Efficient Input Reordering for the DCT Based on a Real-Valued Decimation in Time FFT

by Rainer Storn
TR-95-061
September 1995

Abstract

The possibility of computing the Discrete Cosine Transform (DCT) of length $N=2^\nu$, $\nu$ integer, via an $N$-point Discrete Fourier Transform (DFT) is widely known from the literature. In this correspondence it will be demonstrated that this computation can be done in-place by just employing butterfly swaps if the input reordering - necessary for the DCT computation via DFT - is combined with the bit-reverse scrambling required by the decimation in time Fast Fourier Transform-algorithm.
Introduction

The Discrete Cosine Transform (DCT) has a wide range of applications in image and signal processing and many algorithms for its fast computation have been devised [1]-[4]. One particular attractive approach is the computation via real valued Fast Fourier Transform (FFT) algorithms as the latter are very well developed and high performance computer code is readily available [5]. However, the fact that the input reordering required for this type of computation can be done in-place by just using butterfly swaps has not been addressed so far. In case of transform lengths \( N=2^v \), \( v \) integer, Butterfly swaps mean that if data in location \( P \) are moved to location \( Q \) then the data having previously been in \( Q \) have to be moved to \( P \). This paper elaborates this property and presents pertinent source code in C.

The DCT of an \( N \)-point real sequence \( x_n \) is most often defined as [1]

\[
C_{N,m} = \frac{2\varepsilon_m}{N} \sum_{n=0}^{N-1} x_n \cdot \cos \left( \frac{\pi(2n+1)m}{2N} \right) \quad \text{for } m=0, 1, \ldots, N-1
\]  

(1)

with

\[
\varepsilon_m = \begin{cases} 
\frac{1}{\sqrt{2}} & \text{for } m = 0 \\
1 & \text{otherwise}
\end{cases}
\]

(2)

The Discrete Fourier Transform (DFT)

\[
F_{N,m} = R_{N,m} + j \cdot I_{N,m} = \sum_{n=0}^{N-1} f_n \cdot e^{-j\frac{2\pi nm}{N}} , \quad \text{for } m=0, 1, \ldots, N-1
\]

(3)

for which a huge body of fast computational algorithms exists can be utilized to compute the DCT by employing the mapping found in [4]. It is defined by

\[
f_n = \begin{cases} 
x_{2n} & \text{for } 0 \leq n \leq \left\lfloor \frac{N-1}{2} \right\rfloor \\
x_{2N-2n-1} & \text{for } \left\lfloor \frac{N+1}{2} \right\rfloor \leq n \leq N-1
\end{cases}
\]

(4)

yielding

\[
C_{N,m} = 2(R_{N,m} \cdot \cos \left( \frac{\pi m}{2N} \right) + I_{N,m} \cdot \sin \left( \frac{\pi m}{2N} \right) ) \quad \text{for } m=1, 2, \ldots, \frac{N}{2}-1
\]

\[
C_{N,N-m} = 2(R_{N,m} \cdot \sin \left( \frac{\pi m}{2N} \right) - I_{N,m} \cdot \cos \left( \frac{\pi m}{2N} \right) )
\]

(5)
and \[ C_{N,0} = 2R_{N,0} \] (6)

as well as \[ C_{N, N/2} = \sqrt{2} \cdot R_{N, N/2} \] (7)

From (5), (6) and (7) it can clearly be seen that only half of the DFT outputs are required which is due to the fact that \( f_n \) is real and hence the DFT output values are conjugate complex. The computation of real-valued FFT algorithms, especially for \( N=2^\nu \), has been studied extensively in the literature an excellent survey of which can be found in [5] and [6]. We will concentrate exclusively on the case \( N=2^\nu \) and on the Cooley-Tukey or decimation in time approach. This kind of FFT requires its input values to be in bit-reversed order which is well suited for an efficient in-place computation of the DCT, rendering a Fast Cosine Transform (FCT). The input scrambling of the corresponding FCT for \( N=8 \) is depicted in fig. 1. The scrambling consists of two passes, with the first pass representing the scrambling defined by eq. (4) where the sequence \( x_n \) is transformed into sequence \( f_n \).

The second pass performs the bit-reverse reordering which is required by the decimation in time FFT and renders the sequence \( u_n \).

If we define \( \text{bitrev}_k(n) \) as the function which reverses the bit pattern of the binary representation of index \( n \) with respect to \( k \) bits, we can define

\[ u_n = f_{\text{bitrev}_k(n)} \cdot \quad (8) \]

If we set

\[ n = \text{bitrev}_k(j) \]

we obtain

\[ u_{\text{bitrev}_k(j)} = f_{\text{bitrev}_k(\text{bitrev}_k(j))} = f_j \cdot \quad (9) \]
Eqs. (8) and (9) define the swapability property of the bit reverse input reordering of the decimation in time FFT.

By regarding the right part of Fig. 1 we observe that obviously not only the mapping of \( f_n \) onto \( u_n \) but also the mapping of \( x_n \) onto \( u_n \) exhibits the swapability property. This means that

\[
\text{if } u_m = x_i \quad \text{then } u_i = x_m. \tag{10}
\]

In order to prove the swapability property let us first consider the indices \( n \in \left[0, \frac{N-1}{2}\right] \). According to (4) we can set index \( i \) in (10) to \( 2n \), so that

\[
x_i = x_{2n} = f_n. \tag{11}
\]

For convenience we will represent the index \( 2n \) as a string of \( k \) bits symbolized by "bits"0, where the substring "bits" represents an arbitrary bit pattern consisting of \( k-1 \) bits. The least significant bit (LSB) of the bitstring "bits"0 is always zero as \( 2n \) is an even number. With this new representation of indices we can recast (11) into

\[
x_i = x_{\text{bits}0} = f_{0^*\text{bits}0}. \tag{12}
\]

Note that in order to preserve the number of \( k \) bits we had to augment the index \( n \) of \( f_n \) by a most significant bit (MSB) of value zero. The addition of this MSB can be done without loss of generality.

Combining (9), (10) and (12) yields

\[
u_m = u_{\text{stib}0} = x_i = x_{\text{bits}0} \tag{13}
\]

where "stib" represents the bit reversed \((k-1)\)-bit string "bits". Due to symmetry properties it is evident that also

\[
x_m = x_{\text{stib}0} = u_i = u_{\text{bits}0} \tag{14}
\]

holds. Eqs. (13) and (14) show the swapability property for the above range of \( n \).

To complete the proof of the swapability property we also have to consider the case for

\[
n \in \left[ \frac{N+1}{2}, N-1 \right] \quad \text{where } x_i = x_{2N-2n-1} = f_n. \tag{15}
\]

It is important to realize that \( 2N-2n-1 \) is just another way of representing the one's complement of \( 2n \) with respect to \( k+1 \) bits. An alternative way of representation is \( 1\text{compl}_2 \left( 2n \right) \) or \( 0^{n}\text{bits}^0 \) in the string notation. Using the above relationship we can recover \( n \) from \( 2N-2n-1 \) by taking the one's complement of \( 2N-2n-1 \) with respect to \( k+1 \) bits and dividing by two to eventually obtain

\[
x_i = x_{0^{n}\text{bits}0} = f_{0^*\text{bits}0}. \tag{16}
\]

Combining (9), (10) and (16) finally yields

\[
u_m = u_{\text{stib}0} = u_{0^*\text{stib}0} = x_i = x_{0^{n}\text{bits}0}. \tag{17}
\]

Again we can employ symmetry observations to verify that

\[
x_m = x_{\text{stib}0} = x_{0^*\text{stib}0} = u_i = u_{0^{n}\text{bits}0} \tag{18}
\]
holds, which completes the proof.

With the above knowledge we can easily write an in-place FCT-algorithm where the reordering requires nothing more than butterfly swaps if we utilize a real-valued FFT algorithms based on the decimation in time approach. An example program in C is given below.
The Program Code Example in C

```c
#include <stdio.h>
#include <math.h>
#define pi 3.14159265358979323846
#define pi2 6.28318530717958647692
#define MAX 1024          /* MAX = Maximum transform length */

*sizeof---Type definitions---------------------------------------------------*/
float x[MAX];             /* array for real input and output */
float wr[MAX], wi[MAX];   /* FFT-coefficients */

sizeof---Deklarations-------------------------------------------------------*/
void fct(float x[], float wr[], float wi[], int N, int nexp);
void twiddle(float wr[], float wi[], int N);

sizeof---Main program-------------------------------------------------------*/
void main()
{
  int N, nexp, i;

  printf("Type exponent: ");
  scanf("%d",&nexp);

  sizeof---Determine transform length N-----------------------------------*/
  N = 1;
  if (nexp > 0)              /* N = 2**nexp                 */
    for (i=1; i<=nexp; i++)
      N = N*2;

  sizeof---Generate sequence in the time domain---------------------------*/
  for (i=0; i<N; i++)
    { x[i] = cos(pi2*i/N); }

  sizeof---Compute twiddle factors of real-valued FFT-------------------*/
  twiddle(wr, wi, N);

  sizeof---Fast Cosine Transform of input sequence----------------------*/
  fct(x, wr, wi, N, nexp);

  printf("nFCT
");
  for (i=0; i<N; i++)
    printf("x[%i] = %f 
",i,x[i]);
}

void fct(float x[], float wr[], float wi[], int N, int nexp)
{/**
** fct() computes a DCT via a real valued, in-place Cooley-
** Tukey Radix-2 FFT. **
** Real input and output data are in array x[]. **
** Output will be in order **
** [re[0], re[1], ..., re[N/2], im[N/2-1], ..., im[1]] **
** after the FFT part is finished. The post computation yields **
** the DCT outputs in normal order. **
** The FFT program is mainly taken from "Real-Valued Fast Fourier **
** Transform Algorithms" by Sorensen, H.V. et alii, ASSP-35, **
** June 1987, pp. 849 - 863. **
** Ported and modified by Rainer Storn, **
** ICSI, 1947 Center Street, Berkeley, CA 94707 **
** E-mail: storn@icsi.berkeley.edu. **
**
*******************************************************************/

```c
int   i, i1, i2, i3, i4, j, k, n1, n2, no4, n4, adr, ee;
int   it2, jt2, ip21, jc;
float xt, cc, ss, t1, t2;

/*---------Fill buffer array with the DFT/DCT-sequence---------*/
/*---------Do the reordering.----------------------------------*/
/*---------digit reverse counter-------------------------------*/

j   = 0;
n1  = N-1;
no2 = N/2;
no4 = N/4;
for (i=0; i<= no4; i++)
{
  it2 = i*2;
  jt2 = j*2;
  if (it2 < jt2)
  {
    xt       = x[jt2];
x[jt2]   = x[it2];
x[it2]   = xt;
  }
  ip21 = it2+1;
  jc   = n1-jt2; /* complement */
  if (ip21 < jc)
  {
    xt       = x[jc];
x[jc]    = x[ip21];
x[ip21]  = xt;
  }
  k = no4;       /* small bit reversal */
  while (k < j+1)
  {
    j = j-k;
    k = k/2;
  }
  j = j+k;
}

/*--------Start of real-valued FFT-part------------------------*/
/*--------length two butterflies------------------------------*/

for (i=0; i<N; i=i+2)
{
  xt     = x[i];
x[i]   = xt + x[i+1];
x[i+1] = xt - x[i+1];
}

/*--------other butterflies-----------------------------------*/
```
n2 = 1;
for (k=2; k<=nexp; k++)
{
    n4 = n2;
    n2 = 2*n4;
    n1 = 2*n2;
    ee = N/n1;
    for (i=0; i<N; i=i+n1)
    {
        xt = x[i];
        x[i] = xt + x[i+n2];
        x[i+n2] = xt - x[i+n2];
        x[i+n4+n2] = -x[i+n4+n2];
        adr = ee;
        for (j=1; j<= n4-1; j++) /* note that in the first run n4=1 */
        {
            i1 = i+j;
            i2 = i-j+n2;
            i3 = i+j+n2;
            i4 = i-j+n1;
            cc = wr[adr];
            ss = wi[adr];
            adr = adr + ee;
            t1 = x[i3]*cc + x[i4]*ss;
            t2 = x[i3]*ss - x[i4]*cc;
            x[i4] = x[i2] - t2;
            x[i3] = -x[i2] - t2;
            x[i2] =  x[i1] - t1;
            x[i1] =  x[i1] + t1;
        }
    }
}

/*----------Post computation for DCT output-------------------------*/
/*----------Normalization factor 2/N.-------------------------------*/
/*----------(Exception is x[0] where sqrt(2)/N is the factor)-------*/
x[0] = x[0]*sqrt(2.)/(float)N;
for (i=1; i<N/2; i++)
{
    ss = sin(pi*i*0.5/N);
    cc = cos(pi*i*0.5/N);
    xt = (x[i]*cc + x[N-i]*ss)*2/(float)N;
    x[N-i] = (x[i]*ss - x[N-i]*cc)*2/(float)N;
    x[i] = xt;
}
x[N/2] = x[N/2]*sqrt(2.)/(float)N;
}

void twiddle(float wr[], float wi[], int N)
/**********************************************************************
** twiddle() calculates the twiddle factors for an **
** N-point FFT. **
**********************************************************************/
{
    float inc;
    int i;
    inc = pi2/N;
    for (i=0; i<(N/2); i=i+1)
\{
  \text{wr}[i] = \cos(\text{inc} \times \text{float}i);
  \text{wi}[i] = \sin(\text{inc} \times \text{float}i);
\}

**Conclusion**

It has been demonstrated that an N-point DCT with N=2^\nu can be computed efficiently via a real-valued decimation in time FFT by just employing butterfly swaps for the input reordering. As computer code for many real-valued FFT algorithms is publicly available, this way of DCT-computation becomes even more attractive.
References

Fig. 1: In-place reordering necessary for an 8-point FCT based on the decimation in time FFT.