ALGORITHM 345
AN ALGOL CONVOLUTION PROCEDURE BASED ON THE FAST FOURIER TRANSFORM [C6]

RICHARD C. SINGLETON* (Recd. 30 Dec. 1966, 26 July 1967, 19 July 1968, and 8 Nov. 1968)
Stanford Research Institute, Menlo Park, CA 94025

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

KEY WORDS AND PHRASES: fast Fourier transform, complex Fourier transform, multivariate Fourier transform, Fourier series, harmonic analysis, spectral analysis, orthogonal polynomials, orthogonal transformation, convolution, autocovariance, autocorrelation, cross-correlation, digital filtering, permutation
CR CATEGORIES: 3.15, 3.83, 5.12, 5.14

Stockham [6] and Gentleman and Sande [3] have shown the practical advantages of computing the circular convolution

$$C_k = \sum_{j=0}^{n-1} A_j B_{(j+k) \bmod n}, \qquad k = 0, 1, \cdots, n-1,$$

of two real vectors A and B of period n by the fast Fourier transform [2, 3, 4]. The Fourier transforms

$$\alpha_j = \sum_{p=0}^{n-1} A_p \exp(i2\pi pj/n)$$

and

$$\beta_j = \sum_{q=0}^{n-1} B_q \exp(i2\pi qj/n)$$

are first computed, then the convolution

$$C_k = \frac{1}{n} \sum_{j=0}^{n-1} \alpha_j \beta_j^* \exp(i2\pi jk/n)$$

where β_j^* is the complex conjugate of β_j. By this method the number of arithmetic operations increases by a factor slightly more than 2 when n is doubled, as compared with a factor of 4 for the direct method. Tests show a 16 to 1 time advantage for the transform method at $n = 256$.

The operation of convolution is used in computing autocorrelation and cross-correlation functions, in digital filtering of time series, and many other applications.

Procedure *CONVOLUTION* computes the convolution of two real vectors of dimension $n = 2^m$. The special features of this procedure are: (1) the usual reordering of the fast Fourier transform results is avoided, and (2) the return from frequency to time is made with a transform of dimension $n/2$ instead of n. The two vectors A and B are first transformed with a single complex Fourier transform of dimension n. The complex product $\alpha\beta^*$ is then formed, leaving the result in reverse binary order. Since the convolution is real-valued, the real part x of the complex product is an even function and the imaginary part y is an odd function; thus the Fourier transform of x is real and that of y is imaginary. These properties lead to the identity

$$\begin{aligned} T(x + iy) &= \mathrm{Re}(Tx) - \mathrm{Im}(Ty) \\ &= \mathrm{Re}(T(x - y)) + \mathrm{Im}(T(x - y)) \end{aligned}$$

where T represents the Fourier transform and $T(x + iy)$ is the desired convolution. We subtract y from x, yielding a real vector of dimension n, then transform using a complex transform of dimension $n/2$ and add the resulting cosine and sine coefficients to give the convolution. Thus with procedure *CONVOLUTION* we make maximum use of the complex Fourier transform in each direction and avoid any reverse binary to binary permutation. The Fourier transform

$$T(A + iB) = \alpha + i\beta$$

of the two original vectors is available in reverse binary order on exit from the procedure. We can permute this transform to normal order with procedure *REVERSEBINARY* and readily compute the power spectra and cross spectrum of the two data vectors.

Procedure *CONVOLUTION* uses procedure *REALTRAN*, given in Algorithm 338 [5], but repeated here with revisions to improve accuracy on computers using truncated floating-point arithmetic. Procedures *FFT*4 and *REVFFT*4 are also used and perform the same computation as procedures *FFT*2 and *REVFFT*2 given in Algorithm 338 for use on a system with virtual memory. The transform procedures given here are organized without regard to the problem of memory overlay. This change yields a 10 percent reduction in computing time on the Burroughs B5500 for transforms of dimension $n = 512$ or smaller. Procedure *FFT*4 is based on an organization of the fast Fourier transform due to Sande [3], and procedure *REVFFT*4 is similar to the method proposed by Cooley and Tukey [2], except that the data is in reverse binary order. In both cases, trigonometric functions are used in normal sequence, rather than reverse binary sequence, thus eliminating the need for a reverse binary counter. Another gain in efficiency comes from reducing the time for computing trigonometric function values. The following difference-equation method is used:

$$\cos((k + 1)\theta) = \cos(k\theta) - (C \times \cos(k\theta) + S \times \sin(k\theta))$$

and

$$\sin((k + 1)\theta) = \sin(k\theta) + (S \times \cos(k\theta) - C \times \sin(k\theta)),$$

where the constant multipliers are $C = 2\sin^2(\theta/2)$ and $S = \sin(\theta)$, and the initial values are $\cos(0) = 1$ and $\sin(0) = 0$.

These initial values should be computed to full machine precision; if necessary, a stored table of $\sin(\theta)$ for $\theta = \pi/2, \pi/4, \pi/8, \cdots, \pi/n$ can be added to procedures *FFT*4 and *REVFFT*4. Using the standard sine function to compute initial values, the ratio of rms error to rms data is about 2×10^{-11} for the transform-inverse pair at $n = 512$ on the Burroughs B5500 computer; this error is about the same as that obtained when the sine and cosine functions are used for all trigonometric function values. On a computer using truncated, rather than rounded, arithmetic operations, the sequence of values for $\cos(k\theta) + i \sin(k\theta)$ tends to spiral inward from the unit circle. Since the error is primarily one of magnitude, rather than angle, rescaling to the unit circle at each step gives a satisfactory correction. This correction is included in procedures *FFT*4 and *REVFFT*4 but may be removed to improve running speed if rounded arithmetic is used.

Procedures *FFT*8 and *REVFFT*8 are included as possible substitutes for *FFT*4 and *REVFFT*4. These procedures use radix 8 arithmetic [1], rather than radix 4, and run about 20 percent faster on the Burroughs B5500 computer; however, the compiled code is twice as long. The code could be shortened by use of subscripted variables and FOR statements, but this change would probably eliminate most of the time-saving.

The permutation procedure *REVERSEBINARY* is based on a modified dual counter, one in normal sequence and the other in reverse binary sequence. In permuting a vector of dimension n, the normal sequence counter goes from 1 to $n/2 - 1$, and the elements indexed $1, 3, \cdots, n/2 - 1$ are exchanged with their reverse-binary counterparts (indexed greater than or equal to $n/2$) without need of a test. The reverse binary counter is incremented only $n/4$ times, and exchanges of pairs of elements below $n/2$ are done jointly with pair exchanges in the upper half of the array; i.e. if x_j and x_k are exchanged, where $j,\ k < n/2$, then x_{n-1-j} and x_{n-1-k} are also exchanged. This procedure is twice as fast on the Burroughs B5500 as *REORDER* given in Algorithm 338 [5] and is the better choice when the additional features of *REORDER* are not needed. For a single-variate, complex Fourier transform of dimension $n = 2^m$,

$$REVERSEBINARY(A, B, m);$$
$$REVFFT8(A, B, n, m, 1)$$

was found to be the best combination for $n \leq 512$ on the B5500 computer, giving a time of 0.79 sec. for $n = 512$.

References:

1. Bergland, G. D. A fast Fourier transform algorithm using base 8 iterations. *Math. Comput. 22*, 102 (Apr. 1968), 275–279.
2. 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.
3. Gentleman, W. G., and Sande, G. Fast Fourier transforms—for fun and profit. Proc. AFIPS 1966 Fall Joint Comput. Conf., Vol. 29, Spartan Books, New York, 1966, pp. 563–578.
4. Singleton, R. C. On computing the fast Fourier transform. *Comm. ACM 10* (Oct. 1967), 647–654.
5. Singleton, R. C. Algorithm 338, Algol procedures for the fast Fourier transform. *Comm. ACM 11* (Nov. 1968), 773–776.
6. Stockham, T. G. High-speed convolution and correlation. Proc. AFIPS 1966 Spring Joint Comput. Conf., Vol. 28, Spartan Books, New York, 1966, pp. 229–233;

procedure *CONVOLUTION* $(A, B, C, D, m, scale)$;
value m, $scale$; **integer** m; **real** $scale$; **array** A, B, C, D;
comment This procedure computes the circular convolution

$$C_k = scale \sum_{j=0}^{n-1} A_j B_{(j+k) \bmod n}, \qquad k = 0, 1, \cdots, n-1,$$

where $n = 2^m$ and $p \bmod n$ represents the remainder after division of p by n. (It is assumed that $m \geq 1$.) Arrays $A, B[0 : n-1]$ originally contain the two data vectors to be convoluted, and on exit, contain the Fourier transform of $A + iB$ arranged in reverse binary order. A and B must not be the same array. On exit, array $C[0 : n-1]$ contains the convolution multiplied by the factor *scale*. Array D is a scratch storage array with lower bound zero and upper bound at least $n \div 2$. If the Fourier transform of the data is not needed, the procedure can be called with arrays A and B used for C and D in either order, for example, *CONVOLUTION* $(A, B, A, B, m, scale)$. If the Fourier transform is used, it should first be permuted to normal order by the call $REVERSEBINARY(A, B, m)$. After doing this, the Fourier cosine coefficients of the A vector are

$$(A_k + A_{n-k})/n, \qquad k = 1, 2, \cdots, n/2,$$
$$(2A_0)/n, \qquad k = 0,$$

and the sine coefficients are

$$(B_k - B_{n-k})/n, \qquad k = 1, 2, \cdots, n/2 - 1.$$

The Fourier cosine coefficients of the B vector are

$$(B_k + B_{n-k})/n, \qquad k = 1, 2, \cdots, n/2,$$
$$(2B_0)/n, \qquad k = 0,$$

and the sine coefficients are

$$(A_{n-k} - A_k)/n, \qquad k = 1, 2, \cdots, n/2 - 1.$$

The procedures *FFT*4, *REVFFT*4, and *REALTRAN* are used by this procedure and must also be declared. If convolutions of large dimension are to be computed on a system with virtual memory, procedures *FFT*2 and *REVFFT*2 (Algorithm 338) [5] should be substituted for procedures *FFT*4 and *REVFFT*4;

```
begin integer j, kk, ks, n;  real aa, ab, ba, bb, im;
  n := 2↑m;  j := 1;
  FFT4(A, B, n, m, n);
  C[0] := 4 × (A[0]×B[0]);
L:  kk := j;  ks := j := j + j;
L2: ks := ks − 1;
  aa := A[kk] + A[ks];  ab := A[kk] − A[ks];
  ba := B[kk] + B[ks];  bb := B[kk] − B[ks];
  im := ba × bb + aa × ab;  aa := aa × ba − ab × bb;
  C[kk] := aa − im;  C[ks] := aa + im;
  kk := kk + 1;  if kk < ks then go to L2;
  if j < n then go to L;
  kk := n ÷ 2;  ks := kk − 1;  scale := scale/(8×n);
  for j := 0 step 1 until ks do D[j] := C[j+kk];
  REVFFT4(C, D, kk, m−1, 1);
  REALTRAN(C, D, kk, false);
  C[0] := scale × C[0];  C[kk] := scale × C[kk];
  for j := 1 step 1 until ks do
  begin C[n−j] := scale × (C[j]−D[j]);
  end
  C[j] := scale × (C[j]+D[j])
end CONVOLUTION;
```

procedure *FFT*4(A, B, n, m, ks); **value** n, m, ks;
integer n, m, ks; **array** A, B;
comment This procedure 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 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 $j = 0, 1, \cdots, n-1$ is computed, where $(a+ib)$ represent the initial values and $(x+iy)$ represent the transformed values;

```
begin integer k0, k1, k2, k3, k, span;
  real A0, A1, A2, A3, B0, B1, B2, B3, re, im;
  real rad, dc, ds, c1, c2, c3, s1, s2, s3;
  span := ks;  ks := 2↑m;  rad := 4.0 × arctan(1.0)/ks;
  ks := span ÷ ks;  n := n − 1;  k := m;
  for m := m − 2 while m ≧ 0 do
  begin
    c1 := 1.0;  s1 := 0;  k0 := 0;  k := ks;
    dc := 2.0 × sin(rad)↑2;  rad := rad + rad;
    ds := sin(rad);  rad := rad + rad;
    span := span ÷ 4;
La: k1 := k0 + span;  k2 := k1 + span;  k3 := k2 + 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 + A2 + A1 + A3;
    B[k0] := B0 + B2 + B1 + B3;
    if s1 = 0 then
    begin
      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
    else
    begin
      re := A0 + A2 − A1 − A3;  im := B0 + B2 − B1 − B3;
      A[k1] := re × c2 − im × s2;
      B[k1] := re × s2 + im × c2;
      re := A0 − A2 − B1 + B3;  im := B0 − B2 + A1 − A3;
      A[k2] := re × c1 − im × s1;
      B[k2] := re × s1 + im × c1;
      re := A0 − A2 + B1 − B3;  im := B0 − B2 − A1 + A3;
      A[k3] := re × c3 − im × s3;
      B[k3] := re × s3 + im × c3
    end;
    k0 := k3 + span;  if k0 < n then go to La;
    k0 := k0 − n;  if k0 ≠ k then go to La;
    comment If computing for the current factor of 4 is not
      finished then increment the sine and cosine values;
    if k0 ≠ span then
    begin
      c2 := c1 − (dc×c1+ds×s1);
      s1 := (ds×c1−dc×s1) + s1;
      comment The following three statements compensate
        for truncation error. If rounded arithmetic is used, sub-
        stitute c1 := c2;
      c1 := 1.5−0.5 × (c2↑2+s1↑2);
      s1 := c1 × s1;  c1 := c1 × c2;
      c2 := c1↑2 − s1↑2;  s2 := 2.0 × c1 × s1;
      c3 := c2 × c1 − s2 × s1;  s3 := c2 × s1 + s2 × c1;
      k := k + ks;  go to La
    end;
    k := m
  end;
  comment If m is odd then compute for one factor of 2;
  if k ≠ 0 then
  begin
    span := span ÷ 2;  k0 := 0;
Lb: k2 := k0 + span;  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;
    k0 := k2 + span;  if k0 < n then go to Lb;
    k0 := k0 − n;  if k0 ≠ span then go to Lb
  end
end FFT4;
procedure REVFFT4(A, B, n, m, ks);  value n, m, ks;
  integer n, m, ks;  array A, B;
comment This procedure 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 = n1 × n2 × ··· × np
  for a p-variate transform, and ks = nk+1 × nk+2 × ··· × np ,
  where nk = 2^m is the dimension of the current variable. Arrays
  A[0:n−1] and B[0:n−1] originally contain the real and imagi-
  nary components of the data with the indices of each variable
  in reverse binary order, e.g. ajkl is in A[j'×n2×n3+k'×n3+l']
  for j = 0, 1, ··· , n1 − 1,  k = 0, 1, ··· n2 − 1,  and  l = 0,
  1, ··· n3 − 1, where j', k', and l' are the bit-reversed values of
  j, k, and l. On completion of the multivariate transform, the
  real and imaginary components of the resulting Fourier coeffi-
  cients 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, k, span;
  real A0, A1, A2, A3, B0, B1, B2, B3;
  real rad, dc, ds, c1, c2, c3, s1, s2, s3;
  rad := 4.0 × arctan(1.0);  n := n − 1;
  k0 := 0;  span := ks;
  comment If m is odd then compute for one factor of 2;
  if (m÷2) × 2 ≠ m then
  begin
La: k2 := k0 + span;  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;
    k0 := k2 + span;  if k0 < n then go to La;
    k0 := k0 − n;  if k0 ≠ span then go to La;
    span := span + span;  rad := 0.5 × rad
  end;
  for m := m − 2 while m ≧ 0 do
  begin
    c1 := 1.0;  s1 := 0;  k0 := 0;  rad := 0.25 × rad;
    dc := 2.0 × sin(rad)↑2;
    ds := sin(rad+rad);  k := ks;
Lb: k1 := k0 + span;  k2 := k1 + span;  k3 := k2 + span;
    A0 := A[k0];  B0 := B[k0];
    if s1 = 0 then
    begin
      A2 := A[k1];  B2 := B[k1];
      A1 := A[k2];  B1 := B[k2];
      A3 := A[k3];  B3 := B[k3]
    end
    else
    begin
      A2 := A[k1] × c2 − B[k1] × s2;
      B2 := A[k1] × s2 + B[k1] × c2;
      A1 := A[k2] × c1 − B[k2] × s1;
      B1 := A[k2] × s1 + B[k2] × c1;
      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 − B1 + B3;
    B[k1] := B0 − B2 + A1 − A3;
```

```
    A[k2] := A0 + A2 - A1 - A3;
    B[k2] := B0 + B2 - B1 - B3;
    A[k3] := A0 - A2 + B1 - B3;
    B[k3] := B0 - B2 - A1 + A3;
    k0 := k3 + span;  if k0 < n then go to Lb;
    k0 := k0 - n;  if k0 ≠ k then go to Lb;
    comment If computing for the current factor of 4 is not
      finished then increment the sine and cosine values;
    if k0 ≠ span then
    begin
      c2 := c1 - (dc×c1+ds×s1);
      s1 := (ds×c1-dc×s1) + s1;
      comment The following three statements compensate
        for truncation error. If rounded arithmetic is used, sub-
        stitute c1 := c2;
      c1 := 1.5-0.5 × (c2↑2+s1↑2);
      s1 := c1 × s1;  c1 := c1 × c2;
      c2 := c1↑ 2 - s1↑2;  s2 := 2.0 × c1 × s1;
      c3 := c2 × c1 - s2 × s1;  s3 := c2 × s1 + s2 × c1;
      k := k + ks;  go to Lb
    end;
    span := 4 × span
  end
end REVFFT4;
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 trans-
  forms to give the Fourier cosine coefficients A[0], A[1], ··· ,
  A[n] and sine coefficients B[0], B[1], ··· , 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;
  nh := n ÷ 2;  ds := 2.0 × arctan(1.0)/n;
  dc := 2.0 × sin(ds) ↑ 2;  ds := sin(ds+ds);
  sk := 0;
  if evaluate then
    begin ck := -1.0;  ds := -ds end
  else begin  ck := 1.0;  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;
    aa := ck - (dc×ck+ds×sk);
    sk := (ds×ck-dc×sk) + sk;
    comment The following three statements compensate for
      truncation error. If rounded arithmetic is used, substitute
      ck := aa;
    ck := 1.5-0.5 × (aa↑2+sk↑2);
    sk := ck × sk;  ck := ck × aa
  end
end REALTRAN;
procedure REVERSEBINARY(A, B, m);  value m;
  integer m; array A, B;
comment This procedure permutes the elements A[j] and B[j]
  of arrays A and B, for j = 0, 1, ··· , 2↑m - 1, according to
  the reverse binary transformation. Element
```

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

is moved to location

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

```
  Two successive calls of this procedure give an identity trans-
  formation;
  begin integer j, jj, k, lim, jk, n2, n4, n8, nn;
  real t;
  integer array C[0:m];
  C[0] := nn := 1;  jj := 0;
  for j := 1 step 1 until m do C[j] := nn := nn + nn;
  if m > 1 then n4 := C[m-2];  if m > 2 then n8 := C[m-3];
  n2 := C[m-1];  lim := n2 - 1;  nn := nn - 1;  m := m - 4;
  for j := 1 step 1 until lim do
  begin
    jk := jj + n2;
    t := A[j];  A[j] := A[jk];  A[jk] := t;
    t := B[j];  B[j] := B[jk];  B[jk] := t;
    j := j + 1;
    if jj ≧ n4 then
    begin
    jj := jj - n4;
      if jj ≧ n8 then
      begin
        jj := jj - n8;  k := m;
L:  if C[k] ≦ jj then
  begin jj := jj - C[k];  k := k - 1;  go to L end;
    jj := C[k] + jj
      end
      else jj := jj + n8
    end
    else jj := jj + n4;
    if jj > j then
    begin
      k := nn - j;  jk := nn - jj;
      t := A[j];  A[j] := A[jj];  A[jj] := t;
      t := B[j];  B[j] := B[jj];  B[jj] := t;
      t := A[k];  A[k] := A[jk]; A[jk] := t;
      t := B[k];  B[k] := B[jk]; B[jk] := t
    end
  end
end REVERSEBINARY;
procedure FFT8(A, B, n, m, ks);  value n, m, ks;
  integer n, m, ks;  array A, B;
```

comment This procedure 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, $ks = n_k \times n_{k+1} \times \cdots \times n_p$, 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 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 $j = 0, 1, \cdots, n-1$ is computed, where $(a+ib)$ represent the initial values and $(x+iy)$ represent the transformed values;

```
begin integer k0, k1, k2, k3, k4, k5, k6, k7, k, span;
  real A0, A1, A2, A3, A4, A5, A6, A7, B0, B1, B2, B3, B4, B5,
    B6, B7, x0, x1, x2, x3, x4, x5, x6, x7, y0, y1, y2, y3, y4, y5, y6, y7,
    c1, c2, c3, c4, c5, c6, c7, s1, s2, s3, s4, s5, s6, s7, c45, dc, ds, rad;
  span := ks;  ks := 2↑m;  rad := 4.0 × arctan(1.0)/ks;
  ks := span ÷ ks;  n := n − 1;  c45 := sqrt(0.5);  k := m;
  comment  Radix 8 transform;
  for m := m − 3 while m ≧ 0 do
  begin
    c1 := 1.0;  s1 := 0;  k0 := 0;  k := ks;
    dc := 2.0 × sin(rad)↑2;  rad := rad + rad;
    ds := sin(rad);  rad := 4 × rad;
    span := span ÷ 8;
La: k1 := k0 + span;  k2 := k1 + span;  k3 := k2 + span;
    k4 := k3 + span;  k5 := k4 + span;  k6 := k5 + span;
    k7 := k6 + 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];
    A4 := A[k4];  B4 := B[k4];
    A5 := A[k5];  B5 := B[k5];
    A6 := A[k6];  B6 := B[k6];
    A7 := A[k7];  B7 := B[k7];
    x0 := A0 + A4;  y0 := B0 + B4;
    x4 := A0 − A4;  y4 := B0 − B4;
    x1 := A1 + A5;  y1 := B1 + B5;
    x5 := (A1 − A5 − B1 + B5) × c45;
    y5 := (A1 − A5 + B1 − B5) × c45;
    x2 := A2 + A6;  y2 := B2 + B6;
    x6 := B6 − B2;  y6 := A2 − A6;
    x3 := A3 + A7;  y3 := B3 + B7;
    x7 := (A7−A3−B3+B7) × c45;
    y7 := (A3−A7−B3+B7) × c45;
    A1 := x0 + x2 − x1 − x3;  B1 := y0 + y2 − y1 − y3;
    A2 := x0 − x2 − y1 + y3;  B2 := y0 − y2 + x1 − x3;
    A3 := x0 − x2 + y1 − y3;  B3 := y0 − y2 − x1 + x3;
    A4 := x4 + x6 + x5 + x7;  B4 := y4 + y6 + y5 + y7;
    A5 := x4 + x6 − x5 − x7;  B5 := y4 + y6 − y5 − y7;
    A6 := x4 − x6 − y5 + y7;  B6 := y4 − y6 + x5 − x7;
    A7 := x4 − x6 + y5 − y7;  B7 := y4 − y6 − x5 + x7;
    A[k0] := x0 + x2 + x1 + x3;  B[k0] := y0 + y2 + y1 + y3;
    if s1 = 0 then
    begin
      A[k1] := A1;  B[k1] := B1;
      A[k2] := A2;  B[k2] := B2;
      A[k3] := A3;  B[k3] := B3;
      A[k4] := A4;  B[k4] := B4;
      A[k5] := A5;  B[k5] := B5;
      A[k6] := A6;  B[k6] := B6;
      A[k7] := A7;  B[k7] := B7
    end
    else
    begin
      A[k1] := c4 × A1 − s4 × B1;
      B[k1] := s4 × A1 + c4 × B1;
      A[k2] := c2 × A2 − s2 × B2;
      B[k2] := s2 × A2 + c2 × B2;
      A[k3] := c6 × A3 − s6 × B3;
      B[k3] := s6 × A3 + c6 × B3;
      A[k4] := c1 × A4 − s1 × B4;
      B[k4] := s1 × A4 + c1 × B4;
      A[k5] := c5 × A5 − s5 × B5;
      B[k5] := s5 × A5 + c5 × B5;
      A[k6] := c3 × A6 − s3 × B6;
      B[k6] := s3 × A6 + c3 × B6;
      A[k7] := c7 × A7 − s7 × B7;
      B[k7] := s7 × A7 + c7 × B7
    end;
    k0 := k7 + span;  if k0 < n then go to La;
    k0 := k0 − n;  if k0 ≠ k then go to La;
    comment Increment sine and cosine values;
    if k0 ≠ span then
    begin
      c2 := c1 − (dc×c1+ds×s1);
      s1 := (ds×c1−dc×s1) + s1;
      comment  The following three statements compensate
        for truncation error. If rounded arithmetic is used,
        substitute c1 := c2;
      c1 := 1.5−0.5 × (c2↑2+s1↑2);
      s1 := c1 × s1;  c1 := c1 × c2;
      c2 := c1↑2 − s1↑2;  s2 := 2.0 × c1 × s1;
      c3 := c2 × c1−s2 × s1;  s3 := c2 × s1 + s2 × c1;
      c4 := c2↑2 − s2↑2;  s4 := 2.0 × c2 × s2;
      c5 := c1 × c4 − s1 × s4;  s5 := s1 × c4 + c1 × s4;
      c6 := c3↑2 − s3↑2;  s6 := 2.0 × c3 × s3;
      c7 := c1 × c6 − s1 × s6;  s7 := s1 × c6 + c1 × s6;
      k := k + ks;  go to La
    end;
    k3 := m
  end;
  comment  If m is not a multiple of 3, then complete the trans-
    form with radix 2 steps;
  for k3 := k3 − 1 while k3 ≧ 0 do
  begin
    k0 := 0;  span := span ÷ 2;
Lb: k2 := k0 + span;
    A2 := A[k2];  B2 := B[k2];
    A[k2] := A[k0] − A2;  B[k2] := B[k0] − B2;
    A[k0] := A[k0] + A2;  B[k0] := B[k0] + B2;
    k0 := k2 + span;  if k0 < n then go to Lb;
    k0 := k0 − n;  if k0 < ks then go to Lb;
    if ks = span then go to Ld;
Lc: k2 := k0 + span;
    A2 := A[k0] − A[k2];  B2 := B[k0] − B[k2];
    A[k0] := A[k0] + A[k2];  B[k0] := B[k0] + B[k2];
    A[k2] := −B2;  B[k2] := A2;
    k0 := k2 + span;  if k0 < n then go to Lc;
    k0 := k0 − n;  if k0 < span then go to Lc;
Ld: end
end FFT8;
procedure REVFFT8(A, B, n, m, ks);  value n, m, ks;
  integer n, m, ks;  array A, B;
comment  This procedure 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 = n1 × n2 × ··· × np
  for a p-variate transform, and ks = n(k+1) × n(k+2) × ··· × np ,
  where nk = 2^m is the dimension of the current variable. Arrays
  A[0:n−1] and B[0:n−1] originally contain the real and imagi-
  nary components of the data with the indices of each variable
  in reverse binary order, e.g. a(jkl) is in A[j'×n2×n3+k'×n3+l']
  for j = 0, 1, ··· , n1 − 1, k = 0, 1, ··· , n2 − 1, and l =
  0, 1, ··· , n3 − 1, where j', k', and l' are the bit-reversed values
  of j, k, and l. On completion of the multivariate transform, the
  real and imaginary components of the resulting Fourier coeffi-
  cients 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, k5, k6, k7, k, span;
  real A0, A1, A2, A3, A4, A5, A6, A7, B0, B1, B2, B3, B4, B5,
    B6, B7, x0, x1, x2, x3, x4, x5, x6, x7, y0, y1, y2, y3, y4, y5, y6, y7,
```

```
    c1, c2, c3, c4, c5, c6, c7, s1, s2, s3, s4, s5, s6, s7, c45, dc, ds, rad;
  rad := 4.0 × arctan(1.0);  n := n − 1;
  c45 := sqrt(0.5);  span := ks;
  comment Compute radix 2 steps if m is not a multiple of 3;
  k3 := (m÷3) × 3;
  for k3 := k3 + 1 while k3 ≦ m do
  begin
    k0 := 0;
La: k2 := k0 + span;
    A2 := A[k2];  B2 := B[k2];
    A[k2] := A[k0] − A2;  B[k2] := B[k0] − B2;
    A[k0] := A[k0] + A2;  B[k0] := B[k0] + B2;
    k0 := k2 + span;  if k0 < n then go to La;
    k0 := k0 − n;  if k0 < ks then go to La;
    if ks = span then go to Lc;
Lb: k2 := k0 + span;
    A2 := A[k2];  B2 := B[k2];
    A[k2] := A[k0] + B2;  B[k2] := B[k0] − A2;
    A[k0] := A[k0] − B2;  B[k0] := B[k0] + A2;
    k0 := k2 + span;  if k0 < n then go to Lb;
    k0 := k0 − n;  if k0 < span then go to Lb;
Lc: span := span + span;  rad := 0.5 × rad
  end;
  comment Radix 8 transform;
  for m := m − 3 while m ≧ 0 do
  begin
    c1 := 1.0;  s1 := 0;  k0 := 0;  k := ks;
    rad := 0.125 × rad;  dc := 2.0 × sin(rad) ↑ 2;
    ds := sin(rad+rad);
Ld: k1 := k0 + span;  k2 := k1 + span;  k3 := k2 + span;
    k4 := k3 + span;  k5 := k4 + span;  k6 := k5 + span;
    k7 := k6 + 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];
      A4 := A[k4];  B4 := B[k4];
      A5 := A[k5];  B5 := B[k5];
      A6 := A[k6];  B6 := B[k6];
      A7 := A[k7];  B7 := B[k7]
    end
    else
    begin
      A1 := A[k1] × c4 − B[k1] × s4;
      B1 := A[k1] × s4 + B[k1] × c4;
      A2 := A[k2] × c2 − B[k2] × s2;
      B2 := A[k2] × s2 + B[k2] × c2;
      A3 := A[k3] × c6 − B[k3] × s6;
      B3 := A[k3] × s6 + B[k3] × c6;
      A4 := A[k4] × c1 − B[k4] × s1;
      B4 := A[k4] × s1 + B[k4] × c1;
      A5 := A[k5] × c5 − B[k5] × s5;
      B5 := A[k5] × s5 + B[k5] × c5;
      A6 := A[k6] × c3 − B[k6] × s3;
      B6 := A[k6] × s3 + B[k6] × c3;
      A7 := A[k7] × c7 − B[k7] × s7;
      B7 := A[k7] × s7 + B[k7] × c7
    end;
    x0 := A0 + A1 + A2 + A3;  y0 := B0 + B1 + B2 + B3;
    x1 := A0 − A1 − B2 + B3;  y1 := B0 − B1 + A2 − A3;
    x2 := A0 + A1 − A2 − A3;  y2 := B0 + B1 − B2 − B3;
    x3 := A0 − A1 + B2 − B3;  y3 := B0 − B1 − A2 + A3;
    x4 := A4 + A5 + A6 + A7;  y4 := B4 + B5 + B6 + B7;
    x5 := (A4−A5−B6+B7) × c45;
    y5 := (B4−B5+A6−A7) × c45;
    x6 := A4 + A5 − A6 − A7;  y6 := B4 + B5 − B6 − B7;
    x7 := (A4−A5+B6−B7) × c45;
    y7 := (B4−B5−A6+A7) × c45;
    A[k0] := x0 + x4;  B[k0] := y0 + y4;
    A[k1] := x1 + x5 − y5;  B[k1] := y1 + x5 + y5;
    A[k2] := x2 − y6;  B[k2] := y2 + x6;
    A[k3] := x3 − x7 − y7;  B[k3] := y3 + x7 − y7;
    A[k4] := x0 − x4;  B[k4] := y0 − y4;
    A[k5] := x1 − x5 + y5;  B[k5] := y1 − x5 − y5;
    A[k6] := x2 + y6;  B[k6] := y2 − x6;
    A[k7] := x3 + x7 + y7;  B[k7] := y3 − x7 + y7;
    k0 := k7 + span;  if k0 < n then go to Ld;
    k0 := k0 − n;  if k0 < k then go to Ld;
    comment Increment the sine and cosine values;
    if k0 ≠ span then
    begin
      c2 := c1 − (dc×c1+ds×s1);
      s1 := (ds×c1−dc×s1) + s1;
      comment The following three statements compensate
        for truncation error. If rounded arithmetic is used,
        substitute c1 := c2;
      c1 := 1.5−0.5 × (c2↑2+s1↑2);
      s1 := c1 × s1;  c1 := c1 × c2;
      c2 := c1↑2 − s1↑2;  s2 := 2.0 × c1 × s1;
      c3 := c1 × c2 − s1 × s2;  s3 := s1 × c2 + c1 × s2;
      c4 := c2↑2 − s2↑2;  s4 := 2.0 × c2 × s2;
      c5 := c1 × c4 − s1 × s4;  s5 := s1 × c4 + c1 × s4;
      c6 := c3↑2 − s3↑2;  s6 := 2.0 × c3 × s3;
      c7 := c1 × c6 − s1 × s6;  s7 := s1 × c6 + c1 × s6;
      k := k + ks;  go to Ld
    end;
    span := 8 × span
  end
end REVFFT8
```

ALGORITHM 346
F-TEST PROBABILITIES [S14]

John Morris (Recd. 10 Apr. 1968, 12 Sept. 1968, and 6 Nov. 1968)
Computer Institute for Social Science Research, Michigan State University, East Lansing, MI 48823

KEY WORDS AND PHRASES: F-test, Snedecor F-statistic, Fisher test, distribution function
CR CATEGORIES: 5.5

```
procedure Ftest (f, df1, df2, maxn, prob, gauss, error);
  value f, df1, df2, maxn;  real f, prob;  integer df1, df2, maxn;
  real procedure gauss;  label error;
comment This procedure gives the probability that F will be
  greater than the value of f where
```

$$f = \sigma_1^2/\sigma_2^2,$$

σ_1^2 is the variance of the sample with size N_1, σ_2^2 is the variance of the sample with size N_2, $df1 = N_1 - 1$, $df2 = N_2 - 1$, and F is the Snedecor-Fisher statistic as defined and tabled by Snedecor [4].

The present algorithm computes a value which is directly related to that of Algorithm 322, such that *prob* = 1 − *Fisher*. A number of test runs on various computers suggest that *Ftest* may be considerably faster than *Fisher*.

An approximation is included to limit execution time when sample size is large. It should be used when register overflow would otherwise result, and the appropriate value for *maxn* will therefore depend upon the specific implementation. When *maxn* = 500 the approximation appears to give three-digit accuracy. The **real procedure** *gauss* computes the area under the left-hand portion of the normal curve. Algorithm 209 [3] may be used for this purpose. If $f < 0$ or if $df1 < 1$ or if $df2 < 1$ then exit to the label *error* occurs.

National Bureau of Standards formulas 26.6.4, 26.6.5, and 26.6.8 are used for computation of the statistic, and 26.6.15 is used for the approximation [2].

Thanks to Mary E. Rafter for extensive testing of this procedure and to the referee for a number of suggestions.

REFERENCES:

1. DORRER, EGON. Algorithm 322, F-Distribution. *Comm. ACM 11* (Feb. 1968), 116–117.
2. *Handbook of Mathematical Functions.* National Bureau of Standards, Appl. Math. Ser. Vol., 55, Washington, D.C., 1965, pp. 946–947.
3. IBBETSON, D. Algorithm 209, Gauss. *Comm. ACM 6* (Oct. 1963), 616.
4. SNEDECOR, GEORGE W. *Statistical Methods.* Iowa State U. Press, Ames, Iowa, 1956, pp. 244–250;

```
begin
  if df1 < 1 ∨ df2 < 1 ∨ f < 0.0 then go to error;
  if f = 0.0 then prob := 1.0
  else
  begin
    real f1, f2, x, ft, vp;
    f1 := df1;  f2 := df2;  ft := 0.0;
    x := f2/(f2+f1×f);  vp := f1 + f2 − 2.0;
    if 2 × (df1÷2) = df1 ∧ df1 ≦ maxn then
    begin
      real xx;  xx := 1.0 − x;
      for f1 := f1 − 2.0 step − 2.0 until 1.0 do
      begin
        vp := vp − 2.0;
        ft := xx × vp/f1 × (1.0+ft)
      end;
      ft := x ↑ (0.5×f2) × (1.0+ft)
    end
    else if 2 × (df2 ÷ 2) = df2 ∧ df2 ≦ maxn then
    begin
      for f2 := f2 − 2.0 step − 2.0 until 1.0 do
      begin
        vp := vp − 2.0;
        ft := x × vp/f2 × (1.0+ft)
      end;
      ft := 1.0 − (1.0−x) ↑ (0.5×f1) × (1.0+ft)
    end
    else if df1 + df2 ≦ maxn then
    begin
      real theta, sth, cth, sts, cts, a, b, xi, gamma;
      theta := arctan(sqrt(f1×f/f2));
      sth := sin(theta);  cth := cos(theta);
      sts := sth ↑ 2;  cts: = cth ↑ 2;
      a := b := 0.0;
      if df2 > 1 then
      begin
        for f2 := f2 − 2.0 step − 2.0 until 2.0 do
          a := cts × (f2−1.0)/f2 × (1.0+a);
        a := sth × cth × (1.0+a)
      end;
      a := theta + a;
      if df1 > 1 then
      begin
        for f1 := f1 − 2.0 step − 2.0 until 2.0 do
        begin
          vp := vp − 2.0;
          b := sts × vp/f1 × (1.0+b)
        end;
        gamma := 1.0;  f2 := 0.5 × df2;
        for xi := 1.0 step 1.0 until f2 do
          gamma := xi × gamma/(xi−0.5);
        b := gamma × sth × cth ↑ df2 × (1.0+b)
      end;
      ft := 1.0 + 0.636619772368 × (b−a);
      comment 0.63661977236758134307553 51 ··· = 2.0/π;
    end
    else
    begin
      real cbrf;
      f1 := 2.0/(9.0 × f1);  f2 := 2.0/(9.0×f2);
      cbrf := f ↑ 0.333333333333;
      ft := gauss(−((1.0−f2)×cbrf+f1−1.0)/
        sqrt(f2×cbrf ↑ 2+f1))
    end;
    prob := if ft < 0.0 then 0.0 else ft
  end
end Ftest
```

ALGORITHM 347
AN EFFICIENT ALGORITHM FOR SORTING WITH MINIMAL STORAGE [M1]

RICHARD C. SINGLETON* (Recd. 17 Sept. 1968)
Mathematical Statistics and Operations Research Department, Stanford Research Institute, Menlo Park, CA 94025

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

KEY WORDS AND PHRASES: sorting, minimal storage sorting, digital computer sorting
CR CATEGORIES: 5.31

```
procedure SORT(A, i, j);
  value i, j;  integer i, j;
  array A;
comment This procedure sorts the elements of array A into
  ascending order, so that
```

$$A[k] \leq A[k+1], \qquad k = i, i+1, \cdots, j-1.$$

The method used is similar to *QUICKERSORT* by R. S. Scowen [5], which in turn is similar to an algorithm given by Hibbard [2, 3] and to Hoare's *QUICKSORT* [4]. *QUICKERSORT* is used as a standard, as it was shown in a recent comparison to be the fastest among four ACM algorithms tested [1]. On the Burroughs B5500 computer, the present algorithm is about 25 percent faster than *QUICKERSORT* when tested on random uniform numbers (see Table I) and about 40 percent faster on numbers in natural order $(1, 2, \cdots, n)$, in reverse order $(n, n-1, \cdots, 1)$, and sorted by halves $(2, 4, \cdots, n, 1, 3, \cdots, n-1)$. *QUICKERSORT* is slow in sorting data with numerous "tied" observations, a problem that can be corrected by changing the code to exchange elements $a[k] \geq t$ in the lower segment with elements $a[q] \leq t$ in the upper segment. This change gives a better split of the original segment, which more than compensates for the additional interchanges.

In the earlier algorithms, an element with value t was selected from the array. Then the array was split into a lower segment with all values less than or equal to t and an upper segment with all values greater than or equal to t, separated by a third segment of length one and value t. The method was then applied

TABLE I. SORTING TIMES IN SECONDS FOR *SORT* AND *QUICKERSORT*, ON THE BURROUGHS B5500 COMPUTER—AVERAGE OF FIVE TRIALS

Original order and number of items	*Algorithm* *SORT*	*QUICKERSORT*
Random uniform:		
500	0.48	0.63
1000	1.02	1.40
Natural order:		
500	0.29	0.48
1000	0.62	1.00
Reverse order:		
500	0.30	0.51
1000	0.63	1.08
Sorted by halves:		
500	0.73	1.15
1000	1.72	2.89
Constant value:		
500	0.43	10.60
1000	0.97	41.65

recursively to the lower and upper segments, continuing until all segments were of length one and the data were sorted. The present method differs slightly—the middle segment is usually missing—since the comparison element with value t is not removed from the array while splitting. A more important difference is that the median of the values of $A[i]$, $A[(i+j)\div 2]$, and $A[j]$ is used for t, yielding a better estimate of the median value for the segment than the single element used in the earlier algorithms. Then while searching for a pair of elements to exchange, the previously sorted data (initially, $A[i] \leq t \leq A[j]$) are used to bound the search, and the index values are compared only when an exchange is about to be made. This leads to a small amount of overshoot in the search, adding to the fixed cost of splitting a segment but lowering the variable cost. The longest segment remaining after splitting a segment of n has length less than or equal to $n - 2$, rather than $n - 1$ as in *QUICKERSORT*.

For efficiency, the upper and lower segments after splitting should be of nearly equal length. Thus t should be close to the median of the data in the segment to be split. For good statistical properties, the median estimate should be based on an odd number of observations. Three gives an improvement over one and the extra effort involved in using five or more observations may be worthwhile on long segments, particularly in the early stages of a sort.

Hibbard [3] suggests using an alternative method, such as Shell's [6], to complete the sort on short sequences. An experimental investigation of this idea using the splitting algorithm adopted here showed no improvement in going beyond the final stage of Shell's algorithm, i.e. the familiar "sinking" method of sorting by interchange of adjacent pairs. The minimum time was obtained by sorting sequences of 11 or fewer items by this method. Again the number of comparisons is reduced by using the data themselves to bound the downward search. This requires

$$A[i-1] \leq A[k], \qquad i \leq k \leq j.$$

Thus the initial segment cannot be sorted in this way. The initial segment is treated as a special case and sorted by the splitting algorithm. Because of this feature, the present algorithm lacks the pure recursive structure of the earlier algorithms.

For n elements to be sorted, where $2^k \leq n < 2^{k+1}$, a maximum of k elements each are needed in arrays IL and IU. On the B5500 computer, single-dimensional arrays have a maximum length of 1023. Thus the array bounds [0:8] suffice.

This algorithm was developed as a FORTRAN subroutine, then translated to ALGOL. The original FORTRAN version follows:

```
      SUBROUTINE SORT(A,II,JJ)
C  SORTS ARRAY A INTO INCREASING ORDER, FROM A(II) TO A(JJ)
C  ORDERING IS BY INTEGER SUBTRACTION, THUS FLOATING POINT
C    NUMBERS MUST BE IN NORMALIZED FORM.
C  ARRAYS IU(K) AND IL(K) PERMIT SORTING UP TO 2**(K+1)-1 ELEMENTS
      DIMENSION A(1),IU(16),IL(16)
      INTEGER A,T,TT
      M=1
      I=II
      J=JJ
    5 IF(I .GE. J) GO TO 70
   10 K=I
      IJ=(J+I)/2
      T=A(IJ)
      IF(A(I) .LE. T) GO TO 20
      A(IJ)=A(I)
      A(I)=T
      T=A(IJ)
   20 L=J
      IF(A(J) .GE. T) GO TO 40
      A(IJ)=A(J)
      A(J)=T
      T=A(IJ)
      IF(A(I) .LE. T) GO TO 40
      A(IJ)=A(I)
      A(I)=T
      T=A(IJ)
      GO TO 40
   30 A(L)=A(K)
      A(K)=TT
   40 L=L-1
      IF(A(L) .GT. T) GO TO 40
      TT=A(L)
   50 K=K+1
      IF(A(K) .LT. T) GO TO 50
      IF(K .LE. L) GO TO 30
      IF(L-I .LE. J-K) GO TO 60
      IL(M)=I
      IU(M)=L
      I=K
      M=M+1
      GO TO 80
   60 IL(M)=K
      IU(M)=J
      J=L
      M=M+1
      GO TO 80
   70 M=M-1
      IF(M .EQ. 0) RETURN
      I=IL(M)
      J=IU(M)
   80 IF(J-I .GE. 11) GO TO 10
      IF(I .EQ. II) GO TO 5
      I=I-1
   90 I=I+1
      IF(I .EQ. J) GO TO 70
      T=A(I+1)
      IF(A(I) .LE. T) GO TO 90
      K=I
  100 A(K+1)=A(K)
      K=K-1
      IF(T .LT. A(K)) GO TO 100
      A(K+1)=T
      GO TO 90
      END
```

This FORTRAN subroutine was tested on a CDC 6400 computer. For random uniform numbers, sorting times divided by $n \log_2 n$ were nearly constant at 20.2×10^{-6} for $100 \leq n \leq 10{,}000$, with a time of 0.202 seconds for 1000 items. This subroutine was also hand-compiled for the same computer to produce a more efficient machine code. In this version the constant of proportionality was 5.2×10^{-6}, with a time of 0.052 seconds for 1000 items. In both cases, integer comparisons were used to order normalized floating-point numbers.

REFERENCES:

1. BLAIR, CHARLES R. Certification of algorithm 271. *Comm. ACM 9* (May 1966), 354.
2. HIBBARD, THOMAS N. Some combinatorial properties of certain trees with applications to searching and sorting. *J. ACM 9* (Jan. 1962), 13–28.
3. HIBBARD, THOMAS N. An empirical study of minimal storage sorting. *Comm. ACM 6* (May 1963), 206–213.
4. HOARE, C. A. R. Algorithms 63, Partition, and 64, Quicksort. *Comm. ACM 4* (July 1961), 321.
5. SCOWEN, R. S. Algorithm 271, Quickersort. *Comm. ACM 8* (Nov. 1965), 669.
6. SHELL, D. L. A high speed sorting procedure. *Comm. ACM 2* (July 1959), 30–32;

```
begin
  real t, tt;
  integer ii, ij, k, L, m;
  integer array IL, IU[0:8];
  m := 0;  ii := i;  go to L4;
L1:  ij := (i+j) ÷ 2;  t := A[ij];  k := i;  L := j;
  if A[i] > t then
    begin A[ij] := A[i];  A[i] := t;  t := A[ij] end;
  if A[j] < t then
  begin
    A[ij] := A[j];  A[j] := t;  t := A[ij];
    if A[i] > t then
      begin A[ij] := A[i];  A[i] := t;  t := A[ij] end
  end;
L2:  L := L − 1;
  if A[L] > t then go to L2;
    tt := A[L];
L3:  k := k + 1;
  if A[k] < t then go to L3;
  if k ≤ L then
    begin A[L] := A[k];  A[k] := tt;  go to L2 end;
  if L − i > j − k then
    begin IL[m] := i;  IU[m] := L;  i := k end
  else
    begin IL[m] := k;  IU[m] := j;  j := L end;
  m := m + 1;
L4:  if j − i > 10 then go to L1;
    if i = ii then
    begin if i < j then go to L1 end;
    for i := i + 1 step 1 until j do
    begin
    t := A[i];  k := i − 1;
    if A[k] > t then
    begin
L5:  A[k+1] := A[k];  k := k − 1;
    if A[k] > t then go to L5;
    A[k+1] := t
    end
  end;
  m := m − 1;  if m ≥ 0 then
    begin i := IL[m];  j := IU[m];  go to L4 end
  end SORT
```

REMARK ON ALGORITHM 329 [G6]
DISTRIBUTION OF INDISTINGUISHABLE OBJECTS INTO DISTINGUISHABLE SLOTS [Robert R. Fenichel, *Comm. ACM 11* (June 1968), 430]

M. Gray (Recd. 20 Sept. 1968)
Computing Science Department, University of Adelaide, South Australia

As the procedure stands it is incorrect. Preceding

end skip 99,189,198, etc.

the following statement should be inserted:

if $q[k] \neq 0$ **then** *LeftmostZero* := $k + 1$

Thus the compound statement becomes:

```
begin
  LeftmostZero := LeftmostZero −1;
  q[k] := q[LeftmostZero] − 1;
  q[LeftmostZero] := 0;
  q[LeftmostZero−1] := q[LeftmostZero−1] + 1;
  if q[k] ≠ 0 then LeftmostZero := k + 1
end skip 99, 189, 198, etc.
```

REMARK ON ALGORITH 339 [C6]
AN ALGOL PROCEDURE FOR THE FAST FOURIER TRANSFORM WITH ARBITRARY FACTORS
[Richard C. Singleton, *Comm. ACM 11* (Nov. 1968), 776]

Richard C. Singleton (Recd. 27 Nov. 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

On page 778, column 2, the 7th and 6th lines from the bottom should be corrected to read:

```
LJ: jj := C[i−2] + jj;  if jj ≧ C[i−1] then
      begin i := i − 1;  jj := jj − C[i];  go to LJ end;
```

On page 779, column 1, the 9th and 8th lines from the bottom should be corrected to read:

```
LX: jj := D[k+1] + jj;  if jj ≧ D[k] then
      begin jj := jj − D[k];  k := k + 1;  go to LX end;
```

In both cases *jj* was originally used as the controlled variable of a for clause and thus was undefined after exit; the corrections preserve the value of *jj* for later use.

If the user prefers to compute constants with library functions, line 5 in column 2 on page 777 may be replaced by:

rad := 8.0 × *arctan*(1.0); *c30* := *sqrt*(0.75);

Algorithms 338 [*Comm. ACM 11* (Nov. 1968), 773] and 339 were punched from the printed page and tested on the CDC 6400 Algol compiler. After changing a colon to a semicolon at the end of line 37 in column 2 on page 775, the test results agreed with those obtained earlier with this compiler.

When computing a single-variate Fourier transform of real data, procedure *REALTRAN* may be used with procedure *FFT* (Algorithm 339) to reduce computing time. Two versions of *REALTRAN* have been given (Algorithms 338 and 345 [*Comm. ACM 12* (Mar. 1969), 179–184]); the first version is the faster of the two, but the second should be used if arithmetic results for real quantities are truncated rather than rounded.

In describing the evaluation of a real Fourier series, in the middle of column 2 on page 776, the necessary steps of reversing the signs of the *B* array values both before and after calling *FFT* were omitted. The correct steps, including scaling, are as follows:

```
REALTRAN(A B, n, true);
for j := n − 1 step −1 until 0 do B[j] := −B[j];
FFT(A, B, n, n, n);
for j := n − 1 step −1 until 0 do
  begin A[j] := 0.5 × A[j];  B[j] := −0.5 × B[j] end;
```

The policy concerning the contributions of algorithms to *Communications of the ACM* appears, most recently, in the January 1969 issue, page 39. A contribution should be in the form of an algorithm, a certification, or a remark. An algorithm must normally be written in the ALGOL 60 Reference Language or in USASI Standard FORTRAN or Basic FORTRAN.