as to make practical the computing of large transforms on a system with virtual memory. Using a method proposed by Singleton [2], data is accessed in sub-sequences of consecutive array elements, and as much computing as possible is done in one section of the data before moving on to another. Procedure *FFT2* computes the Fourier transform of data in normal order, giving a result in reverse binary order. Procedure *REVFFT2* computes the Fourier transform of data in reverse binary order and leaves the result in normal binary order. Procedure *REORDER* permutes a complex vector from binary to reverse binary order or from reverse binary to binary order; this procedure also permutes real data in preparation for efficient use of the complex Fourier transform. Procedures *FFT2*, *REVFFT2*, and *REORDER* may also be used to compute multivariate Fourier transforms. The procedure *REALTRAN* is used to unscramble and combine the complex transforms of the even and odd numbered elements of a sequence of real data points. This procedure is not restricted to powers of two and can be used whenever the number of data points is even.

REFERENCES:

1. COOLEY, J. W., and TUKEY, J. W. An algorithm for the machine calculation of complex Fourier series. *Math. Comput. 19*, 90, (Apr. 1965), 297–301.
2. SINGLETON, R. C. On computing the fast Fourier transform. *Comm. ACM 10* (Oct. 1967), 647–654;

* This work was supported by Stanford Research Institute out of Research and Development funds.

```
procedure COMPLEXTRANSFORM (A, B, m, inverse);
  value m, inverse;  integer m;
  Boolean inverse;  array A, B;
comment  Computes the Fourier transform of 2^m complex data
  values. The arrays A[0: n−1] and B[0: n−1], where n = 2^m,
  initially contain the real and imaginary components of the data,
  and on exit contain the corresponding Fourier coefficient values.
  If inverse is false, the Fourier transform
```

$$\frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} (a_k + ib_k) \exp(i2\pi jk/n)$$

```
  is computed. The transform followed by the inverse transform
  (or the inverse transform followed by the transform) gives an
  identity transformation. Procedures FFT2 and REORDER are
  used by this procedure and must also be declared;
begin integer n, j;  real p, q;
  n := 2 ↑ m;  p := q := 1.0/sqrt(n);
  if inverse then
  begin
    q := −q;
    for j := n − 1 step −1 until 0 do B[j] := −B[j]
  end;
  FFT2(A, B, n, m, n);  REORDER(A, B, n, m, n, false);
  for j := n − 1 step −1 until 0 do
    begin A[j] := A[j] × p;  B[j] := B[j] × q end
end COMPLEXTRANSFORM;

procedure REALTRANSFORM(A, B, m, inverse);
  value m, inverse;  integer m;
  Boolean inverse; array A, B;
comment  Computes the finite Fourier transform of 2^(m+1) ≧ 4
  real data points. If inverse is false, the arrays A[0: n] and
  B[0: n], where n = 2^m, initially contain the first 2^m real data
  points x_0, x_1, ···, x_(n−1) as A[0], ···, A[n−1] and the remaining
  2^m real data points x_n, x_(n+1), ···, x_(2n−1) as B[0], B[1], ···, B[n−1].
  On completion of the transform the arrays A and B contain
  respectively the Fourier cosine and sine coefficients a_k and b_k,
  computed according to the relations
```

$$a_k = \frac{1}{n} \sum_{j=0}^{2n-1} x_j \cos(\pi jk/n) \quad \text{for} \quad k = 0, 1, \cdots, n,$$

and

$$b_k = \frac{1}{n} \sum_{k=0}^{2n-1} x_j \sin(\pi jk/n) \quad \text{for} \quad k = 0, 1, \cdots, n.$$

```
  If inverse is true, the arrays A and B initially contain n + 1
  cosine coefficients a_0, a_1, ···, a_n and n + 1 sine coefficients
  b_0, b_1, ···, b_n, where b_0 = b_n = 0. The procedure evaluates the
  corresponding time series x_0, x_1, ···, x_(2n−1), where
```

$$x_j = \frac{a_0}{2} + \sum_{k=1}^{n-1} [a_k \cos(\pi jk/n) + b_k \sin(\pi jk/n)] + \frac{a_n}{2} \cos(\pi j),$$

```
  and leaves the first n values as A[0], A[1], ···, A[n−1] and the
  remaining n values as B[0], B[1], ···, B[n−1]. The procedures
  FFT2, REVFFT2, REORDER, and REALTRAN are used by
  this procedure, and must also be declared;
```

```
begin integer n, j;  real p;
  n := 2 ↑ m;
  if inverse then
  begin
    REALTRAN(A, B, n, true);
    for j := n − 1 step −1 until 0 do B[j] := −B[j];
    FFT2(A, B, n, m, n);
    for j := n − 1 step −1 until 0 do
      begin A[j] := 0.5 × A[j];  B[j] := −0.5 × B[j] end;
    REORDER(A, B, n, m, n, true)
  end
  else
  begin
    REORDER(A, B, n, m, n, true);
    REVFFT2(A, B, n, m, 1);  p := 0.5/n;
    for j := n − 1 step −1 until 0 do
      begin A[j] := p × A[j];  B[j] := p × B[j] end;
    REALTRAN(A, B, n, false)
  end
end REALTRANSFORM;
procedure FFT2(A, B, n, m, ks);  value n, m, ks;
  integer n, m, ks;  array A, B;
```

comment Computes the fast Fourier transform for one variable of dimension 2^m in a multivariate transform. n is the number of data points, i.e., $n = n_1 \times n_2 \times \cdots \times n_p$ for a p-variate transform, and $ks = n_k \times n_{k+1} \times \cdots \times n_p$, where $n_k = 2^m$ is the dimension of the current variable. Arrays $A[0 : n-1]$ and $B[0: n-1]$ originally contain the real and imaginary components of the data in normal order. Multivariate data is stored according to the usual convention, e.g., a_{jkl} is in $A[j \times n_2 \times n_3 + k \times n_3 + l]$ for $j = 0, 1, \cdots, n_1 - 1$, $k = 0, 1, \cdots, n_2 - 1$, and $l = 0, 1, \cdots, n_3 - 1$. On exit, the real and imaginary components of the resulting Fourier coefficients for the current variable are in reverse binary order. Continuing the above example, if the "column" variable n_2 is the current one, column

$$k = k_{m-1}2^{m-1} + k_{m-2}2^{m-2} + \cdots + k_1 2 + k_0$$

is permuted to position

$$k_0 2^{m-1} + k_1 2^{m-2} + \cdots + k_{m-2} 2 + k_{m-1} .$$

A separate procedure may be used to permute the results to normal order between transform steps or all at once at the end. If $n = ks = 2^m$, the single-variate transform

$$(x_j + iy_j) = \sum_{k=0}^{n-1} (a_k + ib_k) \exp (i2\pi jk/n)$$

for $j = 0, \cdots, n - 1$ is computed, where $(a + ib)$ represent the initial values and $(x + iy)$ represent the transformed values;

```
begin integer k0, k1, k2, k3, span, j, jj, k, kb, kn, mm, mk;
  real rad, c1, c2, c3, s1, s2, s3, ck, sk, sq;
  real A0, A1, A2, A3, B0, B1, B2, B3;
  integer array C[0: m];
  sq := 0.707106781187;
  sk := 0.382683432366;
  ck := 0.92387953251;
  C[m] := ks;  mm := (m÷2) × 2;  kn := 0;
  for k := m − 1 step −1 until 0 do C[k] := C[k+1] ÷ 2;
  rad := 6.28318530718/(C[0]×ks);  mk := m − 5;
L:  kb := kn;  kn := kn + ks;
  if mm ≠ m then
  begin
    k2 := kn;  k0 := C[mm] + kb;
L2:  k2 := k2 − 1;  k0 := k0 − 1;
    A0 := A[k2];  B0 := B[k2];
    A[k2] := A[k0] − A0;  A[k0] := A[k0] + A0;
    B[k2] := B[k0] − B0;  B[k0] := B[k0] + B0;
    if k0 > kb then go to L2
  end;
  c1 := 1.0;  s1 := 0;
  jj := 0;  k := mm −2;  j := 3;
  if k ≧ 0 then go to L4 else go to L6;
L3:  if C[j] ≦ jj then
  begin
    jj := jj − C[j];  j := j − 1;
    if C[j] ≦ jj then
    begin
      jj := jj − C[j];  j := j − 1;  k := k + 2;
      go to L3
    end
  end;
  jj := C[j] + jj;  j := 3;
L4:  span := C[k];
  if jj ≠ 0 then
  begin
    c2 := jj × span × rad;  c1 := cos(c2);  s1 := sin(c2);
L5:  c2 := c1 ↑ 2 − s1 ↑ 2;  s2 := 2.0 × c1 × s1;
    c3 := c2 × c1 − s2 × s1;  s3 := c2 × s1 + s2 × c1
  end;
  for k0 := kb + span − 1 step −1 until kb do
  begin
    k1 := k0 + span;  k2 := k1 + span;  k3 := k2 + span;
    A0 := A[k0];  B0 := B[k0];
    if s1 = 0 then
    begin
      A1 := A[k1];  B1 := B[k1];
      A2 := A[k2];  B2 := B[k2];
      A3 := A[k3];  B3 := B[k3]
    end
    else
    begin
      A1 := A[k1] × c1 − B[k1] × s1;
      B1 := A[k1] × s1 + B[k1] × c1;
      A2 := A[k2] × c2 − B[k2] × s2;
      B2 := A[k2] × s2 + B[k2] × c2;
      A3 := A[k3] × c3 − B[k3] × s3;
      B3 := A[k3] × s3 + B[k3] × c3
    end;
    A[k0] := A0 + A2 + A1 + A3;  B[k0] := B0 + B2 + B1 + B3;
    A[k1] := A0 + A2 − A1 − A3;  B[k1] := B0 + B2 − B1 − B3;
    A[k2] := A0 − A2 − B1 + B3;  B[k2] := B0 − B2 + A1 − A3;
    A[k3] := A0 − A2 + B1 − B3;  B[k3] := B0 − B2 − A1 + A3
  end;
  if k > 0 then begin k := k − 2;  go to L4 end;
  kb := k3 + span;
  if kb < kn then
  begin
    if j = 0 then begin k := 2;  j := mk;  go to L3 end;
    j := j − 1;  c2 := c1;
    if j = 1 then
      begin c1 := c1 × ck + s1 × sk;  s1 := s1 × ck − c2 × sk end
    else begin c1 := (c1−s1) × sq;  s1 := (c2 + s1) × sq end;
    go to L5
  end;
L6:  if kn < n then go to L
end FFT2;

procedure REVFFT2(A, B, n, m, ks);  value n, m, ks;
  integer n, m, ks;  array A, B;
```

comment Computes the fast Fourier transform for one variable of dimension 2^m in a multivariate transform. n is the number of data points, i.e., $n = n_1 \times n_2 \times \cdots \times n_p$ for a p-variate transform, and $ks = n_{k+1} \times n_{k+2} \times \cdots \times n_p$, where $n_k = 2^m$ is the dimension of the current variable. Arrays $A[0 : n-1]$ and $B[0: n-1]$ originally contain the real and imaginary components of the data with the indices of each variable in reverse binary order, e.g., a_{jkl} is in $A[j' \times n_2 \times n_3 + k' \times n_3 + l']$ for $j = 0, 1, \cdots,$

```
    n1 - 1, k = 0, 1, ··· , n2 - 1, and l = 0, ··· , n3 - 1, where
    j', k', and l' are the bit-reversed values of j, k, and l. On comple-
    tion of the multivariate transform, the real and imaginary com-
    ponents of the resulting Fourier coefficients are in A and B in
    normal order. If n = 2^m and ks = 1, a single-variate transform
    is computed;
begin
  integer k0, k1, k2, k3, k4, span, nn, j, jj, k, kb, nt, kn, mk;
  real rad, c1, c2, c3, s1, s2, s3, ck, sk, sq;
  real A0, A1, A2, A3, B0, B1, B2, B3, re, im;
  integer array C[0: m];
  sq := 0.707106781187;
  sk := 0.382683432366;
  ck := 0.92387953251;
  C[0] := ks;  kn := 0;  k4 := 4 × ks;  mk := m - 4;
  for k := 1 step 1 until m do C[k] := ks := ks + ks·
  rad := 3.14159265359/(C[0]×ks);
L:  kb := kn + k4;  kn := kn + ks;
  if m = 1 then go to L5;
  k := jj := 0;  j := mk;  nt := 3;
  c1 := 1.0;  s1 := 0;
L2:  span := C[k];
  if jj ≠ 0 then
  begin
    c2 := jj × span × rad;  c1 := cos(c2);  s1 := sin(c2);
L3:  c2 := c1 ↑ 2 - s1 ↑ 2;  s2 := 2.0 × c1 × s1;
    c3 := c2 × c1 - s2 × s1;  s3 := c2 × s1 + s2 × c1
  end else s1 := 0;
  k3 := kb - span;
L4:  k2 := k3 - span;  k1 := k2 - span;  k0 := k1 - span;
  A0 := A[k0];  B0 := B[k0];
  A1 := A[k1];  B1 := B[k1];
  A2 := A[k2];  B2 := B[k2];
  A3 := A[k3];  B3 := B[k3];
  A[k0] := A0 + A1 + A2 + A3;  B[k0] := B0 + B1 + B2 + B3;
  if s1 = 0 then
  begin
    A[k1] := A0 - A1 - B2 + B3;  B[k1] := B0 - B1 + A2 - A3;
    A[k2] := A0 + A1 - A2 - A3;  B[k2] := B0 + B1 - B2 - B3;
    A[k3] := A0 - A1 + B2 - B3;  B[k3] := B0 - B1 - A2 + A3
  end
  else
  begin
    re := A0 - A1 - B2 + B3;  im := B0 - B1 + A2 - A3;
    A[k1] := re × c1 - im × s1;  B[k1] := re × s1 + im × c1;
    re := A0 + A1 - A2 - A3;  im := B0 + B1 - B2 - B3;
    A[k2] := re × c2 - im × s2;  B[k2] := re × s2 + im × c2;
    re := A0 - A1 + B2 - B3;  im := B0 - B1 - A2 + A3;
    A[k3] := re × c3 - im × s3;  B[k3] := re × s3 + im × c3
  end;
  k3 := k3 + 1;  if k3 < kb then go to L4;
  nt := nt - 1;
  if nt ≧ 0 then
  begin
    c2 := c1;
    if nt = 1 then
      begin c1 := c1 × ck + s1 × sk;  s1 := s1 × ck - c2 × sk end
    else begin c1 := (c1-s1) × sq;  s1 := (c2+s1) × sq end;
    kb := kb + k4;  if kb ≦ kn then go to L3 else go to L5
  end;
  if nt = -1 then begin k := 2;  go to L2 end;
  if C[j] ≦ jj then
  begin
    jj := jj - C[j];  j := j - 1;
    if C[j] ≦ jj then
      begin jj := jj - C[j];  j := j - 1;  k := k + 2 end
    else begin jj := C[j] + jj;  j := mk end
  end
  else begin jj := C[j] + jj;  j := mk end;
```

```
  if j < mk then go to L2;  k := 0;  nt := 3;
  kb := kb + k4;  if kb ≦ kn then go to L2;
L5:  k := (m÷2) × 2;
  if k ≠ m then
  begin
    k2 := kn;  k0 := j := kn - C[k];
L6:  k2 := k2 - 1;  k0 := k0 - 1;
    A0 := A[k2];  B0 := B[k2];
    A[k2] := A[k0] - A0;  A[k0] := A[k0] + A0;
    B[k2] := B[k0] - B0;  B[k0] := B[k0] + B0;
    if k2 > j then go to L6
  end;
  if kn < n then go to L
end REVFFT2;

procedure REORDER(A, B, n, m, ks, reel);
  value n, m, ks, reel;  integer n, m, ks;
  Boolean reel; array A, B;
comment Permutes data from normal to reverse binary order
  or from reverse binary to normal order. If reel is false, data for
  one variate of dimension 2^m in a multivariate data set of size n
  is permuted. In a p-variate transform with n = n1 × n2 ×
  ··· × np , ks has the value ks = nk × nk+1 × ··· × np , where
  nk = 2^m is the dimension of the current variable. For a single-
  variate transform, n = ks = 2^m. If reel is true, A[2×j+1] and
  B[2×j] are exchanged for j = 0, 1, ···, (n-2)/2, then adjacent
  pairs of entries in A and B are permuted to reverse-binary order.
  This option is used when transforming 2n real data values, with
  the first n stored in A and the second n in B. After permutation,
  the even-numbered entries are in A and the odd-numberd entries
  are in B, each in reverse-binary order.
    Calling REORDER twice with the same parameter values gives
  an identity transformation;
begin integer i, j, jj, k, kk, kb, k2, ku, lim, p;
  real t;
  integer array C, LST[0: m];
  C[m] := ks;
  for k := m step -1 until 1 do C[k - 1] := C[k] ÷ 2;
  p := j := m - 1;  i := kb := 0;
  if reel then
  begin
    ku := n - 2;
    for k := 0 step 2 until ku do
      begin t := A[k + 1];  A[k + 1] := B[k];  B[k] := t end
  end else m := m - 1;
  lim := (m + 2) ÷ 2;  if p ≦ 0 then go to L4;
L:  ku := k2 := C[j] + kb;  jj := C[m - j];  kk := kb + jj;
L2:  k := kk + jj;
L3:  t := A[kk];  A[kk] := A[k2];  A[k2] := t;
  t := B[kk];  B[kk] := B[k2];  B[k2] := t;
  kk := kk + 1;  k2 := k2 + 1;
  if kk < k then go to L3;
  kk := kk + jj;  k2 := k2 + jj;
  if kk < ku then go to L2;
  if j > lim then
  begin
    j := j - 1;  i := i + 1;
    LST[i] := j;  go to L
  end;
  kb := k2;
  if i > 0 then
    begin j := LST[i];  i := i - 1;  go to L end;
  if kb < n then begin j := p; go to L end;
L4:
end REORDER;

procedure REALTRAN(A, B, n, evaluate);
  value n, evaluate; integer n;
  Boolean evaluate; array A, B;
```

comment If *evaluate* is **false**, this procedure unscrambles the single-variate complex transform of the n even-numbered and n-odd-numbered elements of a real sequence of length $2n$, where the even-numbered elements were originally in A and the odd-numbered elements in B. Then it combines the two real transforms to give the Fourier cosine coefficients $A[0], A[1], \cdots, A[n]$ and sine coefficients $B[0], B[1], \cdots, B[n]$ for the full sequence of $2n$ elements. If *evaluate* is **true**, the process is reversed, and a set of Fourier cosine and sine coefficients is made ready for evaluation of the corresponding Fourier series by means of the inverse complex transform. Going in either direction, *REALTRAN* scales by a factor of two, which should be taken into account in determining the appropriate overall scaling;

```
begin integer k, nk, nh;
  real aa, ab, ba, bb, re, im, ck, sk, dc, ds, r;
  nh := n ÷ 2;  r := 3.14159265359/n;
  ds := sin(r);  r := −(2×sin(0.5×r)) ↑ 2;
  dc := −0.5 × r;  ck := 1.0;  sk := 0;
  if evaluate then
    begin ck := −1.0;  dc := −dc end
  else begin A[n] := A[0];  B[n] := B[0] end;
  for k := 0 step 1 until nh do
  begin
    nk := n − k;
    aa := A[k] + A[nk];  ab := A[k] − A[nk];
    ba := B[k] + B[nk];  bb := B[k] − B[nk];
    re := ck × ba + sk × ab;  im := sk × ba − ck × ab;
    B[nk] := im − bb;  B[k] := im + bb;
    A[nk] := aa − re;  A[k] := aa + re;
    dc := r × ck + dc;  ck := ck + dc;
    ds := r × sk + ds;  sk := sk + ds
  end
end REALTRAN
```

ALGORITHM 339
AN ALGOL PROCEDURE FOR THE FAST FOURIER TRANSFORM WITH ARBITRARY FACTORS [C6]

RICHARD C. SINGLETON*

(Recd. 2 Dec. 1966, 19 July 1967, 2 Aug. 1967 and 18 July 1968)

Stanford Research Institute, Menlo Park, CA 94025

KEY WORDS AND PHRASES: fast Fourier transform, complex Fourier transform, multivariate Fourier transform, Fourier series, harmonic analysis, spectral analysis, orthogonal polynomials, orthogonal transformation, virtual core memory, permutation
CR CATEGORIES: 3.15, 3.83, 5.12, 5.14

```
procedure FFT(A, B, n, nv, ks);  value n, nv, ks;
  integer n, nv, ks;  array A, B;
```

comment This procedure computes the finite Fourier transform for one variate of dimension nv within a multivariate transform of n complex data values. The real and imaginary components of the data are stored in arrays $A[0{:}n-1]$ and $B[0{:}n-1]$, following the usual arrangement for indexing multivariate data in a single-dimensional array, e.g., a_{jkl} is stored in location $A[j\times n_2\times n_3+k\times n_3+l]$ for $j = 0, 1, \cdots, n_1 - 1$, $k = 0, 1, \cdots, n_2 - 1$, and $l = 0, 1, \cdots, n_3 - 1$. The value of ks for the kth variate of a p-variate transform is

$$ks = n_k \times n_{k+1} \times \cdots \times n_p$$

where $nv = n_k$ and $n = n_1 \times n_2 \times \cdots \times n_p$. On completion of the transform, the real and imaginary components of the resulting Fourier coefficients are in A and B respectively. For a single variable, $n = nv = ks$, and the transform

$$\sum_{k=0}^{n-1} (a_k + ib_k) \exp(i2\pi jk/n)$$

is computed for $j = 0, 1, \cdots, n - 1$.

For a single-variate transform of $2n$ real-valued points, the amount of computing can be reduced by approximately one-half by using procedure *REALTRAN* [3] together with *FFT*. The even-numbered data points are stored initially in array A, the odd-numbered data points in array B, the transform is computed with

$$FFT(A, B, n, n, n),$$

and the result is unscrambled with

$$REALTRAN(A, B, n, \textbf{false})$$

and then scaled by $1/2n$ to give the cosine coefficients as $A[0], A[1], \cdots, A[n]$ and the sine coefficients as $B[1], B[2], \cdots, B[n-1]$, with $B[0] = B[n] = 0$. The inverse operation, evaluating the Fourier series with cosine coefficients A and sine coefficients B, is computed by

$$REALTRAN(A, B, n, \textbf{true})$$

followed by

$$FFT(A, B, n, n, n),$$

then scaling by 1/2, yielding the even-numbered time domain values in array A and the odd-numbered values in array B. Note that the upper bounds of array A and B must be increased to n when procedure *REALTRAN* is used.

The method is based on an algorithm due to Cooley and Tukey [1], with modifications proposed by Singleton [2], to allow computing of large transforms on a system with virtual memory. The dimension nv is first decomposed into its prime factors $nv_1, nv_2, \cdots, nv_m$, and then nv/nv_i transforms of dimension nv_i are computed for $i = 1, 2, \cdots, m$. The resulting transformed values are then permuted to normal order in a final step. Computing times, to a first approximation, should be proportional to $n(nv_1+nv_2+\cdots+nv_m)$. The dimension of array *FACTOR* must be increased if nv has more than 20 factors.

In factoring nv at the beginning of the procedure, factors that are squares of primes are first removed, then the square-free portion is factored. The two factors of each square are placed symmetrically about the square-free factors. For example, $nv = 72$ is factored as $2 \times 3 \times 2 \times 3 \times 2$. This arrangement is used to simplify the final reordering in place. One symmetric permutation step is done for each square factor, and the reordering is completed by following the permutation cycles of the square-free portion.

In the transform phase of the procedure, special coding for factors of 2 and 3 is included for efficiency. Adjacent factors of 2 are also paired, and the results stored as for factors of 2 rather than 4. The remaining factors are handled by an odd-factor routine, using trigonometric function symmetries and smaller

* This research was supported by Stanford Research Institute out of Research and Development funds.

real transforms to reduce the number of multiplications by one-half as compared with a straightforward complex transform of an odd factor. The approximate number of complex multiplications is $n/2$ for a factor of 2, $3n/4$ for a factor of 4, and $(p-1)(p+3)n/4p$ for an odd factor p.

In both the transform and reordering phases, data is accessed in subsequences of consecutive array elements, and as much computing as possible is done in one section of the data before moving on to another. This is done to reduce the number of memory overlay operations in a system with virtual memory. After the first transform or symmetric permutation step, the remaining steps can be performed independently on each of nv_1 spans of data. We complete all remaining steps on the first span before beginning with the second. Similarly, after the second step the first span is subdivided in nv_2 independent spans. This subdivision process is continued through the remaining steps.

A number of working storage arrays are declared within this procedure. For large n, the total working storage is small in comparison with the $2n$ locations for data arrays A and B, except in a couple of cases. In the transform phase, approximately $6q$ working storage locations are used, where q is the largest prime factor in the transform. This requirement is minor except in a single-variate transform with n a prime number. During the reordering phase, the worst case occurs when doing a single-variate transform with n a product of two or more primes with no square factors. In this case, approximately n working storage locations are required.

This program was tested on the Burroughs B5500 computer and compared with another program computing a single n-by-n complex Fourier transform. Whenever n had two or more prime factors, procedure *FFT* was much faster. The B5500 ALGOL system limits single-dimension arrays to 1023 words, but larger transforms can be computed by declaring

array A, $B[0: (n-1) \div 512, 0: 511]$,

storing the data 512 entries per row, and using partial word indexing $A[J.[30:9], J.[39:9]]$ instead of $A[J]$ wherever A and B appear in procedure *FFT*.

References:

1. Cooley, J. W., and Tukey, J. W. An algorithm for the machine calculation of complex Fourier series. *Math. Comput. 19*, 90 (Apr. 1965), 297–301.
2. Singleton, R. C. On computing the fast Fourier transform. *Comm. ACM 10* (Oct. 1967), 647–654.
3. Singleton, R. C. Algorithm 338: ALGOL procedures for the fast Fourier transform. *Comm. ACM 11* (Nov. 1968), 771–774;


```
begin integer array FACTOR[0: 20];  Boolean zero;
  real A0, A1, A2, A3, B0, B1, B2, B3, cm, sm,
    c1, c2, c3, s1, s2, s3, c30, rad;
  integer k0, k1, k2, k3, jk, kf, kh, jf, mm,
    i, j, jj, k, kb, m, span, kt, kn;
  comment  Determine the square factors of nv;
  k := nv;  m := 0;  j := 2;  jj := 4;  jf := 0;
  FACTOR [0] : =1;
L:  for i := k ÷ jj while i × jj = k do
    begin m := m + 1;  FACTOR[m] := j;  k := i end;
  if j = 2 then j := 3 else j := j + 2;
  jj := j × j;  if jj ≦ k then go to L;  kt := m;
  comment  Determine the remaining factors of nv;
  for j := 2, 3 step 2 until k do
  for i := k ÷ j while i × j = k do
    begin m := m + 1;  FACTOR[m] := j;  k := i end;
  if FACTOR[kt] > FACTOR[m] then k := FACTOR[kt]
  else k := FACTOR[m];
  for j := kt step −1 until 1 do
    begin m := m+1;  FACTOR[m] := FACTOR[j] end;
  begin integer array C,D[0: m];
  begin array CK, SK, CF, SF[0:k−1];
    array AP, BP, AM, BM[0:(k−1)÷2];
    array RD, CC, SS[0:m];
    Boolean array BB[0:m+1];
    rad := 6.28318530718;  c30 := 0.866025403784;
    for j := m step −1 until 2 do
    begin
      BB[j] := (FACTOR[j−1]+FACTOR[j]) = 4;
      if BB [j] then
        begin j := j − 1;  BB[j] := false end
    end;
    BB[m+1] := BB[1] := false;
    C[0] := ks ÷ nv;  kn := 0;  D[0] := ks;
    for j := 1 step 1 until m do
    begin
      k := FACTOR[j];  C[j] := C[j−1] × k;
      D[j] := D[j−1] ÷ k;  RD[j] := rad/C[j];
      c1 := rad/k;
      if k > 2 then
        begin CC[j] := cos(c1);  SS[j] := sin(c1) end
    end;
    mm := if BB[m] then m−1 else m;
    if mm > 1 then
    begin
      sm := C[mm−2] × RD[m];
      cm := cos(sm);  sm := sin(sm)
    end;
L1:  kb := kn;  kn := kn + ks;  jj := 0;  i := 1;
    c1 := 1.0;  s1 := 0;  zero := true;
L2:  if BB[i+1] then
      begin kf := 4;  i := i + 1 end
    else kf := FACTOR[i];
    span := D[i];
    if ¬ zero then
    begin
      s1 := jj × RD[i];  c1 := cos(s1);  s1 := sin(s1)
    end;
    comment  Factors of 2, 3, and 4 are handled
      separately to gain efficiency;
L3:  if kf = 4 then
    begin
      if ¬ zero then
      begin
        c2 := c1 ↑ 2 − s1 ↑ 2;  s2 := 2.0 × c1  × s1;
        c3 := c2 × c1 − s2 × s1;  s3 := c2 × s1 + s2 × c1
      end;
      for k0 := kb + span −1 step −1 until kb do
      begin
        k1 := k0 + span;  k2 := k1 + span;  k3 := k2 + span;
        A0 := A[k0];  B0 := B[k0];
        if zero then
        begin
          A1 := A[k1];  B1 := B[k1];
          A2 := A[k2];  B2 := B[k2];
          A3 := A[k3];  B3 := B[k3]
        end
        else
        begin
          A1 := A[k1] × c1 − B[k1] × s1;
          B1 := A[k1] × s1 + B[k1] × c1;
          A2 := A[k2] × c2 − B[k2] × s2;
          B2 := A[k2] × s2 + B[k2] × c2;
          A3 := A[k3] × c3 − B[k3] × s3;
          B3 := A[k3] × s3 + B[k3] × c3
        end;
        A[k0] := A0 + A2 + A1 + A3;  B[k0] := B0 + B2 +
          B1 + B3;
        A[k1] := A0 + A2 − A1 − A3;  B[k1] := B0 + B2 −
          B1 − B3;
```

```
          A[k2] := A0 − A2 − B1 + B3;  B[k2] := B0 − B2 +
            A1 − A3;
          A[k3] := A0 − A2 + B1 − B3;  B[k3] := B0 − B2 −
            A1 + A3
        end
      end
      else if kf = 3 then
      begin
        if ¬ zero then
          begin c2 := c1 ↑ 2 − s1 ↑ 2;  s2 := 2.0 × c1 × s1 end;
        for k0 := kb + span − 1 step −1 until kb do
        begin
          k1 := k0 + span;  k2 := k1 + span;
          A0 := A[k0];  B0 := B[k0];
          if zero then
          begin
            A1 := A[k1];  B1 := B[k1];
            A2 := A[k2];  B2 := B[k2]
          end
          else
          begin
            A1 := A[k1] × c1 − B[k1] × s1;
            B1 := A[k1] × s1 + B[k1] × c1;
            A2 := A[k2] × c2 − B[k2] × s2;
            B2 := A[k2] × s2 + B[k2] × c2
          end;
          A[k0] := A0 + A1 + A2;  B[k0] := B0 + B1 + B2;
          A0 := − 0.5 × (A1+A2) + A0;  A1 := (A1−A2) ×
            c30;
          B0 := − 0.5 × (B1+B2) + B0;  B1 := (B1−B2) ×
            c30;
          A[k1] := A0 − B1;  B[k1] := B0 + A1;
          A[k2] := A0 + B1;  B[k2] := B0 − A1
        end
      end
      else if kf = 2 then
      begin
        k0 := kb + span;  k2 := k0 + span;
        if zero then
        begin
          for k0 := k0 − 1 while k0 ≧ kb do
          begin
            k2 := k2 − 1;  A0 := A[k2];  B0 := B[k2];
            A[k2] := A[k0] − A0;  A[k0] := A[k0] + A0;
            B[k2] := B[k0] − B0;  B[k0] := B[k0] + B0
          end
        end
        else
        for k0 := k0 − 1 while k0 ≧ kb do
        begin
          k2 := k2 − 1;
          A0 := A[k2] × c1 − B[k2] × s1;
          B0 := A[k2] × s1 + B[k2] × c1;
          A[k2] := A[k0] − A0;  A[k0] := A[k0] + A0;
          B[k2] := B[k0] − B0;  B[k0] := B[k0] + B0
        end
      end
      else
      begin
        jk := kf − 1;  kh := jk ÷ 2;  k3 := D [i−1];
        k0 := kb + span;
        if ¬ zero then
        begin
          k := jk − 1;  CF[1] := c1;  SF[1] := s1;
          for j := 1 step 1 until k do
          begin
            CF[j+1] := CF[j] × c1 − SF[j] × s1;
            SF[j+1] := CF[j] × s1 + SF[j] × c1
          end
        end;
        if kf ≠ jf then
        begin
          CK[jk] := CK[1] := c2 := CC[i];
          SK[1] := s2 := SS[i];  SK[jk] := −s2;
          for j := 1 step 1 until kh do
          begin
            k := jk − j;
            CK[k] := CK[j+1] := CK[j] × c2 − SK[j] × s2;
            SK[j+1] := CK[j] × s2 + SK[j] × c2;
            SK[k] := −SK[j+1]
          end
        end;
L4:     k1 := k0 := k0 − 1;  k2 := k0 + k3;
        A3 := A0 := A[k0];  B3 := B0 := B[k0];
        for j := 1 step 1 until kh do
        begin
          k1 := k1 + span;  k2 := k2 − span;
          if zero then
          begin
            A1 := A[k1];  B1 := B[k1];
            A2 := A[k2];  B2 := B[k2]
          end
          else
          begin
            k := kf − j;
            A1 := A[k1] × CF[j] − B[k1] × SF[j];
            B1 := A[k1] × SF[j] + B[k1] × CF[j];
            A2 := A[k2] × CF[k] − B[k2] × SF[k];
            B2 := A[k2] × SF[k] + B[k2] × CF[k]
          end;
          AP[j] := A1 + A2;  AM[j] := A1 − A2;
          BP[j] := B1 + B2;  BM[j] := B1 − B2;
          A3 := AP[j] + A3;  B3 := BP[j] + B3
        end;
        A[k0] := A3;  B[k0] := B3;
        k1 := k0;  k2 := k0 + k3;
        for j := 1 step 1 until kh do
        begin
          k1 := k1 + span;  k2 := k2 − span;  jk := j;
          A1 := A0;  B1 := B0;  A2 := B2 := 0;
          for k := 1 step 1 until kh do
          begin
            A1 := AP[k] × CK[jk] + A1;
            A2 := AM[k] × SK[jk] + A2;
            B1 := BP[k] × CK[jk] + B1;
            B2 := BM[k] × SK[jk] + B2;
            jk := jk + j;  if jk ≧ kf then jk := jk − kf
          end;
          A[k1] := A1 − B2;  A[k2] := A1 + B2;
          B[k1] := B1 + A2;  B[k2] := B1 − A2
        end;
        if k0 > kb then go to L4;  jf := kf
      end;
      if i < mm then
        begin i := i + 1;  go to L2 end;
      i := mm;  zero := false;
      kb := D[i − 1] + kb;
      if kb < kn then
      begin
        for jj := C[i−2] + jj while jj ≧ C[i−1] do
          begin i := i − 1;  jj := jj − C[i] end;
        if i = mm then
        begin
          c2 := c1;  c1 := cm × c1 − sm × s1;
          s1 := sm × c2 + cm × s1;  go to L3
        end;
```

```
      if BB[i] then i := i + 1;  go to L2
    end;
    if kn < n then go to L1
  end;
  i := 1;
  for j := kt - 1 step -1 until 1 do
  begin
    FACTOR[j] := FACTOR[j] - 1;  i := FACTOR[j] + i
  end;
  comment We now permute the result to normal order;
  comment The following if statement does the complete re-
    ordering if the square-free portion of n has at most one
    prime factor. Otherwise it does a partial reordering, leaving
    each entry in its correct section of length n ÷ c[kt],
    where c[kt] ↑ 2 is the product of the square factors;
  if kt > 0 then
  begin integer array S[0:i];
    j := 1;  i := kb := 0;
L5:  k3 := k2 := D[j] + kb;  jk := jj := C[j-1];
    k0 := kb + jj;  span := C[j] - jj;
L6:  k := k0 + jj;
L7:  A0 := A[k0];  A[k0] := A[k2];  A[k2] := A0;
    B0 := B[k0];  B[k0] := B[k2];  B[k2] := B0;
    k0 := k0 + 1;  k2 := k2 + 1;
    if k0 < k then go to L7;
    k0 := k0 + span;  k2 := k2 + span;
    if k0 < k3 then go to L6;
    if k0 < (k3+span) then
      begin k0 := k0 - D[j] + jj;  go to L6 end;
    k3 := D[j] + k3;
    if (k3-kb) < D[j-1] then
    begin
      k2 := k3 + jk;  jk := jk + jj;
      k0 := k3 - D[j] + jk;  go to L6
    end;
    if j < kt then
    begin
      k := FACTOR[j] + i;  j := j + 1;
L8:    i := i + 1;  S[i] := j;  if i < k then go to L8;
      go to L5
    end;
    kb := k3;
    if i > 0 then
      begin j := S[i];  i := i - 1;  go to L5 end;
    if kb < n then begin j := 1;  go to L5 end
  end;
  jk := C[kt];  span := D[kt];  m := m - kt;
  kb := span ÷ jk -2;
  comment The following if statement completes the reorder-
    ing if the square-free portion of n has two or more prime
    factors;
  if kt < m - 1 then
  begin integer array R[0:kb];
    array TA, TB[0:jk-1];
    for j := kt step 1 until m do D[j] := D[j] ÷ jk;
    jj := 0;
    for j := 1 step 1 until kb do
      begin
      k := kt;
      for jj := D[k+1] + jj while jj ≧ D[k] do
        begin jj := jj - D[k];  k := k + 1 end;
      if jj = j then R[j] := - j else R[j] := jj
    end;
    comment Determine the permutation cycles of length
      ≧2;
    for j := 1 step 1 until kb do if R[j] > 0 then
    begin
      k2 := j;
      for k2 := abs(R[k2]) while k2 ≠ j do R[k2] := -R[k2]
    end;
    comment Reorder A and B following the permutation
      cycles;
    kn := i := j := 0;
LA:  kb := kn;  kn := kn + ks;
LB:  j := j + 1;  if R[j] < 0 then go to LB;
    k := R[j];  k0 := jk × k + kb;
LC:  TA[i] := A[k0+i];  TB[i] := B[k0+i];
    i := i + 1;  if i < jk then go to LC;  i := 0;
LD:  k := -R[k];  jj := k0;  k0 := jk × k + kb;
LE:  A[jj+i] := A[k0+i];  B[jj+i] := B[k0+i];
    i := i + 1;  if i < jk then go to LE;  i := 0;
    if k ≠ j then go to LD;
LF:  A[k0+i] := TA[i];  B[k0+i] := TB[i];
    i := i + 1;  if i < jk then go to LF;  i := 0;
    if j < k2 then go to LB;  j := 0;
    kb := kb + span;  if kb < kn then go to LB;
    if kn < n then go to LA
  end
 end
end FFT
```

ALGORITHM 340
ROOTS OF POLYNOMIALS BY A ROOT-SQUARING AND RESULTANT ROUTINE [C2]

ALBERT NOLTEMEIER

(Recd. 2 Nov. 1967, 25 Jan. 1968 and 16 July 1968)

Technische Universität Hannover, Rechenzentrum, Hannover, Germany

KEY WORDS AND PHRASES: rootfinders, roots of polynomial equations, polynomial zeros, root-squaring operations, Graeffe method, resultant procedure, subresultant procedure, testing of roots, acceptance criteria

CR CATEGORIES: 5.15

```
procedure AG4(n, c, mm, delta, epsilon, range) Result: (re, im,
  mu, rt, gc, m, i, t) Exit: (fail);
  value n, mm, delta, epsilon, range;
  integer n, m, i, mm;  real delta, epsilon, range;
  integer array mu;
  array c, re, im, rt, gc, t;
  label fail;
comment AG4 finds simultaneously zeros of a polynomial of
  degree n with real coefficients by a root-squaring and resultant
  routine.
    This procedure supersedes Algorithm 59 [2]. The following
  changes were made:
  (a) In the procedure heading, the meaning of the old formal
    parameter alpha is shared by the three new parameters mm,
    delta, and epsilon, and range, m, i, t, fail are added to the formal
    parameter list.
```

(b) In the beginning of the procedure body the polynomial is tested for 0 as a zero (label *ZROTEST*). Although the modulus $\rho = 0$ can be found by squaring operations, the procedure usually will not find the root 0 without that test.

(c) In the program section labeled *SQUARING OPERATION* the iteratively squared coefficient is tested whether it will remain in the allowed range of numbers (formal parameter *range*) for a particular machine after another squaring operation.

(d) If there is a complex zero with a real part of 0, the resultant $R(p)$ is a polynomial of degree n with the coefficients $r_{n-1} = r_n = 0$. Computing the moduli of the zeros of this polynomial in the program section labeled *SQUARING OPERATION* and testing for pivotal coefficients, one would have to divide by 0. This case has been excluded by testing the divisor.

(e) If the acceptance criteria *epsilon* and *delta* are chosen too large, the sum of the multiplicities of the already found zeros may be greater than the degree n of the polynomial. In the program sections labeled *IT* and *D*, the test for the degree of the residual polynomial, the number of zeros, and the sum of the multiplicities of zeros in order to end the procedure has been improved.

Tests: The procedure *AG*4 has been tested on the CDC 1604-A computer at the Rechenzentrum, Technische Universität Hannover. The following results were obtained in a few representative cases. The parameters of acceptance criteria are *delta* = 0.2, *epsilon* = 10^{-7}, and *mm* = 10.

(i) $P_1(x) = x^8 - 30x^6 + 273x^4 - 820x^2 + 576$

$x_1 = 4.000\ 000\ 0010$ $\quad x_2 = -4.000\ 000\ 0010$
$x_3 = 2.999\ 999\ 9990$ $\quad x_4 = -2.999\ 999\ 9990$
$x_5 = 2.000\ 000\ 0000$ $\quad x_6 = -2.000\ 000\ 0000$
$x_7 = 1.000\ 000\ 0000$ $\quad x_8 = -1.000\ 000\ 0000$

(ii) $P_2(x) = x^5 + 7x^4 + 5x^3 + 6x^2 + 3x + 2$

$x_1 = -6.3509936102$
$x_{2,3} = 1.3506884657 \times 10^{-1} \pm i \times 7.7014185283 \times 10^{-1}$
$x_{4,5} = -4.5957204142 \times 10^{-1} \pm i \times 5.5126354891 \times 10^{-1}$

(iii) $P_3(x) = x^6 - 2x^5 + 2x^4 + x^3 + 6x^2 - 6x + 8$

$x_{1,2} = -9.9999999974 \times 10^{-1} \pm i \times 1.0000000002$
$x_{3,4} = 4.9999999999 \times 10^{-1} \pm i \times 8.6602540377 \times 10^{-1}$
$x_{5,6} = 1.4999999997 \pm i \times 1.3228756548$

(iv) $P_4(x) = x^2 - 4.01x + 4.02$

The procedure fails to compute any zero in this case (parameter $m = 0$). After changing the parameter *epsilon* to 10^{-5}, *AG*4 evaluates the zero $x = 2.0049937655$ with multiplicity 2 and remainder term 2.5×10^{-5};

Parameters:
- *n* degree of the polynomial
- *c* real coefficients of the polynomial $c[j](j=0,\cdots,n)$, where $c[n]$ is the constant term
- *delta*, *epsilon* parameters for acceptence criteria practical input *delta* = 0.2, *epsilon* = 10 ↑ (−7)
- *range* upper bound of the range of real constants (for the cDc 1604 -A *range* = 10 ↑ 307)
- *mm* number of root-squaring iterations practical input *mm* = 10
- *re* real part of each zero $re[j](j=1,\cdots,m)$
- *im* imaginary part of each zero $im[j](j=1,\cdots,m)$
- *mu* corresponding multiplicity $mu[j](j=1,\cdots,m)$
- *rt* remainder term $rt[j](j=1,\cdots,m)$
- *gc* coefficients of the polynomial generated from these zeros $gc[j](j=0,\cdots,n-i)$
- *m* number of distinct zeros found by the routine
- *i* degree of the residual polynomial
- *t* coefficients of the residual polynomial $t[j](j=0,\cdots,i)$, where $t[i]$ is the constant term
- *fail* a zero with multiplicity greater than *n* found, change parameters for acceptance criteria.

References:

1. Bareiss, E. H. Resultant procedure and the mechanization of the Graeffe process, *J. ACM 7* (Oct, 1960), 346–386.
2. Bareiss, E. H. and Fisherkeller, M. A. Algorithm 59, Zeros of a real polynomial by resultant procedure, *Comm. ACM 4* (May 1961), 236–237.
3. Thacher, H. C. Certification of algorithm 3, *Comm. ACM 3* (June 1960), 354.
4. Grau, A. A. Algorithm 256, Modified Graeffe method, *Comm. ACM 8* (June 1965), 379;


```
begin
  integer d,numzro;
  Boolean zero;
  numzro := 0;  zero := false;  d := n;
ZROTEST:
  if c[d] = 0 then
  begin
    zero := true;  d := d − 1;  numzro := numzro + 1;
    go to ZROTEST
  end;
  begin
    integer ct, nu, nuc, beta, j, jc, k, p, em, l, mmc, ll, me, sm;
    Boolean root;
    real x, y, gx, rp, h;
    array a, ac[0: d, 0: mm], rr, rc[0: d], s[−1: d],
      ag[0: d+1, −1: d+1], rh, q, g, f[1: 2 × d];
    switch ss := S1, S2;
    switch tt := T1, T2;
    switch vv := V1, V2;
    integer procedure min(u, v);  integer u, v;
      min := if u ≦ v then u else v;
    real procedure synd(ww, qq, ii, tt);
      integer ii;  real ww, qq;  array tt;
SYNTHETICDIV:
    begin
      s[−1] := 0;  s[0] := tt[0];
      for em := 1 step 1 until ii do
        s[em] := tt[em] − ww × s[em−1] − qq × s[em−2];
      if qq = 0 then synd := abs(s[ii])
      else synd := abs(s[ii−1] × sqrt(abs(qq))) + abs(s[ii])
    end synd;
    ct := beta := 1;
SQUARING OPERATION:
    me := mm;
    begin
      for m := 1 step 1 until mm do
      begin
        for j := 0 step 1 until d do
        begin
          h := 0;
          for ll := 1 step 1 until min(d−j,j) do
            h := h + (−1) ↑ ll × a[j−ll, m−1] × a[j+ll, m−1];
          a[j, m] := (−1) ↑ j × (a[j, m−1] ↑ 2 +2 × h)
        end;
        for l := 0 step 1 until d do
        begin
          if abs(a[l, m]) ≧ sqrt(range) then
            begin me := m;  go to W1 end
        end
      end
    end;
W1:
    for j := 0 step 1 until d do
    rr[j] :=if a[j, me] = 0 then 0 else
      (−1) ↑ j × a[j, me−1] ↑ 2/a[j, me];
    ll := 0;
```

```
    for j := d step -1 until 0 do
    begin
      if a[j, me] = 0 then
        begin ll := ll + 1;  rr[j] := ll end
      else go to W2
    end;
W2:
    j := 1;  nu := 1;
RD:
    if (1-delta≦rr[j]) ∧ (rr[j]≦1+delta) then
    begin
      rp := abs(a[j, me]/a[j-nu, me]) ↑ (1/(2↑me×nu));
      go to tt[beta]
    end;
M1:
    nu := nu + 1;
M2:
    j := j + 1;
    if j = d + 1 then go to ss[beta] else go to RD;
M3:
    nu := 1;  go to M2;
T1: rh[ct] := rp;  x := rp + epsilon × rp;
    y := x + epsilon × rp;
    for k := 0 step 1 until d do t[k] := abs(c[k]);
    f[ct] := synd(-y, 0.0, d, t) - synd(-x, 0.0, d, t);
    g[ct] := synd(-rh[ct], 0.0, d, c);
    if abs(f[ct]) > g[ct] then
    begin
      root := true;  q[ct] := 0;
      ct := ct + 1;  f[ct] := f[ct-1]
    end;
    rh[ct] := -rp;
    g[ct] := synd(-rh[ct], 0.0, d, c);
    if abs(f[ct]) > g[ct] then
    begin
      root := true;  q[ct] := 0;
      ct := ct + 1;  f[ct] := f[ct-1]
    end;
    if nu = 1 then go to M2;
    q[ct] := rp ↑ 2;  nuc := nu;  jc := j;
    mmc := me;
    for j := 0 step 1 until d do
    begin
      rc[j] := rr[j];  ac[j, me] := a[j, me]
    end;
RESULTANT:
    begin
      array b[-1:d+1, -1:d+1], aa[0:d],
        r[0:d, 0:d], cb[-1:d+1];
      cb[-1] := cb[d+1] := 0;
      for j := 0 step 1 until d do
        cb[j] := c[j];
      b[0, 0] := 1;
      for k := 0 step 1 until d do
      begin
        b[k, -1] := 0;  b[k-1, k] := 0;
        for j := 0 step 1 until k do
          b[k+1, j] := b[k, j-1] - q[ct] × b[k-1, j];
        b[k+1, k+1] := 1;  h := 0;
        for j := d - k step -1 until 0 do
          h := h + (cb[j] × cb[k+j] - cb[j-1]
          × cb[k+j+1]) × q[ct] ↑ (d-k-j);
        aa[k] := (-1) ↑ k × h;
        for j := 0 step 1 until k - 1 do
          r[k, j] := r[k-1, j] + aa[k] × b[k, j];
        r[k, k] := aa[k]
      end;
      beta := 2;
      for j := 0 step 1 until d do
        a[j, 0] := r[d, d-j]/r[d, d]
    end;
    go to SQUARING OPERATION;
T2:
    if (rp/2) ↑ 2 > q[ct] then go to M3;
    rh[ct] := rp;
    g[ct] := synd(-rh[ct], q[ct], d, c);
    if abs(f[ct]) > g[ct] then
    begin
      ct := ct + 1;  f[ct] := f[ct-1];
      q[ct] := q[ct-1]
    end;
    rh[ct] := -rp;
    g[ct] := synd(-rh[ct], q[ct], d, c);
    if abs(f[ct]) > g[ct] then
    begin
      ct := ct + 1;  f[ct] := f[ct-1];
      q[ct] := q[ct-1]
    end;
    go to M3;
S2:
    me := mmc;
    for j := 0 step 1 until d do
    begin
      a[j, me] := ac[j, me];  rr[j] := rc[j]
    end;
    j := jc;  beta := 1;
    if root then go to M3 else nu := nuc;
    go to M1;
S1:
    for j := 0 step 1 until d do ag[j, 0] := 1;
    for j := -1, 1 step 1 until d do
    for m := 0 step 1 until d do
      ag[m, j] := 0;
    k := 1;  i := d;  m := 1;  ll := 0;
    for j := 0 step 1 until d do t[j] := c[j];
MULT:
    mu[m] := 0;
    p := if q[k] = 0 then 1 else 2;
IT:
    gx := synd(-rh[k], q[k], i, t);
    if abs(f[k]) > gx then
    begin
      ll := ll + p;
      for j := 1 step 1 until ll do
        ag[ll, j] := ag[ll-p, j] - rh[k] × ag[ll-p, j-1] + q[k] ×
          ag[ll-p, j-2];
      mu[m] := mu[m] + p;  i := i - p;
      if i < 0 then go to fail;
      if i = 0 then go to E1;
      for j := 0 step 1 until i do t[j] := s[j];
      go to IT
    end
    else if mu[m] ≠ 0 then
E1:
    begin
      rt[m] := q[k];  go to vv[p];
    end
    else go to D1;
V1:
    re[m] := rh[k];  im[m] := 0;  go to E;
V2:
    re[m] := rh[k]/2;
    im[m] := sqrt(q[k] - re[m] ↑ 2);
E:
    m := m + 1;
```

```
D1:
    k := k + 1;
    sm := 0;
    if m ≠ 1 then
    for j := 1 step 1 until m - 1 do sm := sm + mu[j];
    if k ≦ ct ∧ sm ≦ d ∧ i > 0 then go to MULT;
    for j := 0 step 1 until d do gc[j] := ag[ll, j];
    m := m - 1;
    if zero then
    begin
      for j := d + 1 step 1 until d + numzro do gc[j] := 0;
      m := m + 1;
      re[m] := 0;  im[m] := 0;  mu[m] := numzro;  rt[m] := 0
    end
  end
end AG4
```

ALGORITHM 341
SOLUTION OF LINEAR PROGRAMS IN 0-1 VARIABLES BY IMPLICIT ENUMERATION [H]

J. L. Byrne and L. G. Proll

(Recd. 8 Nov. 1967 and 17 June 1968)

Department of Mathematics, University of Southampton, Hampshire, England

KEY WORDS AND PHRASES: linear programming, zero-one variables, partial enumeration
CR CATEGORIES: 5.41

```
procedure IMPLEN (m, n, A, x, api, nosoln, count, inf);
  value m, n, inf;  integer m, n, count;  real inf;
  Boolean api, nosoln;  real array A; integer array x;
comment This procedure solves the integer linear program,
  minimize    A[0, 1] × x[1] + ··· + A[0, n] × x[n]
  subject to  A[i, 1] × x[1] + ··· + A[i, n] × x[n]
                                + A[i, 0] ≧ 0  (i=1, 2, ··· , m)
  and          x[j] = 0 or 1  (j=1, 2, ··· , n).
  It is assumed that A[0, j] ≧ 0  (j=1, 2, ··· , n). The algorithm
  used is that of Geoffrion (SIAM Rev. 9, No. 2). On entry, inf
  is the largest positive real number available and api is set to
  true if a priori information concerning the solution is supplied
  in the form of a binary vector x[1: n] and its associated cost
  A[0, 0]. On exit nosoln is true if no feasible solution to the con-
  straints has been found, otherwise it is false and x contains the
  optimal solution, A[0, 0] contains the optimal value of the ob-
  jective function and A[i, 0] contains the values of the slack
  variables. In either case count contains the number of iterations
  performed;
begin
  integer i, j, k, ia, e, d;  real z, q, max, r;  Boolean null;
  integer array s, v[1: n];
  comment  s holds the current partial solution in order of as-
    signment, v is a state vector associated with s;
  if api then
  begin
    for j := 1 step 1 until n do
    if x[j] = 0 then begin s[j] := -j;  v[j] := 2 end
    else
    begin
      s[j] := j;  v[j] := 3;
      for i := 1 step 1 until m do
        A[i, 0] := A[i, 0] + A[i, j]
    end;
    e := n;  z := A[0, 0];  go to L0
  end;
  for j := 1 step 1 until n do s[j] := v[j] := 0;
  z := 0.0;  e := 0;
L0: nosoln := true;  count := 0;  A[0, 0] := inf;
  comment  all relevant variables are now initialized;
START: count := count + 1;
  for i := 1 step 1 until m do
    if A[i, 0] < 0.0 then go to FORMT;
  comment  best completion of s is feasible;
  go to INCUMBENT;
FORMT: null := true;
  comment  form set T of free variables to which 1 may be profit-
    ably assigned;
  for j := 1 step 1 until n do
  begin
    if ¬ (v[j] = 0 ∧ A[0, j] + z < A[0, 0]) then go to L1;
    for k := i step 1 until m do
    if A[k, 0] < 0.0 ∧ A[k, j] > 0.0 then
      begin null := false;  v[j] := 1;  go to L1 end;
L1: end;
  if null then go to NEWS;
  comment  if T is empty then s is fathomed;
  for k := i step 1 until m do
  begin
    if A[k, 0] ≧ 0.0 then go to L2;
    q := A[k, 0];
    for j := 1 step 1 until n do
      if v[j] = 1 ∧ A[k, j] > 0.0 then q := q + A[k, j];
    if q < 0.0 then go to NEWS;
    comment  if q is negative s is fathomed;
L2: end;
  max := -inf;
  for j := 1 step 1 until n do
  begin
    if v[j] ≠ 1 then go to L3;  q := 0.0;
    for i := 1 step 1 until m do
    begin
      r := A[i, 0] + A[i, j];
      if r < 0.0 then q := q + r
    end;
    if max ≦ q then
      begin max := q;  d := j end;
L3: end;
  e := e + 1;  s[e] := d;  v[d] := 3;  ia := 1;
  comment  Augment s by assigning 1 to x[d];
RESET: for j := 1 step 1 until n do
    if v[j] = 1 then v[j] := 0;
  comment  clear T;
  for i := 1 step 1 until m do
    A[i, 0] := A[i, 0] + ia × A[i, d];
  z := z + ia × A[0, d];
  comment  Recalculate slacks and objective function;
  go to START;
INCUMBENT: nosoln := false;
  if z ≧ A[0, 0] then go to NEWS;
  A[0, 0] := z;
  if api then begin api := false; go to L4 end;
  for j := 1 step 1 until n do
    x[j] := if v[j] = 3 then 1 else 0;
NEWS: if e = 0 then go to RESULT;
L4: d := s[e];
  if d > 0 then go to UNDERLINE;
  v[-d] := 0;  e := e - 1;  comment  backtrack;
  go to NEWS;
UNDERLINE: s[e] := -d;  v[d] := 2;  ia := -1;
  comment  Assign 0 to x[d];
  go to RESET;
RESULT:
end
```