DIF Radix-2 FFT Algorithm

Below is the Fortran code for a Decimation-in-Frequency, Radix-2, three butterfly Cooley-Tukey FFT followed by a bit-reversing unscrambler.

```
C   A COOLEY-TUKEY RADIX 2, DIF  FFT PROGRAM
C   THREE-BF, MULT BY  1  AND  J  ARE REMOVED
C   COMPLEX INPUT DATA IN ARRAYS X AND Y
C   TABLE LOOK-UP OF W VALUES
C      C. S. BURRUS, RICE UNIVERSITY, SEPT 1983
C---------------------------------------------------------
    SUBROUTINE FFT (X,Y,N,M,WR,WI)
    REAL X(1), Y(1), WR(1), WI(1)
C--------------MAIN FFT LOOPS-----------------------------
C
    N2 = N
    DO 10 K = 1, M
        N1 = N2
        N2 = N2/2
        JT = N2/2 + 1
        DO 1 I = 1, N, N1
        L = I + N2
        T    = X(I) - X(L)
        X(I) = X(I) + X(L)
        X(L) = T
        T    = Y(I) - Y(L)
        Y(I) = Y(I) + Y(L)
        Y(L) = T
   1        CONTINUE
        IF (K.EQ.M) GOTO 10
        IE  = N/N1
        IA  = 1
        DO 20 J = 2, N2
        IA = IA + IE
        IF (J.EQ.JT) GOTO 50
        C = WR(IA)
        S = WI(IA)
        DO 30 I = J, N, N1
            L = I + N2
            T    = X(I) - X(L)
            X(I) = X(I) + X(L)
            TY   = Y(I) - Y(L)
            Y(I) = Y(I) + Y(L)
            X(L) = C*T + S*TY
```

This page is in 2 books:

- **Fast Fourier Transforms**
  - **Authors:** C. Sidney Burrus
  - **Revised:**Nov 18, 2012
  - Go to book
- **Fast Fourier Transforms (6x9 Version)**
  - **Authors:** C. Sidney Burrus
  - **Revised:**Aug 16, 2012
  - Go to book

principle, size 1). This might naturally suggest a recursive implementation in which the tree is traversed "depth-first" as in Figure(right) and the algorithm of Pre—one size $n/2$ transform is solved completely before processing the other one, and so on. However, most traditional FFT implementations are non-recursive (with rare exceptions [link]) and traverse the tree "breadth-first" [link] as in Figure(left)—in the radix-2 example, they would perform n (trivial) size-1 transforms, then $n/2$ combinations into size-2 transforms, then $n/4$ combinations into size-4 transforms, and so on, thus making $\log_2 n$ passes over the whole array. In contrast, as we discuss in "Discussion", FFTW employs an explicitly recursive strategy that encompasses **both** depth-first and breadth-first styles, favoring the former since it has some theoretical and practical advantages as discussed in "FFTs and the Memory Hierarchy".

$\mathbf{Y}[0,...,n-1] \leftarrow \mathbf{recfft\ 2}(n, \mathbf{X}, \iota)$:

IF n=1 THEN

$\quad Y[0] \leftarrow X[0]$

ELSE

$\quad \mathbf{Y}[0,...,n/2-1] \leftarrow \mathbf{recfft2}\ (n/2, \mathbf{X}, 2\iota)$

$\quad \mathbf{Y}[n/2,...,n-1] \leftarrow \mathbf{recfft2}\ (n/2, \mathbf{X}+\iota, 2\iota)$

$\quad$ FOR $k_1 = 0$ TO $(n/2)-1$ DO

$\quad\quad t \leftarrow \mathbf{Y}[k_1]$

$\quad\quad \mathbf{Y}[k_1] \leftarrow t + \omega_n^{k_1}\mathbf{Y}[k_1+n/2]$

$\quad\quad \mathbf{Y}[k_1+n/2] \leftarrow t - \omega_n^{k_1}\mathbf{Y}[k_1+n/2]$

$\quad$ END FOR

END IF

NOT_CONVERTED_YET: caption

A depth-first recursive radix-2 DIT Cooley-Tukey FFT to compute a DFT of a power-of-two size $n = 2^m$. The input is an array $\mathbf{X}$ of length n with stride ι (i.e., the inputs are $\mathbf{X}[\ell\iota]$ for $\ell = 0,...,n-1$) and the output is an array $\mathbf{Y}$ of length n (with stride 1), containing the DFT of $\mathbf{X}$ [Equation 1]. $\mathbf{X}+\iota$ denotes the array beginning with $\mathbf{X}[\iota]$. This algorithm operates out-of-place, produces in-order output, and does not require a separate bit-reversal stage.

A second ordering distinction lies in how the digit-reversal is performed. The classic approach is a single, separate digit-reversal pass following or preceding the arithmetic computations; this approach is so common and so deeply embedded into FFT lore that many practitioners find it difficult to imagine an FFT without an explicit bit-reversal stage. Although this pass requires only $O(n)$ time [link], it can still be non-negligible, especially if the data is out-of-cache; moreover, it neglects the possibility that data reordering during the transform may improve memory locality. Perhaps the oldest alternative is the Stockham **auto-sort** FFT [link], [link], which transforms back and forth between two arrays with each butterfly, transposing one digit each time, and was popular to improve contiguity of access for vector computers [link]. Alternatively, an explicitly recursive style, as in FFTW, performs the digit-reversal implicitly at the "leaves" of its computation when operating out-of-place (see section "Discussion"). A simple example of this style, which computes in-order output using an out-of-place radix-2 FFT without explicit bit-reversal, is shown in the algorithm of Pre [corresponding to Figure(right)]. To operate in-place with $O(1)$ scratch storage, one can interleave small matrix transpositions with the butterflies [link], [link], [link], [link], and a related strategy in FFTW [link] is briefly described by

$N = 2^p 3^q 5^r$. Set $NI = 2^p$, $IP = p$. We first compute the required rotation and set up the table of twiddle factors:

```
      COMPLEX TRIGS(NI)
      DEL=4.0*ASIN(1.0)/FLOAT(NI)
      IROT=MOD((N/NI),NI)
      KK=0
      DO 10 K=1, NI
      ANGLE=FLOAT(KK)*DEL
      TRIGS(K)=CMPLX(COS(ANGLE),SIN(ANGLE))
      KK=KK+IROT
      IF (KK.GT.NI) KK=KK-NI
10    CONTINUE
```

The first $(p+1)/2$ radix-2 passes are then performed by the following code:

```
      COMPLEX X(N), W, Z
      NH=N/2
      INC=N/NI
      DO 50 L=1, (IP+1)/2
      LA=2**(L-1)
      JA=0
      JB=NH/LA
      KK=1
      DO 40 K=0, JB-1, INC
      W=TRIGS(KK)
      DO 30 J=K+1, N, N/LA
      IA=JA+J
      IB=JB+J
      DO 20 I=1, INC
      Z=W*(X(IA)-X(IB))
      X(IA)=X(IA)+X(IB)
      X(IB)=Z
      IA=IA+NI
      IF (IA.GT.N) IA=IA-N
      IB=IB+NI
      IF (IB.GT.N) IB=IB-N
20    CONTINUE
30    CONTINUE
      KK=KK+LA
40    CONTINUE
50    CONTINUE
```

The details of the indexing may be understood by comparing this code with Table 1 ($N = 40$, $NI = 8$). The three outer loops are very similar to the three loops of the code presented in [18], which performed the first half of a self-sorting in-place radix-2 algorithm. In the present case these loops set up base addresses in the first column of Table 1. The advantage of the Ruritanian map is that the entries in the first column

SIAM J. SCI. STAT. COMPUT.
Vol. 13, No. 3, pp. 676–686, May 1992

003

A GENERALIZED PRIME FACTOR FFT ALGORITHM FOR ANY $N = 2^p3^q5^r$ *

CLIVE TEMPERTON†

Abstract. Prime factor fast Fourier transform (FFT) algorithms have two important advantages: they can be simultaneously self-sorting and in-place, and they have a lower operation count than conventional FFT algorithms. The major disadvantage of the prime factor FFT has been that it was only applicable to a limited set of values of the transform length N. This paper presents a generalized prime factor FFT, which is applicable for any $N = 2^p3^q5^r$, while maintaining both the self-sorting in-place capability and the lower operation count. Timing experiments on the Cray Y-MP demonstrate the advantages of the new algorithm.

Key words. fast Fourier transform (FFT), prime factor algorithm (PFA), self-sorting FFT, in-place FFT

AMS(MOS) subject classification. 65T05

1. Introduction. Fast Fourier transform (FFT) algorithms can be defined whenever the transform length N can be factorized as $N = N_1N_2 \cdots N_k$, where the factors N_i are integers. Though there are many variants of these algorithms, they fall into two basic categories: those based on the prime factor algorithm (PFA) of Good [5], which are only applicable if the factors N_i are mutually prime, and those descended from the algorithm of Cooley and Tukey [3], for which there is no such restriction (indeed the most familiar case is $N_i = 2$ for all i).

The prime factor algorithms have two important advantages. For a given value of N, the operation count is lower than that for the corresponding Cooley–Tukey algorithm. Moreover, the PFA can be made both self-sorting (input and output both

10	3.01

Fig. 3. Minimization of number of operations.

optimally programmed FORTR

Fig. 6 shows a slightly differe


```
      SUBROUTINE FFT(A,M)
      COMPLEX A(1024),U,W,T
      N= 2**M
      NV2 = N/2
      NM1 = N-1
      J=1
      DO 7 I=1,NM1
      IF(I.GE.J) GO TO 5
      T = A(J)
      A(J) = A(I)
      A(I) = T
    5 K=NV2
    6 IF(K.GE.J) GO TO 7
      J = J-K
      K=K/2
      GO TO 6
    7 J= J+K
      PI = 3.14159265358979
      DO 20 L=1,M
      LE = 2**L
      LE1 = LE/2
      U = (1.0,0.)
      W=CMPLX(COS(PI/LE1),SIN(PI/LE1))
      DO 20 J=1,LE1
      DO 10 I=J,N,LE
      IP = I+LE1
      T=A(IP)*U
      A(IP)=A(I)-T
   10 A(I)=A(I)+T
   20 U=U*W
      RETURN
      END
```

algorithm. This one requires t
mutually prime and uses a di
one-dimensional indices j and
(j_1, j_0) and (n_1, n_0), respectively
2. The result is a procedure quit
above, except that the phase
calculation of $A_1(j_0, n_0)$. This c
in combination with the prev
ample, to introduce an odd fac
is a power of two. Then one
factor algorithm using the pow
the r-point subseries.

The time required for compu
conventional program and by
illustrated in Fig. 7. It is seen h
arrive at values of N which are
ventional methods unfeasible.
N of the fast methods shows t
vances permit the measurement

The Fast Fourier Transform and Its Applications

JAMES W. COOLEY, PETER A. W. LEWIS, AND PETER D. WELCH

Abstract—The advent of the fast Fourier transform method has greatly extended our ability to implement Fourier methods on digital computers. A description of the alogorithm and its programming is given here and followed by a theorem relating its operands, the finite sample sequences, to the continuous functions they often are intended to approximate. An analysis of the error due to discrete sampling over finite ranges is given in terms of aliasing. Procedures for

It is most likely that with the relatively small values of N used in preelectronic computer days, the former methods were easier to use and took fewer operations. Consequently, the methods requiring $N \log N$ operations were neglected. With the arrival of electronic computers capable of doing calculations of Fourier transforms with

```
procedure FFT (A, n, w)

    # Preconditions:
    #   A is a Vector of length n;
    #   n is a power of 2;
    #   w is a primitive n-th root of unity.
    #
    # The Vector A represents the polynomial
    #   a(z)  =  A[1] + A[2]*z + ... + A[n]*z^(n-1) .
    #
    # The value returned is a Vector of the values
    #    [ a(1), a(w), a(w^2), ... , a(w^(n-1)) ]
    # computed via a recursive FFT algorithm.

    if n = 1 then
        return A
    else
        Aeven <-- Vector( [A[1], A[3], ..., A[n-1]] )
        Aodd  <-- Vector( [A[2], A[4], ..., A[n]] )

        Veven <-- FFT( Aeven, n/2, w^2 )
        Vodd <-- FFT( Aodd, n/2, w^2 )

        V <-- Vector(n)  # Define a Vector of length n
        for i from 1 to n/2 do
            V[i] <-- Veven[i] + w^(i-1)*Vodd[i]
            V[n/2 + i] <-- Veven[i] - w^(i-1)*Vodd[i]
        end do
        return V
    end if
end procedure
```

Vector radix fast Fourier transform

4 Author(s) D. Harris ; J. McClellan ; D. Chan ; H. Schuessler View All Authors

Abstract
Authors
References
Citations
Keywords
Metrics

Abstract:
A new radix-2 two-dimensional direct FFT developed by Rivard is generalized in this paper to include arbitrary radices and non-square arrays. It is shown that the radix-4 version of this algorithm may require significantly fewer computations than conventional row-column transform methods. Also, the new algorithm eliminates the matrix transpose operation normally required when the array must reside on a bulk storage device. It requires the same number of passes over the array on bulk storage as efficient matrix transpose routines, but produces the transform in bit-reversed order. An additional pass over the data is necessary to sort the array if normal ordering is desired.

Published in: ICASSP '77. IEEE International Conference on Acoustics, Speech, and Signal Processing

Date of Conference: 9-11 May 1977

DOI: 10.1109/ICASSP.1977.1170349

Date Added to IEEE *Xplore*: 29 January 2003

Publisher: IEEE

Conference Location: Hartford, CT, USA, USA

Authors
References
Citations
Keywords

in each step is proportional to n.

Code 1.1 (recursive radix 2 DIT FFT) *Pseudo code for a recursive procedure of the (radix 2) DIT FFT algorithm,* is *must be +1 (forward transform) or -1 (backward transform):*

```
procedure rec_fft_dit2(a[], n, x[], is)
// complex a[0..n-1] input
// complex x[0..n-1] result
{
    complex b[0..n/2-1], c[0..n/2-1]   // workspace
    complex s[0..n/2-1], t[0..n/2-1]   // workspace

    if n == 1 then  // end of recursion
    {
        x[0] := a[0]
        return
    }

    nh := n/2

    for k:=0 to nh-1  // copy to workspace
    {
        s[k] := a[2*k]    // even indexed elements
        t[k] := a[2*k+1]  // odd  indexed elements
    }

    // recursion: call two half-length FFTs:
    rec_fft_dit2(s[],nh,b[],is)
    rec_fft_dit2(t[],nh,c[],is)

    fourier_shift(c[],nh,is*1/2)

    for k:=0 to nh-1  // copy back from workspace
    {
        x[k]    := b[k] + c[k];
        x[k+nh] := b[k] - c[k];
    }
}
```

[source file: recfftdit2.spr]

The data length n must be a power of 2. The result is in x[]. Note that normalization (i.e. multiplication of each element of x[] by $1/\sqrt{n}$) is not included here.

[FXT: recursive_dit2_fft in slow/recfft2.cc] The procedure uses the subroutine

Code 1.2 (Fourier shift) *For each element in c[0..n-1] replace c[k] by c[k] times* $e^{v\,2\,\pi\,i\,k/n}$. *Used with* $v = \pm 1/2$ *for the Fourier transform.*

```
procedure fourier_shift(c[], n, v)
{
    for k:=0 to n-1
    {
        c[k] := c[k] * exp(v*2.0*PI*I*k/n)
    }
}
```

cf. [FXT: fourier_shift in fft/fouriershift.cc]

The recursive FFT-procedure involves $n \log_2(n)$ function calls, which can be avoided by rewriting it in a non-recursive way. One can even do all operations *in place*, no temporary workspace is needed at all. The price is the necessity of an additional data reordering: The procedure revbin_permute(a[],n) rearranges the array a[] in a way that each element a_x is swapped with $a_{\tilde{x}}$, where $\tilde{x}$ is obtained from x by reversing its binary digits. This is discussed in section 8.1.

Code 1.3 (radix 2 DIT FFT, localized) *Pseudo code for a non-recursive procedure of the (radix 2) DIT algorithm,* is *must be -1 or +1:*

```
procedure fft_dit2_localized(a[], ldn, is)
// complex a[0..2**ldn-1] input, result
{
    n := 2**ldn  // length of a[] is a power of 2
    revbin_permute(a[],n)
    for ldm:=1 to ldn  // log_2(n) iterations
    {
        m  := 2**ldm
        mh := m/2
        for r:=0 to n-m step m  // n/m iterations
        {
            for j:=0 to mh-1  // m/2 iterations
            {
                e := exp(is*2*PI*I*j/m)  // log_2(n)*n/m*m/2 = log_2(n)*n/2 computations
                u := a[r+j]
                v := a[r+j+mh] * e
                a[r+j]    := u + v
                a[r+j+mh] := u - v
            }
        }
    }
}
```

[source file: fftdit2localized.spr]

[FXT: dit2_fft_localized in fft/fftdit2.cc]

This version of a non-recursive FFT procedure already avoids the calling overhead and it works in place. It works as given, but is a bit wasteful. The (expensive!) computation e := exp(is*2*PI*I*j/m) is done $n/2 \cdot \log_2(n)$ times. To reduce the number of trigonometric computations, one can simply swap the two inner loops, leading to the first 'real world' FFT procedure presented here:

Code 1.4 (radix 2 DIT FFT) *Pseudo code for a non-recursive procedure of the (radix 2) DIT algorithm,* is *must be -1 or +1:*

```
procedure fft_dit2(a[], ldn, is)
// complex a[0..2**ldn-1] input, result
```

```
{
    n := 2**ldn
    revbin_permute(a[],n)
    for ldm:=1 to ldn  // log_2(n) iterations
    {
        m  := 2**ldm
        mh := m/2
        for j:=0 to mh-1  // m/2 iterations
        {
            e := exp(is*2*PI*I*j/m)  // 1 + 2 + ... + n/8 + n/4 + n/2 = n-1 computations
            for r:=0 to n-m step m
            {
                u := a[r+j]
                v := a[r+j+mh] * e
                a[r+j]    := u + v
                a[r+j+mh] := u - v
            }
        }
    }
}
```

[source file: fftdit2.spr]

[FXT: `dit2_fft` in `fft/fftdit2.cc`]

Swapping the two inner loops reduces the number of trigonometric (`exp()`) computations to `n` but leads to a feature that many FFT implementations share: Memory access is highly nonlocal. For each recursion stage (value of `ldm`) the array is traversed `mh` times with `n/m` accesses in strides of `mh`. As `mh` is a power of 2 this can (on computers that use memory cache) have a very negative performance impact for large values of `n`. On a computer where the CPU clock (366MHz, AMD K6/2) is 5.5 times faster than the memory clock (66MHz, EDO-RAM) I found that indeed for small `n` the localized FFT is slower by a factor of about 0.66, but for large `n` the same ratio is in favour of the 'naive' procedure!

It is a good idea to extract the `ldm==1` stage of the outermost loop, this avoids complex multiplications with the trivial factors $1 + 0\,i$: Replace

```
    for ldm:=1 to ldn
    {
```

by

```
    for r:=0 to n-1 step 2
    {
        {a[r], a[r+1]} := {a[r]+a[r+1], a[r]-a[r+1]}
    }
    for ldm:=2 to ldn
    {
```

$z^{(2j+\delta)n/2} = e^{\pm \pi i \delta}$ is equal to plus/minus 1 for $\delta = 0/1$ (k even/odd), respectively.

The last two equations are, more compactly written, the

Idea 1.2 (radix 2 DIF step) *Radix 2 decimation in frequency step for the FFT:*

$$\mathcal{F}[a]^{(even)} \stackrel{n/2}{=} \mathcal{F}\left[a^{(left)} + a^{(right)}\right] \tag{1.36}$$

$$\mathcal{F}[a]^{(odd)} \stackrel{n/2}{=} \mathcal{F}\left[\mathcal{S}^{1/2}\left(a^{(left)} - a^{(right)}\right)\right] \tag{1.37}$$

Code 1.5 (recursive radix 2 DIF FFT) *Pseudo code for a recursive procedure of the (radix 2) decimation in frequency FFT algorithm,* `is` *must be +1 (forward transform) or -1 (backward transform):*

```
procedure rec_fft_dif2(a[], n, x[], is)
// complex a[0..n-1] input
// complex x[0..n-1] result
{
    complex b[0..n/2-1], c[0..n/2-1]   // workspace
    complex s[0..n/2-1], t[0..n/2-1]   // workspace
    if n == 1 then
    {
        x[0] := a[0]
        return
    }
    nh := n/2
    for k:=0 to nh-1
    {
        s[k] := a[k]      // 'left'  elements
        t[k] := a[k+nh]   // 'right' elements
    }
    for k:=0 to nh-1
    {
        {s[k], t[k]} := {(s[k]+t[k]), (s[k]-t[k])}
    }
    fourier_shift(t[],nh,is*0.5)
    rec_fft_dif2(s[],nh,b[],is)
    rec_fft_dif2(t[],nh,c[],is)
    j := 0
    for k:=0 to nh-1
    {
        x[j]   := b[k]
        x[j+1] := c[k]
        j := j+2
    }
}
```

[source file: recfftdif2.spr]

The data length `n` must be a power of 2. The result is in `x[]`.

Algorithms for programmers

ideas and source code

Jörg Arndt
arndt@jjj.de

This document[1] was LaTeX'd at September 26, 2002

Algorithm 7.2 The iterative radix-2 DIT FFT algorithm in pseudo-code.

```
begin
  PairsInGroup := N/2                         Begin with N/2 butterflies in one group
  NumOfGroups := 1                            Same twiddle factor is employed in a group
  Distance := N/2
  while NumOfGroups < N do
       for K := 0 to NumOfGroups − 1 do       Combine pairs in each group
           JFirst := 2 * K * PairsInGroup
           JLast := JFirst + PairsInGroup − 1
           Jtwiddle := K                      Access consecutive w[m]
           W := w[Jtwiddle]                   Assume w[m] = ω_N^ℓ, m bit-reverses ℓ
           for J := JFirst to JLast do
               Temp := W * a[J + Distance]
               a[J + Distance] := a[J] − Temp
               a[J] := a[J] + Temp
           end for
       end for
       PairsInGroup := PairsInGroup/2
       NumOfGroups := NumOfGroups * 2
       Distance := Distance/2
  end while
end
```

with the understanding that on input, $a[i_4 i_3 i_2 i_1 i_0]$ contains $x_{i_4 i_3 i_2 i_1 i_0}$, and on output, $a[i_4 i_3 i_2 i_1 i_0]$ contains the bit-reversed $X_{i_0 i_1 i_2 i_3 i_4}$. Refer to Figure 7.6 for the decimal subscripts of all 32 bit-reversed output elements $X_{i_0 i_1 i_2 i_3 i_4}$.

COMPUTATIONAL MATHEMATICS SERIES

INSIDE the FFT BLACK BOX

Serial and Parallel Fast Fourier Transform Algorithms

Eleanor Chu
Alan George