

fft

Fast Fourier Transform

Spring 2008

Notes compiled by Alpar Sevgen

Consider an ordered sequence $\{x\}$ with N -elements: $\{x\} := \{x_0, x_1, x_2, \dots, x_{N-1}\}$. We assume N is a power of two, $N = 2^L$. (If not, zero pad the sequence). We want to take the Discrete Fourier Transform (DFT) of this sequence,

$$F_{[N]}(k; \{x\}) = \sum_{j'=0}^{N-1} W_{[N]}^{kj'} x_{j'} , \quad \text{where } k = 0, 1, \dots, N-1 , \quad (1)$$

and

$$W_{[N]} = e^{-i\frac{2\pi}{N}} \equiv W . \quad (2)$$

The notation here is slightly changed from our previous notes,

$$F_{[N]}(k; \{x\}) = \sqrt{N} \langle \omega_k | x \rangle . \quad (3)$$

This is done here to help us follow the computations better, and note the change in the normalization also.

Now do the sum in Eq.(1) on even numbered and odd numbered elements separately [1],

$$F_{[N]}(k; \{x\}) = \sum_{j'=0,2,4,\dots}^{N-2} W_{[N]}^{kj'} x_{j'} + \sum_{j'=1,3,5,\dots}^{N-1} W_{[N]}^{kj'} x_{j'} . \quad (4)$$

Set the dummy summation index $j' = 2j$ for the even terms and $j' = 2j + 1$ for the odd terms,

$$F_{[N]}(k; \{x\}) = \sum_{j=0,1,2,\dots}^{\frac{N}{2}-1} W_{[N]}^{k2j} x_{2j} + \sum_{j=0,1,2,\dots}^{\frac{N}{2}-1} W_{[N]}^{k(2j+1)} x_{2j+1} . \quad (5)$$

Note that

$$W_{[N]}^2 = W_{[\frac{N}{2}]}, \quad (6)$$

so that we can renotate Eq.(5),

$$\begin{aligned} F_{[N]}(k; \{x\}) &= \sum_{j=0,1,2,\dots}^{\frac{N}{2}-1} W_{[\frac{N}{2}]}^{kj} x_{2j} + W_{[N]}^k \sum_{j=0,1,2,\dots}^{\frac{N}{2}-1} W_{[\frac{N}{2}]}^{kj} x_{2j+1}, \\ &= F_{[\frac{N}{2}]}(k; \{x_{\text{even}}\}) + W_{[N]}^k F_{[\frac{N}{2}]}(k; \{x_{\text{odd}}\}), \end{aligned} \quad (7)$$

where we used the definition Eq.(1) and wrote the sums over the even terms and odd terms as DFT's involving the $N/2$ element ordered sets $\{x_{\text{even}}\} := \{x_0, x_2, x_4, \dots, x_{N-2}\}$ and $\{x_{\text{odd}}\} := \{x_1, x_3, x_5, \dots, x_{N-1}\}$, and of dimension *not* N , but $N/2$. Thus a DFT of dimension N is reduced to the sum of two DFT's of dimension $N/2$. Let us define the even and odd decimation operators \mathcal{D}_e and \mathcal{D}_o such that

$$\begin{aligned} \mathcal{D}_e(x) &= \{x_{\text{even}}\} := \{x_0, x_2, x_4, \dots, x_{N-2}\} \\ \mathcal{D}_o(x) &= \{x_{\text{odd}}\} := \{x_1, x_3, x_5, \dots, x_{N-1}\} \end{aligned} \quad (8)$$

So that let us rewrite Eq.(6) again,

$$F_{[N]}(k; \{x\}) = F_{[\frac{N}{2}]}(k; \mathcal{D}_e(x)) + W_{[N]}^k F_{[\frac{N}{2}]}(k; \mathcal{D}_o(x)), \quad (9)$$

It is to be noted that the index k still runs from $0, 1, 2, \dots, N-1$. But the DFT's $F_{[\frac{N}{2}]}(k; \{y\})$ with $\{y\} = \mathcal{D}_e(x)$ or $\{y\} = \mathcal{D}_o(x)$ are each periodic in $N/2$,

$$F_{[\frac{N}{2}]}(k; \{y\}) = F_{[\frac{N}{2}]}(k + \frac{N}{2}; \{y\}). \quad (10)$$

We also note that $W_{[N]}^{k+\frac{N}{2}} = -W_{[N]}^k$. Thus, restricting the index k now only to the range $0, 1, 2, \dots, \frac{N}{2} - 1$ appropriate for DFT's of dimension $N/2$, we can re-express Eq.(9),

$$\begin{aligned} F_{[N]}(k; \{x\}) &= F_{[\frac{N}{2}]}(k; \mathcal{D}_e(x)) + W_{[N]}^k F_{[\frac{N}{2}]}(k; \mathcal{D}_o(x)), \\ F_{[N]}(k + \frac{N}{2}; \{x\}) &= F_{[\frac{N}{2}]}(k; \mathcal{D}_e(x)) - W_{[N]}^k F_{[\frac{N}{2}]}(k; \mathcal{D}_o(x)) \quad (11) \\ \text{with } k &= 0, 1, 2, \dots, \frac{N}{2} - 1. \end{aligned}$$

Clearly, this process can continue down the ladder, a DFT of dimension $N/2$ can be written as a sum of two DFT's of dimension $N/4$, a DFT of dimension $N/4$ can be written as a sum of two DFT's of dimension $N/8$, and the process stops when a DFT of dimension 2 is written as a sum of two DFT's of dimension 1, and which is trivial to evaluate

$$F_{[N=1]}(k; \{x_\ell\}) = F_{[N=1]}(k = 0; \{x_\ell\}) = x_\ell, \quad (12)$$

so that the DFT of a one element sequence is just that element itself. We also note that partitioning a sequence into even and odd terms and then continuing again shuffles the members of the sequence. To consider a specific example, take $N = 8 = 2^3$ and $\{x\} := \{x_o, x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$, note that \mathcal{D}_e selects out the zeroth, second, fourth...terms of a sequence (even terms) and \mathcal{D}_o selects out the first, third, fifth... terms of a sequence (odd terms),

$$\begin{aligned}
& \text{first stage} && : \\
& \mathcal{D}_e(x) &= & \{x_o, x_2, x_4, x_6\} \\
& \mathcal{D}_o(x) &= & \{x_1, x_3, x_5, x_7\} \\
& \text{second stage} && : \\
& \mathcal{D}_e\mathcal{D}_e(x) &= & \mathcal{D}_e\{x_o, x_2, x_4, x_6\} = \{x_o, x_4\} \\
& \mathcal{D}_o\mathcal{D}_e(x) &= & \{x_2, x_6\} \\
& \mathcal{D}_e\mathcal{D}_o(x) &= & \{x_1, x_5\} \\
& \mathcal{D}_o\mathcal{D}_o(x) &= & \{x_3, x_7\} \\
& \text{third and last stage} && : \\
& \mathcal{D}_e\mathcal{D}_e\mathcal{D}_e(x) &= & \mathcal{D}_e\{x_o, x_4\} = x_o \\
& \mathcal{D}_o\mathcal{D}_e\mathcal{D}_e(x) &= & x_4 \\
& \mathcal{D}_e\mathcal{D}_o\mathcal{D}_e(x) &= & x_2 \\
& \mathcal{D}_o\mathcal{D}_o\mathcal{D}_e(x) &= & x_6 \\
& \mathcal{D}_e\mathcal{D}_e\mathcal{D}_o(x) &= & x_1 \\
& \mathcal{D}_o\mathcal{D}_e\mathcal{D}_o(x) &= & x_5 \\
& \mathcal{D}_e\mathcal{D}_o\mathcal{D}_o(x) &= & x_3 \\
& \mathcal{D}_o\mathcal{D}_o\mathcal{D}_o(x) &= & x_7.
\end{aligned} \quad (13)$$

Is there pattern to this shuffling? Lets look at the binary representation of the indices of the elements of x in this shuffled order (as given above in the last stage in Eq.(13):

decimal	binary	\mathcal{D} operators
0	000	eee
4	100	oee
2	010	oeo
6	110	ooe
1	001	eeo
5	101	oeo
3	011	eoo
7	111	ooo

where , for example, 4 is the index of x_4 . We see that 0, 2, 5, 7 keep their positions after the shuffling. But instead of 1 we have 4. Comparison of their binary structures show that their bits are reversed. Also instead of 3 we have 6. Again, comparison of their binary structures show that their bits are reversed. We also see that binary digit 0 corresponds to \mathcal{D}_e and binary digit 1 corresponds to \mathcal{D}_o . So, if we want to get the Fourier transform components $F_{[N]}(k)$ of $\{x\}$ in correct order, we must at the beginning, bit reverse the data $\{x\}$ and use $\{x_{BR}\}$. Otherwise the Fourier components will *not* come out right.

Now, we start by bit-reversing the data. Then every element, considered as a one-element sequence, is equal to its Fourier transform, Eq.(12). We then use Eq.(11) to recombine elements into sequences twice as large, two-element sequences. After which we combine two-element sequences into four element sequences until we reach at the L^{th} stage F_N . Thus, for example, at the K^{th} stage (dimension $n = 2^K$) we have a paired sequence of *evens* $\{e_0, e_1, \dots, e_{\frac{n}{2}-1}\}$ and *odds* $\{o_0, o_1, \dots, o_{\frac{n}{2}-1}\}$, and the combination proceeds in accordance with Eq.(11). It is better , however, to rewrite that equation for an intermediate stage. Let $\{x_A\}$ denote a particular sequence, decimated out of the full sequence $\{x\}$, (for example, $\mathcal{D}_o\mathcal{D}_e\mathcal{D}_o(x) = x_{oeo}$). Our recombination equation then reads:

$$\begin{aligned}
F_{[n]}(k; \{x_A\}) &= F_{[\frac{n}{2}]}(k; \mathcal{D}_e(x_A)) + W^{\frac{N}{n}k} F_{[\frac{n}{2}]}(k; \mathcal{D}_o(x_A)), \\
F_{[n]}(k + \frac{n}{2}; \{x_A\}) &= F_{[\frac{n}{2}]}(k; \mathcal{D}_e(x_A)) - W^{\frac{N}{n}k} F_{[\frac{n}{2}]}(k; \mathcal{D}_o(x_A)) \quad (14) \\
&\text{with } k = 0, 1, 2, \dots, \frac{n}{2} - 1 \text{ and } W = e^{-i\frac{2\pi}{N}}.
\end{aligned}$$

We can also express the Eq.(14) in a vector format [2]:

$$\vec{F}_{[n]}(x_A) = \begin{pmatrix} I_{[\frac{n}{2}]} & D_{[\frac{n}{2}]} \\ I_{[\frac{n}{2}]} & -D_{[\frac{n}{2}]} \end{pmatrix} \begin{bmatrix} \vec{F}_{[\frac{n}{2}]}(x_{eA}) \\ \vec{F}_{[\frac{n}{2}]}(x_{oA}) \end{bmatrix}, \quad (15)$$

where $I_{\frac{n}{2}}$ is a unit matrix in $\frac{n}{2}$ dimensions, $D_{[\frac{n}{2}]}$ is a diagonal matrix in $\frac{n}{2}$ dimensions, with the elements along the main diagonal: $1, W_{\frac{n}{n}^1}, W_{\frac{n}{n}^2}, \dots, W_{\frac{n}{n}^{\frac{n}{2}-1}}$. In more detail, lists are given as:

Table A	
I	II
e_o	$e_o + W_{[n]}^0 o_0 = e_0 + o_0$
e_1	$e_1 + W_{[n]}^1 o_1$
e_2	$e_2 + W_{[n]}^2 o_2$
\cdot	\cdot
\cdot	\cdot
\cdot	\cdot
$e_{\frac{n}{2}-1}$	$e_{\frac{n}{2}-1} + W_{[n]}^{\frac{n}{2}-1} o_{\frac{n}{2}-1}$

o_0	$e_o - W_{[n]}^0 o_0 = e_0 - o_0$
o_1	$e_1 - W_{[n]}^1 o_1$
o_2	$e_2 - W_{[n]}^2 o_2$
\cdot	\cdot
\cdot	\cdot
\cdot	\cdot
$o_{\frac{n}{2}-1}$	$e_{\frac{n}{2}-1} - W_{[n]}^{\frac{n}{2}-1} o_{\frac{n}{2}-1}$

where $W_{[n]} = W_{[N]}^{N/n}$. Column labelled I indicates two lists of dimension $n/2$, one even, the other odd. They are then combined, according to Eq.(14), to form a list of dimension n . This table displays in detail how to recombine lists.

We now provide a step by step procedure for $N = 8$. The input data is $\{x\} := \{x_o, x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$, and after bit reversal we have $\{x_{BR}\} := \{x_o, x_4, x_2, x_6, x_1, x_5, x_3, x_7\}$, that is eight 1-element DFTs. We combine them according to Table A above.

one-term lists	two term lists
$e_o = x_0$	$x_0 + x_4$
- - -	
$o_o = x_4$	$x_0 - x_4$
$e_0 = x_2$	$x_2 + x_6$
- - -	
$o_0 = x_6$	$x_2 - x_6$
x_1	$x_1 + x_5$
- - -	
x_5	$x_1 - x_5$
x_3	$x_3 + x_7$
- - -	
x_7	$x_3 - x_7$

Two term ordered lists are: $\{\vec{F}_{[2]}(x_{ee}), \vec{F}_{[2]}(x_{oe}), \vec{F}_{[2]}(x_{eo}), \vec{F}_{[2]}(x_{oo})\}$, notice that they are still ordered in the bit reversed form ($e = 0, o = 1, 00 = 0, 10 = 2, 01 = 1, 11 = 3$) Now we have four two-term lists. And we combine them according to Table A:

two-term lists	four-term lists
$e_0 = x_0 + x_4$	$x_0 + x_4 + x_2 + x_6$
$e_1 = x_0 - x_4$	$x_0 - x_4 + W^2(x_2 - x_6)$

$o_0 = x_2 + x_6$	$x_0 + x_4 - (x_2 + x_6)$
$o_1 = x_2 - x_6$	$x_0 - x_4 - W^2(x_2 - x_6)$
$e_0 = x_1 + x_5$	$x_1 + x_5 + x_3 + x_7$
$e_1 = x_1 - x_5$	$x_1 - x_5 + W^2(x_3 - x_7)$

$o_0 = x_3 + x_7$	$x_1 + x_5 - (x_3 + x_7)$
$o_1 = x_3 - x_7$	$x_1 - x_5 - W^2(x_3 - x_7)$

Note that we used $W_{[4]}^1 = W_{[8]}^2 = W^2$. Four term ordered lists are: $\{\vec{F}_{[4]}(x_e), \vec{F}_{[4]}(x_o)\}$. We now combine the two four-term lists to obtain the final result, $\vec{F}_{[8]}(x)$.

four-term lists	the eight-term final list
$e_0 = x_0 + x_4 + x_2 + x_6$	$F_{[8]}(0, \{x\}) = e_0 + o_0$
$e_1 = x_0 - x_4 + W^2(x_2 - x_6)$	$F_{[8]}(1, \{x\}) = e_1 + W o_1$
$e_2 = x_0 + x_4 - (x_2 + x_6)$	$F_{[8]}(2, \{x\}) = e_2 + W^2 o_2$
$e_3 = x_0 - x_4 - W^2(x_2 - x_6)$	$F_{[8]}(3, \{x\}) = e_3 + W^3 o_3$

$o_0 = x_1 + x_5 + x_3 + x_7$	$F_{[8]}(4, \{x\}) = e_0 - o_0$
$o_1 = x_1 - x_5 + W^2(x_3 - x_7)$	$F_{[8]}(5, \{x\}) = e_1 - W o_1$
$o_2 = x_1 + x_5 - (x_3 + x_7)$	$F_{[8]}(6, \{x\}) = e_2 - W^2 o_2$
$o_3 = x_1 - x_5 - W^2(x_3 - x_7)$	$F_{[8]}(7, \{x\}) = e_3 - W^3 o_3$

And the final eight term list is $\vec{F}_{[8]}(x)$. So, we first bit-reverse the data. Then the computation of DFT proceeds in $L = 3 = \log_2 2^L$ stages. In the first stage combine single-element lists to two-element lists. Then in the second stage combine two-element lists to four element lists. then in the last stage combine the four-element lists into the final eight term list, which provides the DFT of $\{x\}$. At every stage list-labels "A" are bit-reversed.

Computational aspects: There are various strategies:

- I) non-recursive programming[3, 4, 5, 6]
- II) recursive programming

We first study the non-recursive programming.

I) non-recursive programming

There are several steps:

- i) Bit reverse data
- ii) Combine the single element lists to get two-term lists, label the two-term lists
- iii) Combine the two-term lists to get four term lists. List labels are bit-reversed at every stage. So we know which list to combine with which list.
- iv) Process ends after the L^{th} stage.
- v) Powers of $W = e^{-i\frac{2\pi}{N}}$ are needed great many times. Rather than evaluating each time, again and again, a look-up table can be constructed to save time.
- vi) For large N , memory space has to be used carefully. For that purpose *in place* programming style may be employed.

We first go over at some length of *Bit Reversal*. I will base this discussion on [5] and the comments of Yuksel Gunal on how the code works: Consider an array of data of 8 elements, $x[i]$ where $i = 0, 1, 2, 3, 4, 5, 6, 7$. Let us write the index i together with its bit-reverse $j = \text{BitReverse}(i) = \text{BR}(i)$:

i:decimal	i:binary	j:binary=BR(i)	j: decimal
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Let $j = \text{BR}(i)$ and $j' = \text{BR}(i + 1)$. An observation: as $i \rightarrow i + 1$ their

bit-reversed counterparts $j \rightarrow j'$ in the following way: Take j in binary form as shown in the table above. Start from the left, flip 1's to 0's and at the first encounter of 0 flip it to 1 and stop. Example :

$j' = BR(3)$ is to be obtained from $j = BR(2) = 010$ in the following way: start from left, convert 0 to 1, and stop, $j' = 110$.

$j' = BR(4)$ is to be obtained from $j = BR(3) = 110$ in the following way: start from left, convert 1 to 0, convert 1 to 0, convert 0 to 1 and stop $j' = 001$.

cShow that if you know the binary form of i you can get that of $i + 1$ as follows: start from the right, flip 1's to 0's and at the first encounter of 0 flip it to 1 and stop.

And why does it work? Let $N = 2^L$, so index i goes from 0 to $N - 1$. Let us take $L = 3$ to be specific here. A binary number is then to be written as:

$$i = a_{L-1}2^{L-1} + a_{L-2}2^{L-2} + \dots + a_12^1 + a_02^0, \quad (16)$$

Adding 1 to i means adding (0...001). This number is to be written as

$$i + 1 = a'_{L-1}2^{L-1} + a'_{L-2}2^{L-2} + \dots + a'_12^1 + a'_02^0, \quad (17)$$

What are the new coefficients? If $a_0 = 0 \rightarrow a'_0 = 1$ and stop. If $a_0 = 1 \rightarrow a'_0 = 0$ and move onto a_1 and check it. If it is 0 process stops after conversion. If not, 1 is flipped to 0 and move onto and check a_2 . Until you meet the first 0. Then by flipping it to 1 the process stops.

Now for the bit-reversed numbers: $j = \text{bit-reversed}(i)$ is

$$j = a_02^{L-1} + a_12^{L-2} + \dots + a_{L-2}2^1 + a_{L-1}2^0, \quad (18)$$

and the $j' = BR(i + 1)$ is then,

$$j' = a'_02^{L-1} + a'_12^{L-2} + \dots + a'_{L-2}2^1 + a'_{L-1}2^0, \quad (19)$$

Thus here we start from left on the binary expression of j , and convert 1's to 0 until we meet a first 0 on our approach from left and flip it to 1 and stop. This explains our observation. Note that converting $a_{L-1} = 1$ to 0 means subtracting the number 2^{L-1} , next $1 \rightarrow 0$ conversion means subtracting again this time 2^{L-2} , and if at the k^{th} step 0 is encountered, flipping it to 1 means adding 2^{L-k} . This is how the code works. Consider $j = 000$. First digit on the left is 0 so that for $BR(1)$ we add $2^{L-1} = 2^{3-1} = 2^2 = 4$ and get $BR(1)=100$. For $BR(2)$ now we subtract from $BR(1)$ 4 and add 2 and get

BR(2)=010. For BR(3) we add 4 to BR(2) and get 110.

Exercise: Go over the rest of the list. (In the code below, n_1 does these subtractions and additions). Now the C-code that does all this (note that changing $\text{LENGTH} = N = 2^L$ you can compute for a very long array as well. Try $N = 16$ and $N = 32$):
(copy the file bitReverse.c compile and run. Be sure to understand how it works.)

```
/******  
*  
* code by Douglas Jones  
* http://cnx.rice.edu/content/m12016/latest  
*  
* critical comments by Yuksel Gunal  
*  
* file name: bitReverse.c  
*  
*****/  
  
#include <stdio.h>  
#include <math.h>  
  
#define LENGTH 8  
  
int main()  
{  
  
    int N=LENGTH;  
    double x[LENGTH];  
  
    int i,j,n1;
```

```

/* integer j will be found as the bit reverse of i */

    double t1;

/* construct the array to be "bit-reversed" */

    for (i=0; i<=N-1;i++)
        x[i]=i;

/* array x[i] is ready now to be "bit-reversed" */

    j=0;

    for (i=1; i<N-1; i++)
        {

/*
** N=2^L
** n1 is calculated to be powers of 2, and in this order:
** 2^(L-1), 2^(L-2), 2^(L-3),...
**
**
n1=N/2;

/*
** the "while" loop below is flipping the bits "1" at the left,
** of the binary expression of "previous j" by subtracting
** powers of two. Starts from the left-most "1".
**
** This flipping continues until the first "0"
** on the left is met in the binary expression of "previous j"
**
** as i->i+1 then j=(BitReverse of i)-> j' .And j' is to be
** determined from the "previous j"
** N=2^L
** n1 is calculated to be powers of 2, and in this order:

```

```

** 2^(L-1), 2^(L-2), 2^(L-3),...
**
*/

    while (j>=n1)
        {
            j=j-n1;
            n1=n1/2;
        }

/*
** Flip now the first "0" on the left to "1" .
**
*/

    j=j+n1;

/*
** "j" will be accepted now only if this is not a reversal of
** a previous bit-reversal.
**
*/

    if (i<j)
        {
/*
** Swap the numbers stored in x[i] with the number stored in x[j]
**
*/
            t1=x[i];
            x[i]=x[j];
            x[j]=t1;
        }
    }

/* print the "in place bit-reversed" data */

    for (i=0; i<N; i++)
        printf("i=%2d    x[%2d]=%6.2f \n", i, i, x[i]);

    return(0);

```

}

Problem 1.) Do the bit-reversing in *reverse* order. Say you have $\{x_0, x_1, x_2, x_3\}$. Jones's code bit-reverses starting from the first element and obtains $\{x_0, x_1 \rightarrow x_2, x_2 \rightarrow x_1, x_3\}$. Your code should start from the last element, and then obtain $\{x_3, x_2 \rightarrow x_1, x_1 \rightarrow x_2, x_0\}$. Explain your theory and the code.

Now that the bit-reversing and its code understood, the full fft C-code is given below [5]. Go over the code and see how FFT is built up starting from the lowest level. Several helpful comments by assistant Ismail ARI are also inserted. Note that n, m (our N and L) and the signal $z = x + iy$ must be given in the calling program. Try with N=8, and various z signals.

```
/* **** */
/* fft.c */
/* (c) Douglas L. Jones */
/* University of Illinois at Urbana-Champaign */
/* January 19, 1992 */
/* */
/* fft: in-place radix-2 DIT DFT of a complex input */
/* */
/* input: */
/* n: length of FFT: must be a power of two */
/* m: n = 2**m */
/* input/output */
/* x: double array of length n with real part of data */
/* y: double array of length n with imag part of data */
/* */
/* Permission to copy and use this program is granted */
/* under a Creative Commons "Attribution" license */
/* http://creativecommons.org/licenses/by/1.0/ */
/* **** */
```

```

fft(n,m,x,y)
int n,m;
double x[],y[];
{
int i,j,k,n1,n2;
double c,s,e,a,t1,t2;

j = 0; /* bit-reverse */
n2 = n/2;
for (i=1; i < n - 1; i++)
{
n1 = n2;
while ( j >= n1 )
{
j = j - n1;
n1 = n1/2;
}
j = j + n1;

if (i < j)
{
t1 = x[i];
x[i] = x[j];
x[j] = t1;
t1 = y[i];
y[i] = y[j];
y[j] = t1;
}
}

n1 = 0; /* FFT */
n2 = 1;

/*loop over stages_ ARI */
for (i=0; i < m; i++)

```

```

{
  n1 = n2;
  n2 = n2 + n2;
  e = -6.283185307179586/n2;
  a = 0.0;

  /* loop over distinct W factors _ ARI */
  for (j=0; j < n1; j++)
  {
    c = cos(a);
    s = sin(a);
    a = a + e;

    /*loop over different instances of the same W _ ARI */
    for (k=j; k < n; k=k+n2)
    {
      t1 = c*x[k+n1] - s*y[k+n1];
      t2 = s*x[k+n1] + c*y[k+n1];
      x[k+n1] = x[k] - t1;
      y[k+n1] = y[k] - t2;
      x[k] = x[k] + t1;
      y[k] = y[k] + t2;
    }
  }
}

return;
}

```

Next item in the agenda is to do it in the MATLAB with MEX files. See your Lab-sheets.

Problem 2.) Take the signal $x = (a, b, c, d)$ as input, and do not bit-reverse. Show that in the fft-way of computing, the Fourier transform does not come out correctly, and does not come out in the bit-reversed form either. (Hint: In Problem Set 5 of the Fourier notes, in problem 5, the Fourier transform of x was given explicitly. Skipping the bit-reversing, means an

interchange of the elements ($b \leftrightarrow c$) in the expressions of \hat{X} there. And the correct transform cannot be recovered simply from this output.

II) recursive programming

Here is a very crude structure of the fft function recursive code [6, 7]:

```
fft(signal y)
{
    if (length of signal y equals one)
        return;

    divide signal y into two half-length signals, y_even and y_odd;
    fft(y_e);
    fft(y_o);
    merge the two half length signals using Eq.(14);
    U=1; /* to generate powers of W */
    for (k=0 ; k<n/2 ; k++)
    {
        (first-half element) + U* (second-half element);
        U=U*W_n;
    }
    /* to get the values for k=n/2 to n-1, replace k by n-k */
}
```

Again, see your Lab-sheets.

Bibliography

- [1] J.W.Cooley and J.W.Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(2):297–301, April 1965.
- [2] Albert Boggess and Francis J. Narcowich. *A First Course in Wavelets with Fourier Analysis*. Prentice-Hall,Inc., 2001.
- [3] F. Richard Moore. *Elements of Computer Music*. Prentice-Hall Inc., 1990.
- [4] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 1999.
- [5] Douglas Jones. 'decimation-in time (dit) radix-2 fft'. World Wide Web, <http://cnx.rice.edu/content/m12016/latest>, 2004. The Connexions Project module m12016.
- [6] Engineering Productivity Tools Ltd. 'the radix 2 decimation in time (dit) algorithm.'. World Wide Web, <http://www.eptools.com/tn/T0001/PT04.HTM>, 1999. From the FFT demystified.
- [7] Ken Steiglitz. *A Digital Signal Processing Primer*. Addison-Wesley Publishing Company, Inc., 1996.