Fast Fourier transform free from tears

A. M. Macnaghten and C. A. R. Hoare

Department of Computer Science, The Queen's University Belfast, Belfast BT7 1NN, Northern Ireland

Many descriptions of Fast Fourier Transform exist in the literature. Several of these appeal to matrix concepts, such as Kronecker multiplication. This paper shows the essential simplicity of the algorithm and the reasoning behind it. However it deals only with the case when the number of points is an exact power of 2.

(Received October 1974)

1. Introduction—Basic version

Let Y be a vector of real or complex numbers with N elements, numbered from 0 to $N-1$. The N-point Fourier transform of Y is a vector F with the same subscript bounds defined by the formula

$$F_P^N(Y) = \frac{2}{N}\sum_{J=0}^{N-1}((\alpha_N)\uparrow(P*J))*Y_J$$

$$(\text{for } P = 0 \text{ to } N-1) \qquad (1)$$

where $\alpha_N = \exp(\sqrt{-1}*2\pi/N)$ is the principal complex Nth root of unity. The values of F can be computed using a simple nested loop.

This 'program' (as well as all the subsequent programs included in this paper) is written in a form of publication ALGOL [Naur], in which we assume the availability of complex variables and complex arithmetic, as in FORTRAN IV. Since the purpose of this paper is to convey understanding to the reader rather than to the computer, the use of informal mathematical notation to express a program is quite helpful. If the reader is worried by the informality he may like to recode some of the examples in his favourite programming language. The program is coded as a procedure with three parameters Y (the complex array containing the input vector), F (the complex array in which the Fourier coefficients are to be stored) and N (the size of these two arrays), and is listed in **Table 1.**

2. First improvement—pre-calculation of factors

The program described in the previous section is clear in its operation, but it contains several inefficiencies which will be removed in the version described in this section.

Since α_N is an Nth root of unity it follows that $(\alpha_N)\uparrow N = 1$

$$\therefore \quad ((\alpha_N)\uparrow N)\uparrow K = 1 \text{ where } K \text{ is an integer}$$

$$\therefore \quad (\alpha_N)\uparrow N*K = 1$$

$$\therefore \quad (\alpha_N)\uparrow(N*K + R) = (a_N)\uparrow R$$

$$\therefore \quad (\alpha_N)\uparrow I = (\alpha_N)\uparrow (I \textbf{ mod } N)$$

Table 1

```
procedure Fourier1(Y, F, N);
 value N; complex array Y, F;
 integer N;
begin
 integer J, P; complex AlphaN;
 AlphaN := exp(√-1*2π/N);
 for P := 0 step 1 until N - 1 do
 begin
  F[P] := 0;
  for J := 0 step 1 until N - 1 do
   F[P] := F[P]+(AlphaN↑(P*J))*Y[J];
   F[P] := F[P]*2/N
 end
end;
```

where I **mod** N is the remainder after dividing I by N. This remainder lies in the range 0 to $N-1$ and so there are only N possible values of $(\alpha_N)\uparrow I$. These may be precomputed and stored in a complex array FACTOR (whose subscript range is 0 to $N-1$) by the following loop:

```
X := 1;
for P := 0 step 1 until N - 1 do
begin
  FACTOR[P] := X; X := X*AlphaN
end;
```

Now within the program of the previous section we may replace $AlphaN\uparrow(P*J)$ with $FACTOR[(P*J)$ **mod** $N]$.

A further improvement may be obtained if we eliminate the integer multiplication $P*J$, using a method adopted by many optimising translators. Since, within the inner loop, J varies in steps of 1 we may introduce a new integer variable C, which is given an initial value 0, and to which P is added on each execution of the loop. A conditional subtraction of N can replace the **mod** operation giving a further gain in speed.

These two optimisations are included in the version of the program in **Table 2.**

This program describes essentially the method used to carry out Fourier transforms until 1965. Its chief disadvantage is that the innermost loop is iterated N^2 times, and each iteration involves a complex multiplication. Thus for large N (say $N \simeq 1{,}000$), the calculation was very expensive. In 1965 a new method was discovered (Cooley and Tukey, 1965) which in many cases reduces the number of complex multiplications to approximately $N*\log_2(N)$, thereby giving a factor of 100 saving in time when $N \simeq 1{,}000$. It is the purpose of this paper to explain the reasoning behind this faster algorithm. It uses the method of stepwise refinement (Wirth, 1971); the first step of the refinement has already taken place.

3. Separate treatment of even and odd Fourier coefficients

In this section we will show how the number of multiplications may be reduced by nearly a half where N is even. This is done by using different formulae to compute the even numbered and the odd numbered elements of F. The two formulae are

1. For *even* coefficients

$$F_{2R}^N(Y) = \frac{2}{M}\sum_{J=0}^{M-1}((\alpha_M\uparrow(R*J))*SUM[J])$$

$$(\text{for } R = 0 \text{ to } M-1) \qquad (2)$$

where $M = N \div 2$

2. For *odd* coefficients

$$F_{2R+1}^N(Y) = \frac{2}{M}\sum_{J=0}^{M-1}((\alpha_M\uparrow(R*J))*DIFF[J])$$

$$(\text{for } R = 0 \text{ to } M-1) \qquad (3)$$

Table 2

```
procedure Fourier2(Y, F, N);
 value  N; complex array Y, F;
 integer N;
begin
 complex array FACTOR[0:N − 1];
 integer J, P, C; complex X, AlphaN;
 X := 1; AlphaN := exp(√−1*2π/N);
 for P := 0 step 1 until N − 1 do
 begin
  FACTOR[P] := X; X := X*AlphaN
 end;
 for P := 0 step 1 until N − 1 do
 begin
  X := 0; C := 0;
  for J := 0 step 1 until N − 1 do
  begin
   X := X + FACTOR[C]*Y[J];
   C := C + P;
   if C > N − 1 then C := C − N
  end;
  F[P] := X*2/N
 end
end;
```

SUM and *DIFF* are defined by the two equations

$$SUM[J] = (Y[J] + Y[J + M])/2 \qquad (4)$$

and

$$DIFF[J] = ((Y[J] - Y[J + M])/2)*\alpha_N\uparrow J \qquad (5)$$

$$(\text{for } J = 0 \text{ to } M - 1) \ .$$

A proof of these formulae is given in the appendix.

We therefore introduce four new arrays *SUM*, *DIFF*, *EVENF* and *ODDF*, with subscript ranges 0 to $M - 1$. *SUM* and *DIFF* are computed according to formulae (4) and (5). *EVENF* and *ODDF* are then computed using formulae (2) and (3) and will contain, respectively, the even numbered and odd numbered Fourier coefficients. Finally these values must be exactly interleaved to give the required array *F*.

The formulae (2) and (3) are identical to the original formula (1), using *SUM* and *DIFF* in place of *Y* and using *M* in place of *N*, so we may use the already coded procedure *Fourier*2, to compute *EVENF* and *ODDF*.

We have renamed the procedure *Even Fourier*, since it is only applicable where *N* is even—see **Table 3.**

The speed advantage obtained by using *Even Fourier* results from the fact that the basic procedure *Fourier*2 performs N^2 complex multiplications (apart from the setting up of the array *FACTOR*). Thus its execution time is approximately proportional to the *square* of its parameter *N*. By replacing one call of Fourier (parameter *N*) by two calls (each with parameter *N*/2) we clearly reduce the number of complex multiplications from N^2 to $2 \times (N/2)^2$ i.e. from N^2 to $N^2/2$. Of course we will incur, as a penalty, the setting up of the arrays *SUM* and *DIFF* (this latter array needs *N*/2 complex multiplications) and the final interleaving operation; but it is clear that where *N* is large the saving of the complex multiplication load from N^2 to $N^2/2$ will have a dominant effect on the overall execution time.

4. Further improvements possible where *N* is a power of 2—introduction of recursion

In the previous section it was shown that a significant gain can be achieved when *N* is even; and it is fairly obvious that a similar gain could be made if *N*/2 is also even, and if *N*/4 is also even; and the greatest gain of all should be achievable if *N* is an exact power of two. Although this seems to be a special case, in fact it can often be achieved, since the array usually represents a set of evenly spaced samples from a continuous observed function or waveform, and it is easy to select an interval and sample size which is an exact power of two (say 1,024).

To take advantage of this case, the main change that is necessary in the body of the procedure *Even Fourier* is to replace the two calls on Fourier by recursive calls on *Even Fourier* itself! But there is one case which *Even Fourier* can never deal with, namely the case when $N = 1$. However, in this case the Fourier formula (1) simplifies to

$$F_0^1(Y) = 2*Y_0$$

and this case is easily treated. Since the procedure works only when *N* is an exact binary power, we will use a new name: *Bin Fourier* (see **Table 4**). This is the basic idea underlying the fast Fourier transform. The remainder of this paper merely describes refinements of this algorithm.

5. Reintroduction of the pre calculation of FACTOR

In the program of the previous section, the precomputation of the array *FACTOR* has entirely disappeared, and been replaced by the comparatively expensive recalculation of the powers of α_N. In this section we will see how the precomputation method can be reintroduced into the recursive program.

The optimisation is based on the fact that

$$\alpha_{N\div 2} = \alpha_N\uparrow 2$$

and consequently

$$\alpha_{N\div 2}\uparrow I = \alpha_N\uparrow(2*I)$$

$$\text{for } I = 0 \text{ to } N \div 2 \ .$$

Similarly if *D* (like *N*) is a power of 2 then $\alpha_{N\div D}\uparrow I = \alpha_N\uparrow(D*I)$ (for $I = 0$ to $N \div D$). Each recursive call of *Bin Fourier*1 uses the powers of α_M where $M = N \div 2$ and *N* is the parameter of *Bin Fourier*1 at the next outer level of recursion.

Let us write *NN* for the size of the array on which the given call of *Bin Fourier*1 is operating, and let *N* be the size of the original array, and let $D = N \div NN$—(note that *D* is a measure of the depth of recursion). Thus *Bin Fourier*1 uses the powers of α_{NN} or $\alpha_{N\div D}$, which are a subset of the powers of α_N, being of the form $\alpha_N\uparrow(D*I)$ (for $I = 0$ to $N \div D$).

The integers of the form $D*I$ (where *I* goes from 0 to *NN*)

Table 3

```
procedure Even Fourier(Y, F, N);
 value  N; complex array Y, F;
 integer N;
begin
 complex array SUM, DIFF, EVENF, ODDF[0:N ÷ 2 − 1];
 integer M, J; complex X, AlphaN;
 M := N ÷ 2; AlphaN := exp(√−1*2π/N);
 X := 1;
 for J := 0 step 1 until M − 1 do
 begin
  SUM[J] := (Y[J] + Y[J + M])/2;
  DIFF[J] := (Y[J] − Y[J + M])*X/2;
  X := X*AlphaN
 end;
 Fourier2(SUM, EVENF, M);
 Fourier2(DIFF, ODDF, M);
 for J := 0 step 1 until M − 1 do
 begin
  F[2*J] := EVENF[J];
  F[2*J + 1] := ODDF[J]
 end
end;
```

represent the subscripts of the required elements of *FACTOR*. To access these elements we may even avoid the integer multiplication *D*I*, by using the counting technique described in Section 2.

We must therefore set up the array *FACTOR*, in an *outer* procedure *Bin Fourier*2 (with parameters *Y*, *F* and *N*) before calling the recursive procedure *sub Bin Fourier*2 (with parameters *YY*, *FF* and *NN*) which is declared within it (see **Table 5**).

6. Storage saving by computation in situ

So far the refinements introduced have been entirely concerned with improvements in speed. Locally declared arrays have been employed within the recursively-used procedure with an almost reckless disregard of space economy: four such complex arrays, each with $NN \div 2$ elements, are declared within the procedure. This section explores various techniques for reducing the storage required.

First, it can be seen that once the *SUM* and *DIFF* arrays have been set up within a particular call of *sub Bin Fourier*2, no further reference is made to the elements of *YY* within this call. It may also be verified that these original array elements of *YY* are not referred to in any relevant recursive call, i.e. at an inner level of recursion, included within the execution of the original call.

Thus we may use the first half of the *YY* array in place of *SUM* and the second half (i.e. elements $(N \div 2)$ to $(N - 1)$ inclusive) in place of *DIFF*. The parameter list of *sub Bin Fourier* will now have to include an integer parameter *START*, indicating the subscript of the first element to be processed. Since the same overall array *Y* may be used for all calls of *sub Bin Fourier* we may remove *YY* from the parameter list of *sub Bin Fourier*.

A second consequence of the fact that *YY* is not referred to within *sub Bin Fourier* after the two recursive calls, is that *Y* can also be used to store the values previously held in *EVENF* and *ODDF* (in the first and second halves respectively) and thus finally to store the result *F*. So *F* and *FF* can be removed from the parameter lists of *Bin Fourier* and *sub Bin Fourier* respectively; and we redefine the effect of a call of *Bin Fourier* so that it now replaces the elements of *Y* with the elements of its Fourier transform.

This last step raises one difficulty in that we must now perform in situ an interleaving operation on the elements of *Y* (or, in the case of recursive calls, some of the elements of *Y*). Ingenious and fairly fast solutions of this problem exist, but, to save unnecessarily complicating this version of the program, and because in the next section we explore a different and more efficient approach to the interleaving stage, we have excluded the coding of in situ interleaving from the version in **Table 6.** This then incorporates an outer procedure *Bin Fourier*3 (with two parameters *Y* and *N*) which includes a declaration of *sub Bin Fourier*3, a section of code to set up the array *FACTOR* and a single call of *sub Bin Fourier*3. This latter procedure will have two parameters *START* and *NN* and will convert the elements of *Y* numbered from *START* to *START* + *NN* − 1 into the Fourier transform of this subarray.

Table 4

```
procedure Bin Fourier1(Y, F, N);
 value N; complex array Y, F; integer N;
begin
 complex array SUM, DIFF, EVENF,
  ODDF[0:N ÷ 2 − 1];
 integer M, J;
 complex X, AlphaN;
 if N = 1 then F[0] := 2*Y[0]
 else
 begin
  M := N ÷ 2; X := 1; AlphaN := exp(√−1*2π/N);
  for J := 0 step 1 until M − 1 do
  begin
   SUM[J] := (Y[J] + Y[J + M])/2;
   DIFF[J] := (Y[J] − Y[J + M])*X/2;
   X := X*AlphaN
  end;
  Bin Fourier1(SUM, EVENF, M);
  Bin Fourier1(DIFF, ODDF, M);
  for J := 0 step 1 until M − 1 do
  begin
   F[2*J] := EVENF[J];
   F[2*J + 1] := ODDF[J]
  end
 end
end;
```

Table 5

```
procedure Bin Fourier2(Y, F, N);
 value N; complex array Y, F; integer N;
begin
 integer J; complex array FACTOR[0:(N ÷ 2) − 1];
 complex X, AlphaN;
 procedure sub Bin Fourier2(YY, FF, NN);
  value YY, NN; complex array YY, FF;
  integer NN;
 begin complex array SUM, DIFF, EVENF, ODDF
   [0:(NN ÷ 2) − 1];
  integer JJ, MM, C, D;
  if NN = 1 then FF[0] := 2*YY[0]
  else
  begin
   C := 0; D := N ÷ NN; MM := NN ÷ 2;
   comment C is location of required factor—D is separation;
   for JJ := 0 step 1 until MM − 1 do
   begin
    SUM[JJ] := (YY[JJ] ÷ YY[JJ + MM])/2;
    DIFF[JJ] := (YY[JJ] − YY[JJ + MM])*
     FACTOR[C]/2;
    C := C + D
   end;
   sub Bin Fourier2(SUM, EVENF, MM);
   sub Bin Fourier2(DIFF, ODDF, MM);
   for JJ := 0 step 1 until MM − 1 do
   begin
    FF[2*JJ] := EVENF[JJ];
    FF[2*JJ ÷ 1] ;= ODDF[JJ]
   end
  end
 end;
 X := 1; AlphaN := exp(√−1*2π/N);
 for J := 0 step 1 until (N ÷ 2) − 1 do
 begin
  FACTOR[J] := X; X := X*AlphaN
 end;
 sub Bin Fourier2(Y, F, N)
end;
```

7. Final optimisation involving postponed interleaving of elements

We have already mentioned the problem of in situ interleaving of *Y*. In this section we explore the exact effect of interleaving at each stage of the complete process and devise a method by which the combined interleavings may be achieved in a single operation carried out at the end.

This postponed interleaving depends on the fact that once a particular location has a value placed in it as a result of an interleave, that value is not referred to except in possibly further interleaving operations at outer levels of recursion.

To investigate the rule for determining the final destination of the element which starts, say at position I, within the array, we will consider the effect of each interleaving operation on this element. The sequence of recursive calls will result in the interleaving of elements first (at the deepest level of recursion) in groups of two (no effect), then in groups of four, then in groups of eight, 16, etc. until the whole array is subjected to the interleaving process.

Table 6

```
procedure Bin Fourier3(Y, N);
 value N; complex array Y; integer N;
begin
 integer J; complex array FACTOR[0:N ÷ 2 − 1];
 complex X, AlphaN;
 procedure sub Bin Fourier3(START, NN);
  value START, NN; integer START, NN;
 begin
  integer MM, JJ, C, D; complex XX;
  if NN = 1 then Y[START] := 2*Y[START]
  else
  begin
   C := 0; D := N ÷ NN; MM := NN ÷ 2;
   for JJ := START step 1 until START + MM − 1 do
   begin
    XX := Y[JJ];
    Y[JJ] := (XX + Y[JJ + MM])/2;
    Y[JJ + MM] := (XX − Y[JJ + MM])*
      FACTOR[C]/2; C := C + D
   end;
   sub Bin  Fourier3(START, MM);
   sub Bin  Fourier3(START + MM, MM);
   Comment code for interleaving
           Y[START] → Y[START + MM − 1]
           with Y[START+MM] → Y[START+NN−1]
           in situ should be included here but has been
           omitted for reasons given;
  end
 end;
 X := 1; AlphaN := exp(√−1*2π/N);
 for J := 0 step 1 until (N ÷ 2) − 1 do
 begin
  FACTOR[J] := X; X := X*AlphaN
 end;
 sub bin Fourier3(0, N)
end;
```

Consider interleaving in groups of eight (say). The bottom three bits of the indices of such a group will be 0, 1, 2, 3, 4, 5, 6, 7. Dealing with the *relative* position within the group of eight elements, the elements 0, 1, 2 and 3 will be transferred to positions 0, 2, 4, 6 (respectively) and the elements 4, 5, 6, 7 will be transferred to positions 1, 3, 5, 7 (respectively). So, those starting in the *lower* half will have their relative addresses *doubled*, while for those in the upper half a typical element whose address is I will go to location $(I - 4)*2 + 1$.

Thus if the most significant bit of the *old* relative address is 0, the other bits are shifted *up* one place to obtain the *new* relative address, and the zero inserted at the least significant end. However, if the most significant bit is 1, then this bit is taken off, the other bits shifted *up* one place and a 1 bit inserted at the least significant place. In either case a one-place cyclic upward shift of the bottom three bits defines the relation between the initial and final position of the element. This is also true for all other group sizes (e.g. 4, 16, 32, 64, etc).

Let us now consider the final destination of an element which starts in position I, where the binary representation of I (an integer in the range 0 to $N - 1$) is

$$I_p\ I_{p-1}\ I_{p-2} \ldots I_1, I_0$$

where $2^p = N \div 2$ and I_p, I_{p-1}, etc. are the binary digits, I_p being the most significant.

At the first effective stage of interleaving (at a deep level of recursion) merely the bottom two bits are cycled—i.e. new address will be $(I_p, I_{p-1}, \ldots I_2, I_0, I_1)$. Then, after the second stage of interleaving, the address becomes $I_p, I_{p-1}, \ldots I_3, I_0, I_1, I_2$. At each stage a new bit, (working steadily up from the least significant end) will be put in the least significant position and intermediate bits will be shifted up one place.

Finally the address will become

Table 7

```
procedure Bin Fourier4(Y, N);
 value N; complex array Y; integer N;
begin
 integer S, J, K; complex array FACTOR[0:N ÷ 2 − 1];
 complex X, AlphaN;
 procedure sub Bin Fourier4(START, NN);
  value START, NN; integer START, NN;
 begin
  integer MM, JJ, C, D; complex XX;
  if NN = 1 then Y[START] := 2*Y[START]
  else
  begin
   C := 0; D := N ÷ NN; MM := NN ÷ 2;
   for JJ := START step 1 until START + MM − 1 do
   begin
    XX := Y[JJ];
    Y[JJ] := (XX + Y[JJ + MM])/2;
    Y[JJ + MM] := (XX − Y[JJ + MM])*
      FACTOR[C]/2;
    C := C + D
   end;
   sub Bin Fourier4(START, MM);
   sub Bin Fourier4(START + MM, MM)
  end
 end;
 integer procedure BACKWARDS(I, P);
  value I, P; integer I, P;
 begin
 Comment this integer procedure produces as a result the
         integer found by reversing the P-bit integer I
 end;
 X := 1; AlphaN := exp(√−1*2π/N);
 for J := 0 step 1 until N ÷ 2 − 1 do
 begin
  FACTOR[J] := X; X := X*AlphaN
 end;
 sub Bin Fourier4(0, N);
 S := log2 (N);
 for J := 1 step 1 until N − 2 do
 begin
  K := BACKWARDS(J, S);
  if K > J then
  begin X := Y[J]; Y[J] := Y[K]; Y[K] := X
  end
 end
end;
```

$$I_0, I_1, I_2, \ldots I_{p-1}, I_p \ ,$$

which is the number formed by reversing the binary digits of I.

Thus the relation between I and J where J is the ultimate destination of the element starting in location I, is that J is the number found by reversing the binary digits in I.

This relation is clearly symmetric so that I and J are the locations of two elements which become interchanged. To implement this idea in the procedure we merely omit the interleaving section at the end of *sub Bin Fourier3* (which was not coded anyway) and incorporate an interchanging routine after the call of *sub Bin Fourier* in the outer procedure *Bin Fourier4*.

This routine employs an integer procedure *BACKWARDS* which will carry out a bit-reversing operation. It has two integer parameters the number I to be reversed and P, the number of bits in I.

Since the operation is symmetric the elements are arranged in pairs, for interchanging, with the exception of the elements which are unmoved by the process. Typical members of this latter class are the first and last elements, (numbered 0 and $N-1$ respectively).

The method adopted is to vary J from 1 in steps of 1 to $N-2$ (thus omitting the boundary elements which are to be left unmoved) and for each value of J work out its partner location, K, (by using *BACKWARDS*). Thus J will take on, at some point, each value of any particular pair. To prevent double-swapping (with fatal results) we only carry out the interchange if J is the *lower* of the two values J and K. Also swapping is inhibited if $J = K$.

This final version, in **Table 7**, thus includes all the refinements described in the paper. Further improvements in efficiency are still possible; but they do not alter the method in principle, and may be incorporated as part of the process of coding the algorithm in some high level language available on a computer.

The development given here applies only to the case where N is an exact power of two. In fact, similar reasoning can be applied to any value of N which is not prime. But this would be more complicated and less efficient.

Appendix

The basic formula is

$$F_P^N = \frac{2}{N} * \sum_{J=0}^{N-1} \alpha_N\uparrow(P*J)*Y[J]$$

$$(0 \leq P \leq N-1)$$

where $\alpha_N = \exp(\sqrt{-1}*2\pi/N)$.

Let us assume that N is even and write $M = N \div 2$.

For even Fourier coefficients, the basic formula becomes

$$F_{2R}^N = \frac{2}{N} * \sum_{J=0}^{N-1} \alpha_N\uparrow(2*R*J)*Y[J]$$

$$(0 \leq R \leq M-1)$$

$$= \frac{2}{N} * \left(\sum_{J=0}^{M-1} \alpha_N\uparrow(2*R*J)*Y[J] + \sum_{J=M}^{2*M-1} \alpha_N\uparrow(2*R*J)*Y[J] \right)$$

Since $\alpha_N\uparrow(N*R) = +1$ for all integral values of R we may divide the 2nd term by this factor.

This term then becomes

$$\sum_{J=M}^{2*M-1} (\alpha_N\uparrow(2*R*J - N*R))*Y[J]$$

$$= \sum_{J=M}^{2*M-1} (\alpha_N\uparrow(2*R*(J-M)))*Y[J]$$

If we write $JJ = J - M$ this becomes

$$\sum_{JJ=0}^{M-1} (\alpha_N\uparrow(2*R*JJ))*Y[JJ+M]$$

$$= \sum_{J=0}^{M-1} (\alpha_N\uparrow(2*R*J))*Y[J+M]$$

(change of variable)

$$\therefore \quad F_{2R}^N = \frac{2}{N} * \left(\sum_{J=0}^{M-1} (\alpha_N\uparrow(2*R*J))*Y[J] + \sum_{J=0}^{M-1} (\alpha_N\uparrow(2*R*J))*Y[J+M] \right)$$

$$= \frac{2}{M} * \sum_{J=0}^{M-1} (\alpha_N\uparrow(2*R*J))* \left(\frac{Y[J] + Y[J+M]}{2} \right)$$

Since $M = N \div 2$

$$\alpha_M = \alpha_N\uparrow 2$$

$$\therefore \quad F_{2R}^N = \frac{2}{M} * \sum_{J=0}^{M-1} \alpha_M\uparrow(R*J)*SUM[J]$$

where

$$SUM[J] = \frac{(Y[J] + Y[J+M])}{2} . \qquad (6)$$

For *odd* terms the basic formula becomes

$$F_{(2R+1)}^N = \frac{2}{N} * \sum_{J=0}^{N-1} \alpha_N\uparrow((2*R+1)*J)*Y[J]$$

$$(0 \leq R \leq M-1)$$

$$= \frac{2}{N} * \left(\sum_{J=0}^{M-1} \alpha_N\uparrow((2*R+1)*J)*Y[J] + \sum_{J=M}^{2*M-1} \alpha_N\uparrow((2*R+1)*J)*Y[J] \right)$$

Since

$$\alpha_N\uparrow \left(N * \frac{(2*R+1)}{2} \right)$$

$$= (\alpha_N\uparrow(N*R))*\alpha_N\uparrow(N/2) = -1 \ ,$$

for all integral values of R we may divide the second term by

$$-\alpha_N\uparrow \left(N * \frac{(2*R+1)}{2} \right) .$$

This term then becomes

$$-\sum_{J=M}^{2*M-1} \alpha_N\uparrow((2*R+1)*J - (2*R+1)*M)*Y[J]$$

$$= -\sum_{J=M}^{2*M-1} \alpha_N\uparrow((2*R+1)*(J-M))*Y[J] \ .$$

If we write $JJ = J - M$.

This becomes $-\sum_{JJ=0}^{M-1} \alpha_N\uparrow((2*R+1)*JJ)*Y[JJ+M]$

$$= -\sum_{J=0}^{M-1} \alpha_N\uparrow((2*R+1)*J)*Y[J+M]$$

(change of variable)

$$F_{2R+1}^N = \frac{2}{N} * \left(\sum_{J=0}^{M-1} \alpha_N\uparrow((2*R+1)*J)*Y[J] \right.$$

$$- \sum_{J=0}^{M-1} \alpha_N\uparrow((2*R + 1)*J)*Y[J + M])\Bigg)$$

$$= \frac{2}{M} * \sum_{J=0}^{M-1} \alpha_N\uparrow((2*R + 1)*J) * \frac{(Y[J] - Y[J + M])}{2}.$$

Since $\alpha_M = (\alpha_N)\uparrow 2$

$$F^N_{2R+1} = \frac{2}{M} * \left(\sum_{J=0}^{M-1} \alpha_N\uparrow(2*R*J)*\alpha_N\uparrow(J) * \frac{(Y[J] - Y[J+M])}{2} \right)$$

$$= \frac{2}{M} * \sum_{J=0}^{M-1} \alpha_M\uparrow(R*J)*DIFF[J]$$

where

$$DIFF[J] = (\alpha_N\uparrow J) * \frac{(Y[J] - Y[J + M])}{2} \qquad (7)$$

Equations (6) and (7) correspond to the two formulae assumed in Section 3.

References

COOLEY, J. W., and TUKEY, J. W. (1965). An Algorithm for the machine calculation of complex Fourier series, *Mathematics of Computation*, Vol. 19, pp. 297-301, April 1965.

NAUR, P. (ed) (1960) Report on the Algorithmic Language ALGOL 60

WIRTH, N. (1971). Program development by stepwise refinement, *CACM*, Vol. 14, No. 4, pp. 221-227.