Simple Fast Fourier Transform algorithms in C

November 24, 2012

J.L.

Often when programming 'quick-and-dirty' data-analysis software for do-it-yourself projects, it is beneficial to include short stand-alone algorithms to the code instead of using a whole library worth of complex black-box algorithms. In the context of the Fast Fourier Transform (FFT) it is not so simple to compile for example the freely available FFTW code collection along with ones own small projects. This example explains some details on the FFT algorithm given in the book 'Numerical Recipes in C'. This algorithm is the most simplest FFT implementation and it is suitable for many practical applications which require fast evaluation of the Discrete Fourier Transform.

The algorithm given in the 'Numerical Recipes in C' belongs to a group of algorithms that implement the Radix-2 Decimation-In-Time (DIT) transform. In this context the background of the algorithm is explained using the so-called 'butterfly diagram', which visualises how the original data-elements flow through the transform process. This example also uses the butterfly diagrams to modify the DIT implementation to a Radix-2 Decimation-In-Frequency (DIF) type of algorithm.

In basic principles the FFT algorithms rely on the symmetries of the general DFT evaluation when the amount of data points is 2^n (n can be any integer). The FFT typically defines a multiplier

$$W_N^M = e^{-2\pi i \frac{M}{N}},$$

which in the algorithms is sequentially used for multiplying the data sequence. When $N = 4$ the transform matrix is

$$\begin{bmatrix} W_4^0 & W_4^0 & W_4^0 & W_4^0 \\ W_4^0 & W_4^1 & W_4^2 & W_4^3 \\ W_4^0 & W_4^2 & W_4^4 & W_4^6 \\ W_4^0 & W_4^3 & W_4^6 & W_4^9 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & W_4^1 & W_4^2 & W_4^3 \\ 1 & W_4^2 & W_4^0 & W_4^2 \\ 1 & W_4^3 & W_4^2 & W_4^1 \end{bmatrix}$$

The matrix on the right side emphasises the symmetry of the 3x3 submatrix ($W_N^M \neq W_N^0$) on the diagonal direction. The FFT algorithm tries to minimise the mathematical operations used for calculating the 'twiddle factors' W_N^M and the minimisation is achieved with the help of the symmetrical structure of the transformation matrix.

When considering the CPU time (processing time of the computer) used for mathematical operations, the trigonometric functions typically require a large amount time since those need to be approximated as a series of sums

and products. In the FFT, the complex exponential function needs to be evaluated using the sine and cosine functions (Euler formula). The CPU time can be saved considerably if the value of the sine function is evaluated only once and the following values would be obtained by a constant increment, which is added to the previous value of the sine function. A common procedure is to write

$$e^{ia+ib} = e^{ia} \cdot e^{ib} = e^{ia} + Z \cdot e^{ia},$$

where a is the starting angle and b is a constant incremental angle. When b is constant, then also the function of $\sin b$ is constant. Solving the previous equation for Z gives:

$$Z = e^{ib} - 1 = -(1 - \cos b) + i \sin b = -2 \sin^2 \left(\frac{b}{2} \right) + i \sin b,$$

using the *versine* identity formula for the term $1 - \cos b$. After the value for the multiplier Z has been evaluated, the increments of the trigonometric functions cosine and sine can be calculated with respect to the starting-angle exponential e^{ia} as

$$e^{ia} + Z \cdot e^{ia}$$

as already indicated above. The comments in the code later on will explain the use of this incremental method in practise.

1 A basic Decimation-In-Time (DIT) algorithm

The second edition of the book 'Numerical Recipes in C' introduces the most basic DIT algorithm. The algorithm is most easily explained with the help of the butterfly diagram in Figure 1, which shows the transform sequence for a data sequence consisting of 8 data points. The notation $x(n)$ refers to the original data and the notation $X(n)$ refers to the values obtained after the transform. The use of the twiddle factors W_N^M in each point is indicated along with the need to multiply with -1 before summing. The dots in the conjunction locations indicate the addition of the values at that location.

Because the transformed components $X(n)$ do not appear in orderly fashion with respect to the original data points, the input data sequence needs to be shuffled prior to using this FFT algorithm. The shuffling can be done with a so-called 'bit reversal' algorithm, which is explained in detail in the book of 'Numerical Recipes'.

The C-code that implements the butterfly flow of Figure 1 is given in the

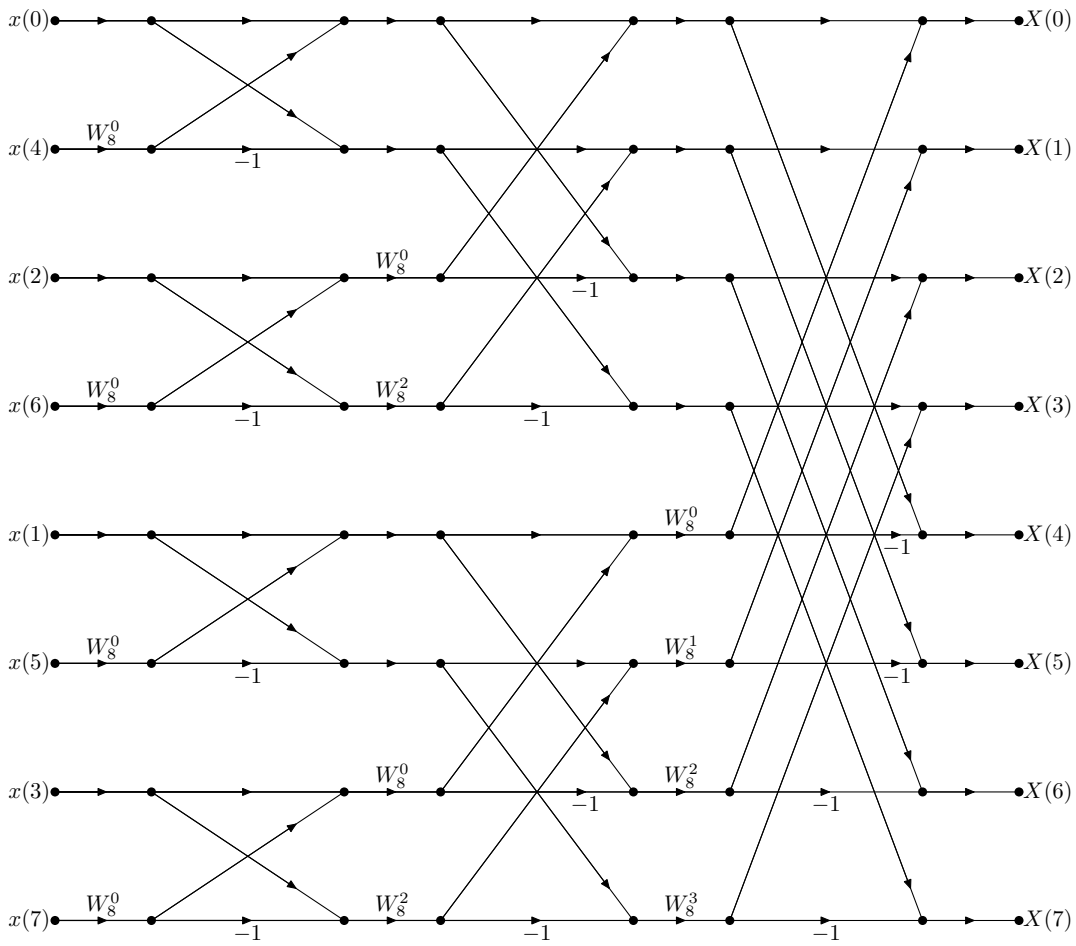

Figure 1: A butterfly diagram of the Radix-2 Decimation-In-Time FFT

following listing. Compared to the code in the 'Numerical Recipes' I have added a long comment section in the beginning and I have rewritten the trigonometric incrementation part of the algorithm so that the idea would be more clear from the context. Some debug prints have been added as well.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define TRSIZ 8
#define SWAP(a,b) tempr=(a); (a)=(b); (b)=tempr
/*****************************************************************************
* FUNCTION: dittt()
*
* PURPOSE:
*     - Calculates Fast Fourier Transform of given data series using bit
*       reversal prior to FFT. This function is directly taken from the book
*       "Numerical Recipes in C". The algorithm follows the butterfly diagram
```

```
*       of the radix-2 Decimation In Time (DIT) implementation of FFT, where
*       the bit reversal is mandatory. The function replaces the (input) array
*       data[1..2*nn] by its discrete Fourier transfor, if the parameter isign
*       is given as -1; or replaces data[1..2*nn] by nn times its inverse
*       discrete Fourier transform, if isign is given as 1. The array of data[]
*       is a complex array of length nn or, equivalently, a real array of
*       length 2*nn. nn MUST be an integer power of 2, but this is not checked!
*
*       The needed increments of trigonometric functions are handled via
*       a clever identity using the fact that:
*
*       exp(ia + ib) = exp(ia)*exp(ib) = exp(ia) + Z*exp(ia),
*
*       where a is the starting angle and b is the increment. Solving for Z
*       gives:
*
*       Z = exp(ib) - 1 = -(1 - cos(b)) + isin(b) = -2*sin^2(b/2) + isin(b).
*
*       Then the increments can be calculated with respect to the zero-angle
*       set by wr (omega-real) and wi (omega-imaginary) by relation:
*
*       (wr + wi) + Z*(wr + wi) = (wr + wi) + [-2*sin^2(b/2) + isin(b)](wr + wi).
*
*       By setting wpr (omega-phase-real) = -2*sin^2(b/2) and wpi = sin(b),
*       one has
*
*       (wr + wi) + (wpr + wpi)*(wr + wi) = exp(ia) + Z*exp(ia),
*
*       where the real part is: wr + wpr*wr - wpi*wi
*       and the imaginary part is: wi + wpr*wi + wpi*wr.
*       The actual incremental parts here are:
*       wpr*wr - wpi*wi for real part and wpr*wi + wpi*wr for imaginary part.
*
*
* INPUT:
*       - data[] = Array containing the data to be transformed
*                  The transformed data is stored back to this array
*                  so that the real and imaginary parts are following
*                  each other --> size of array = 2*size of data
*       - nn     = size of data = size of array/2, has to be a power of 2
*       - isign  = if -1, calculates the FFT;
*                  if 1, calculates the IFFT i.e. the inverse.
*
* OUTPUT: - (void)
*/
```

```
void dittt()
{
double wtemp, wr, wpr, wpi, wi, theta;
double tempr, tempi;
int N = TRSIZ;
int i = 0, j = 0, n = 0, k = 0, m = 0, isign = -1,istep,mmax;
double data1[2*TRSIZ] = {0,0,1,0,4,0,9,0,2,0,3,0,4,0,5,0};
double *data;

data = &data1[0] - 1;
n = N*2;
j = 1;
// do the bit-reversal
  for (i = 1; i < n; i += 2) {
    if (j > i) {
      SWAP(data[j], data[i]);
      SWAP(data[j+1], data[i+1]);
    }

    m = n >> 1;
    while (m >= 2 && j > m) {
      j -= m;
      m >>= 1;
    }
    j +=m;
  }

// calculate the FFT
  mmax = 2;
  while (n > mmax) {
    istep = mmax << 1;
    theta = isign*(6.28318530717959/mmax);
    wtemp = sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi = sin(theta);
    wr = 1.0;
    wi = 0.0;

    for (m = 1; m < mmax; m += 2) {
      for (i = m; i <= n; i += istep) {
        j = i + mmax;
        tempr = wr*data[j] - wi*data[j+1];
        tempi = wr*data[j+1] + wi*data[j];
        data[j] = data[i] - tempr;
        data[j+1] = data[i+1] - tempi;
```

```
        data[i] =  data[i] + tempr;
        data[i+1] = data[i+1] + tempi;
        printf("\ni = %d ,j = %d, m = %d, wr = %f , wi = %f",(i-1)/2,(j-1)/2,m,wr,wi);
      }
      printf("\nm = %d ,istep = %d, mmax = %d, wr = %f , wi = %f, Z = %f"
             ,m,istep,mmax,wr,wi,atan(wi/wr)/(6.28318530717959/(1.0*n/2)));
      wtemp = wr;
      wr += wtemp*wpr - wi*wpi;
      wi += wtemp*wpi + wi*wpr;
    }
  mmax = istep;
  }

// print the results
printf("\nFourier components from the DIT algorithm:");
for (k = 0; k < 2*N; k +=2 )
printf("\n%f  %f", data[k+1], data[k+2]);
}  // end of dittt()
```

2 A basic Decimation-In-Frequency (DIF) algorithm

According to the theory of the Discrete Fourier Transform, time and frequency are on opposite sides of the transform boundary. Therefore it is not surprising that the frequency-tagged DIF algorithm is kind of a mirror image of the time-tagged DIT algorithm. The butterfly diagram of the DIF FFT is shown in Figure 2. If the same bit-reversal reordering of the data points is used here, the bit-reversal needs to be done *after* the transform.

The C-code that implements the butterfly flow of Figure 2 is given in the following listing. This code is the authors own modification from the DIT version and this version assumingly is not available in the related literature. The modification was relatively easy, but only after studying the butterfly graph very carefully and adding debug prints inside the FFT loops. It is easy to see that the DIF algorithm runs in an opposite order compared to the DIT version. However, the DIF implementation requires a few additional lines to store some temporary values and therefore the algorithm requires slightly more computational steps than the DIT algorithm. Obviously the FFT components obtained from the DIF should equal the results from DIT.

Due to the mirroring properties of the DIT and DIF algorithms, in some cases it is possible to save some CPU time. For example, if the DIT is used for evaluating the FFT transform and the DIF is used right after for the inverse

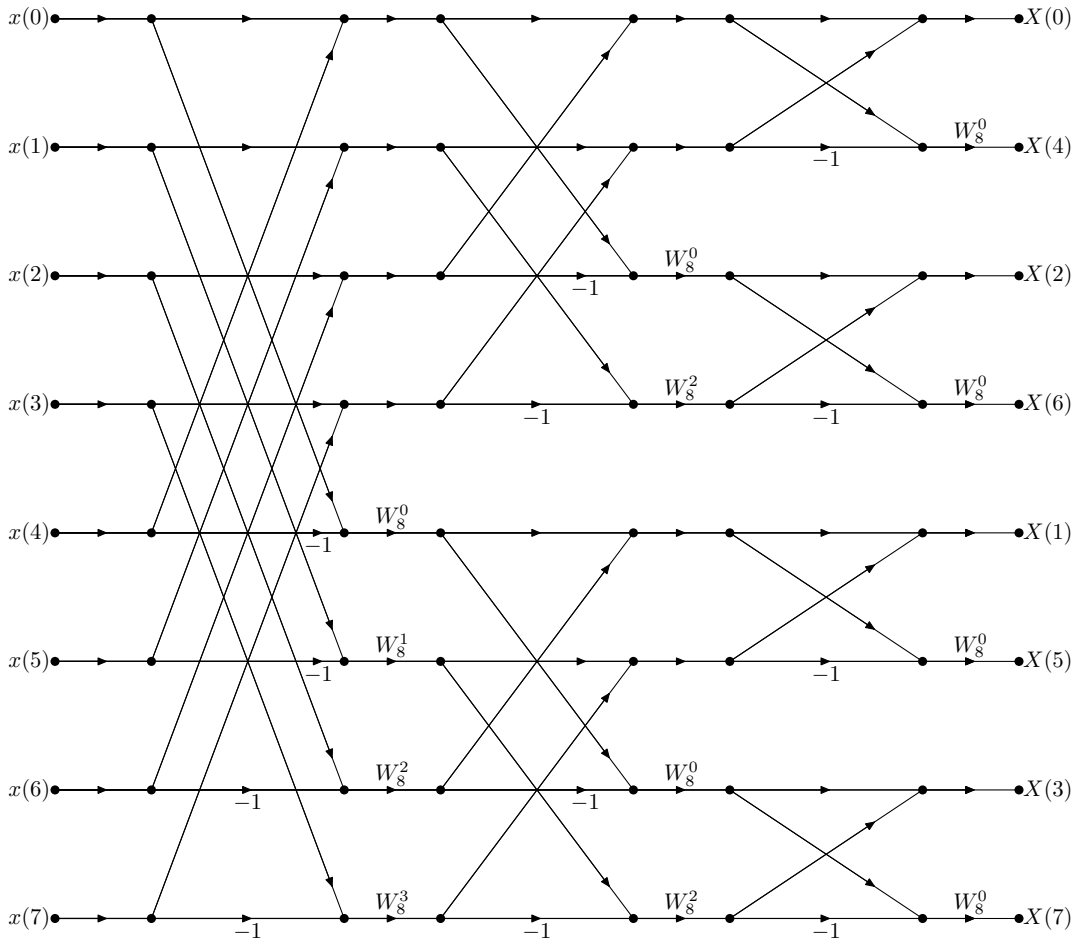

Figure 2: A butterfly diagram of the Radix-2 Decimation-In-Frequency FFT

transform, then the bit-reversal is not needed at all. This might be useful when using the FFT in correlation calculations.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define TRSIZ 8
#define SWAP(a,b) tempr=(a); (a)=(b); (b)=tempr
/*****************************************************************************
* FUNCTION: diftt()
*
* PURPOSE:
*     - Calculates Fast Fourier Transform of given data series using bit
*       reversal after the FFT. This algorithm follows the butterfly diagram
*       of the radix-2 Decimation In Frequency (DIF) implementation of FFT,
*       which is modified from the previous Decimation In Time (DIT) algorithm.
*       The function replaces the (input) array
```

```
*       data[1..2*nn] by its discrete Fourier transfor, if the parameter isign
*       is given as -1; or replaces data[1..2*nn] by nn times its inverse
*       discrete Fourier transform, if isign is given as 1. The array of data[]
*       is a complex array of length nn or, equivalently, a real array of
*       length 2*nn. nn MUST be an integer power of 2, but this is not checked!
*
*       The needed increments of trigonometric functions are handled via
*       a clever identity using the fact that:
*
*       exp(ia + ib) = exp(ia)*exp(ib) = exp(ia) + Z*exp(ia),
*
*       where a is the starting angle and b is the increment. Solving for Z
*       gives:
*
*       Z = exp(ib) - 1 = -(1 - cos(b)) + isin(b) = -2*sin^2(b/2) + isin(b).
*
*       Then the increments can be calculated with respect to the zero-angle
*       set by wr (omega-real) and wi (omega-imaginary) by relation:
*
*       (wr + wi) + Z*(wr + wi) = (wr + wi) + [-2*sin^2(b/2) + isin(b)](wr + wi).
*
*       By setting wpr (omega-phase-real) = -2*sin^2(b/2) and wpi = sin(b),
*       one has
*
*       (wr + wi) + (wpr + wpi)*(wr + wi) = exp(ia) + Z*exp(ia),
*
*       where the real part is: wr + wpr*wr - wpi*wi
*       and the imaginary part is: wi + wpr*wi + wpi*wr.
*       The actual incremental parts here are:
*       wpr*wr - wpi*wi for real part and wpr*wi + wpi*wr for imaginary part.
*
*
* INPUT:
*       - data[] = Array containing the data to be transformed
*                  The transformed data is stored back to this array
*                  so that the real and imaginary parts are following
*                  each other --> size of array = 2*size of data
*       - nn     = size of data = size of array/2, has to be a power of 2
*       - isign  = if -1, calculates the FFT;
*                  if 1, calculates the IFFT i.e. the inverse.
*
* OUTPUT: - (void)
*/
void diftt()
{
```

```
double wtemp, wr, wpr, wpi, wi, theta;
double tempr, tempi;
int N = TRSIZ;
int i = 0, j = 0, n = 0, k = 0, m = 0, isign = -1,istep,mmax;
double data1[2*TRSIZ] = {0,0,1,0,4,0,9,0,2,0,3,0,4,0,5,0};
double *data;

data = &data1[0] - 1;
n = N*2;
mmax = n/2;
// calculate the FFT
  while (mmax >= 2) {
    istep = mmax << 1;
    theta = isign*(6.28318530717959/mmax);
    wtemp = sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi = sin(theta);
    wr = 1.0;
    wi = 0.0;

    for (m = 1; m < mmax; m += 2) {
      for (i = m; i <= n; i += istep) {
        j = i + mmax;
        tempr = data[i];
        tempi = data[i+1];
        data[i] = data[i] + data[j];
        data[i+1] = data[i+1] + data[j+1];
        data[j] = tempr - data[j];
        data[j+1] = tempi - data[j+1];
        tempr = wr*data[j] - wi*data[j+1];
        tempi = wr*data[j+1] + wi*data[j];
        data[j] = tempr;
        data[j+1] = tempi;
        printf("\ni = %d ,j = %d, m = %d, wr = %f , wi = %f",(i-1)/2,(j-1)/2,m,wr,wi);
      }
       printf("\nm = %d ,istep = %d, mmax = %d, wr = %f , wi = %f, Z = %f"
              ,m,istep,mmax,wr,wi,atan(wi/wr)/(6.28318530717959/(1.0*n/2)));
      wtemp = wr;
      wr += wtemp*wpr - wi*wpi;
      wi += wtemp*wpi + wi*wpr;
    }
  mmax = mmax/2;
  }

// do the bit-reversal
```

```
j = 1;
  for (i = 1; i < n; i += 2) {
    if (j > i) {
      SWAP(data[j], data[i]);
      SWAP(data[j+1], data[i+1]);
    }

    m = n >> 1;
    while (m >= 2 && j > m) {
      j -= m;
      m >>= 1;
    }
    j +=m;
  }

// print the results
printf("\nFourier components from the DIF algorithm:");
for (k = 0; k < 2*N; k +=2 )
printf("\n%f  %f", data[k+1], data[k+2]);
} // end of diftt()
```