哈!经过连续几个晚上的奋战,终于弄懂了 FFT 推导过程及实现! Happy☺
基2 FFT 总的思想是将输入信号对半分割,再对半分割,再再对半分割(以下省略10000个再再...☺)
直至分割到2点.

两点 DFT 简化

假设输入为 x[0],x[1];输出为 X[0],X[1]. 伪代码如下 :

```
// ------------------------------------------------------------
#define     N       2
#define     PI      3.1415926

// ------------------------------------------------------------
int i, j
for(i=0, X[i]=0.0; i<N; i++)
    for(j=0; j<N; j++)
        X[i] += x[j] * ( cos(2*PI*i*j/N) - sin(2*PI*i*j/N) );
```

注意到(我想 Audio 编解码很多时候都是对 cos,sin 进行优化!)

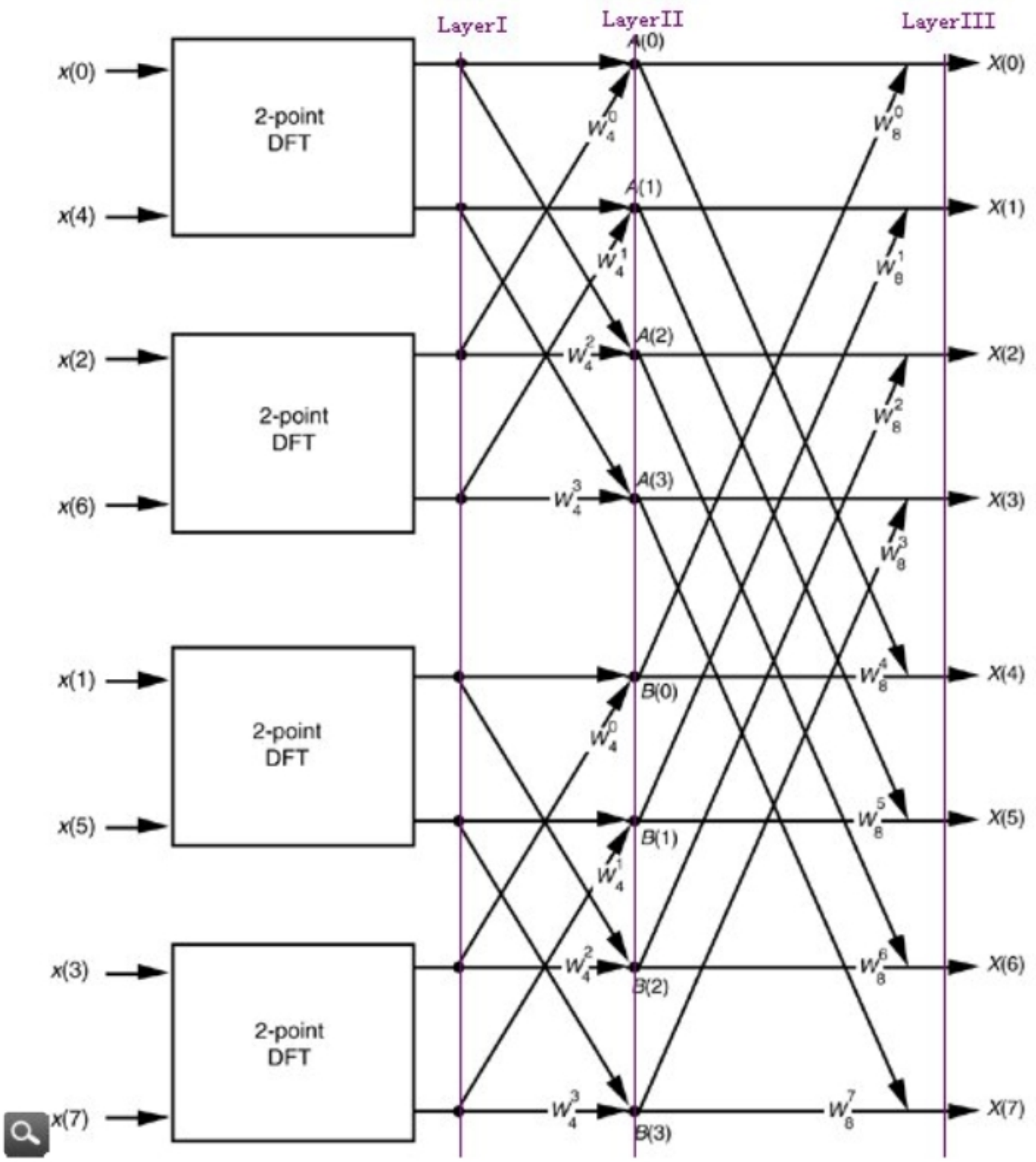
	j=0	j=1
i=0	cos 1 sin 0 tw 1	cos 1 sin 0 tw 1
i=1	cos 1 Sin 0 tw 1	cos -1 sin 0 tw -1

X[0] = x[0]*(1-0) + x[1]*(1-0) = x[0] + 1*x[1];
X[1] = x[0]*(1-0) + x[1]*(-1-0) = x[0] - 1*x[1];

这就是单个2点蝶形算法.

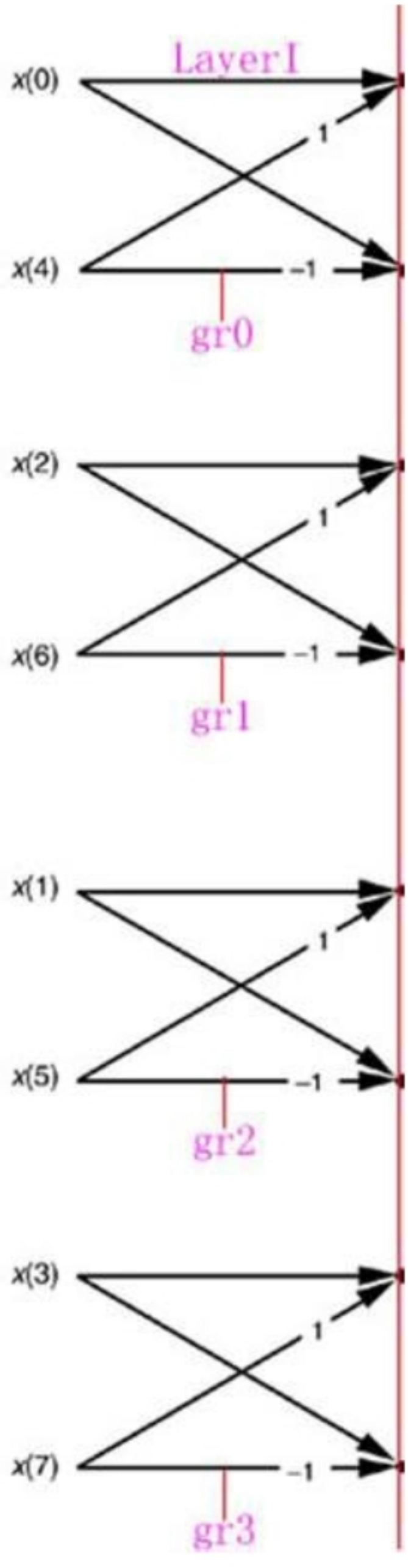
FFT 实现流程图分析(N=8,以8点信号为例)

FFT implementation of an 8-point DFT as two 4-point DFTs and four 2-point DFTs



8点 FFT 流程图(Layer 表示层, gr 表示当前层的颗粒)

下面以 LayerI 为例.



LayerI 部分,具有4个颗粒,每个颗粒2个输入

(注意2个输入的来源,由时域信号友情提供,感谢感谢☺)

我们将输入 x[k]分为两部分 x_r[k], x_i[k].具有实部和虚部,时域信号本没有虚部的,因此可以让 x_i[k]为0.那么为什么还要画蛇添足分为实部和虚部呢?这是因为 LayerII, LayerIII 的输入是复数,为了编码统一而强行分的.当然你编码时可以判断当前层是否为1来决定是否分.但是我想每个

人最后都会倾向分的.

旋转因子 tw = cos(2*PI*k/N)-j*sin(2*PI*k/N);也可以分为实部和虚部,令其为 tw_r, tw_i;

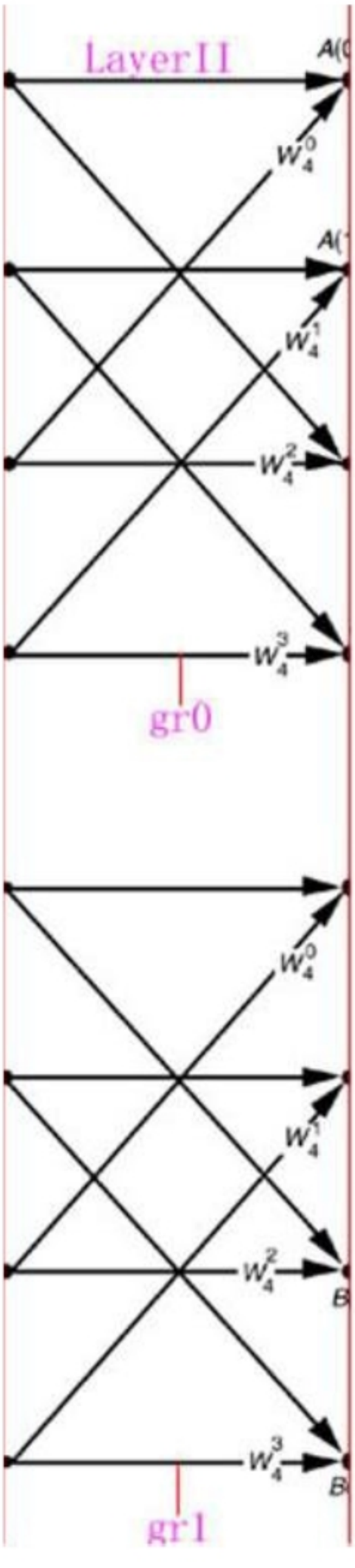
则 tw = tw_r - j*tw_i;

X[k] = (x_r[k] + j*x_i[k]) + (tw_r–j*tw_i) * (x_r[k+N/2]+j*x_i[k+N/2])

则

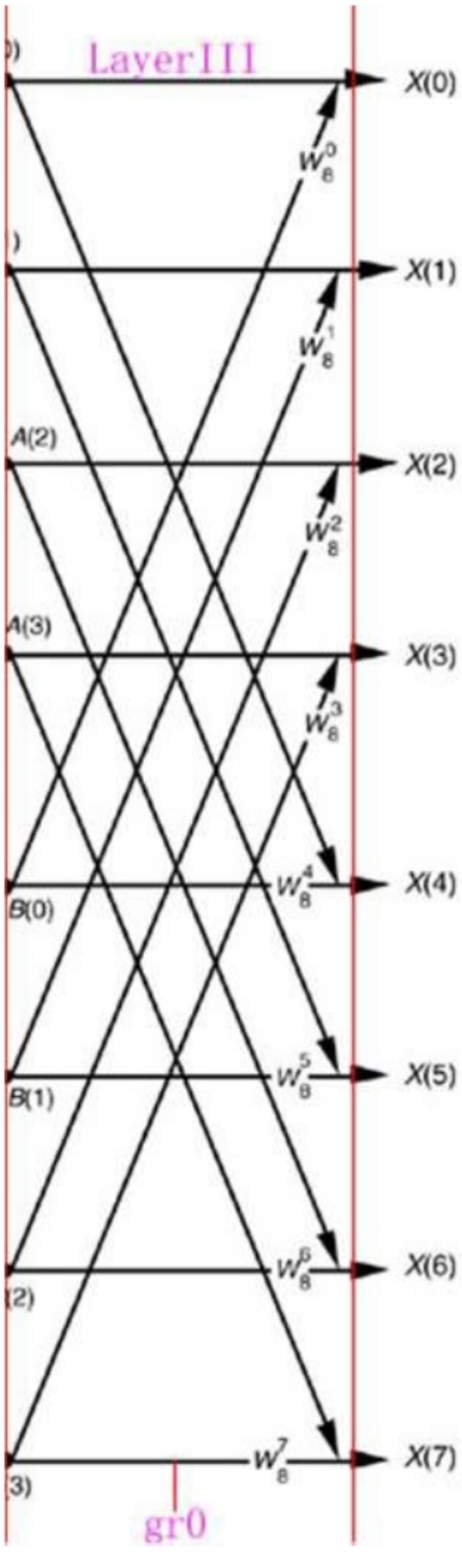
X_R[k] = x_r[k] + tw_r*x_r[k+N/2] + tw_i*x_i[k+N/2];

X_I[k] = x_i[k] - tw_i*x_r[k+N/2] + tw_r*x_i[k+N/2];



LayerII 部分,具有2个颗粒,每个颗粒4个输入

(注意4个输入的来源,由 LayerI 友情提供,感谢感谢☺)



LayerIII 部分,具有1个颗粒,每个颗粒8个输入

(注意8个输入的来源,由 LayerII 友情提供,感谢感谢☺)

LayerI, LayerII, LayerIII 从左往右,蝶形信号运算流非常明显!

假令输入为 x[k], x[k+N/2],输出为 X[k], X[k+N/2]. x[k]分解为 x_r[k], x_i[k]部分
则该蝶形运算为
X[k]
= (x_r[k]-j*x_i[k]) + (x_r[k+N/2]-j*x_i[k+N/2])*(cos(2*PI*k/N)-j*sin(2*PI*k/N));
再令 cos(2*PI*k/N)为 tw1, sin(2*PI*k/N)为 tw2则
X[k] = (x_r[k]-j*x_i[k]) + (x_r[k+N/2]-j*x_i[k+N/2])*(tw1-j*tw2);

X_R[k] = x_r[k] + x_r[k+N/2]*tw1 - x_i[k+N/2]*tw2;
X_I[K] = x_i[k]

```
x_r[k] = x_r[k] + x_r[k+b]*tw1 + x_i[k+b]*tw2;
x_i[k] = x_i[k] - x_r[k+b]*tw2 + x_i[k+b]*tw1;
```

譬如8点输入 x[8]

1. 先分割成2部分: x[0], x[2], x[4], x[6]和 x[1], x[3], x[5], x[7]
2. 信号 x[0], x[2], x[4], x[6]再分割成 x[0], x[4]和 x[2], x[6]
   信号 x[1], x[3], x[5], x[7]再分割成 x[1], x[5]和 x[3], x[7]

3.无法分割了,已经分割成2点了☺.

如上图:

在 LayerI 的时候,我们是对2点进行 DFT.(一共4次 DFT)

输入为　　x[0]&x[4];　　x[2]&x[6];　　x[1]&x[5];　　x[3]&x[7]

输出为　　y[0],y[1]; Y[2],y[3]; Y[4],y[5]; Y[6],y[7];

流程:

I.希望将输入直接转换为 **x[0], x[4], x[2], x[6], x[1], x[5], x[3], x[7]**的顺序

II.对转换顺序后的信号进行**4**次 **DFT**

步骤 I 代码实现

```
/**
 *反转算法. 这个算法效率比较低!先用起来在说,之后需要进行优化.
 */
static void bitrev(void )
{
    int     p=1, q, i;
    int     bit_rev[ N ];
    float   xx_r[ N ];
```

```
bit_rev[ 0 ] = 0;
while( p < N )
{
    for(q=0; q<p; q++)
    {
        bit_rev[ q ]       = bit_rev[ q ] * 2;
        bit_rev[ q + p ] = bit_rev[ q ] + 1;
    }
    p *= 2;
    }
    for(i=0; i<N; i++)    xx_r[ i ] = x_r[ i ];
    for(i=0; i<N; i++)    x_r[i] = xx_r[ bit_rev[i] ];
}
// ----------------------此刻序列 x 重排完毕----------------------
```

步骤 II 代码实现

```
int j;
float TR;       //临时变量
float tw1; //旋转因子
/*两点 DFT */
for(k=0; k<N; k+=2)
{
    //两点 DFT 简化告诉我们 tw1=1
    TR = x_r[k]; // TR 就是 A, x_r[k+b]就是 B.
    x_r[k]    = TR + tw1*x_r[k+b];
    x_r[k+b] = TR - tw1*x_r[k+b];
}
```

在 LayerII 的时候,我们希望得到 z,就需要对 y 进行 DFT.

y[0],y[2]; y[1],y[3]; y[4],y[6]; y[5],y[7];

z[0], z[1]; z[2],z[3]; z[4],z[5]; z[6],z[7];

在 LayerIII 的时候,我们希望得到 v,就需要对 z 进行 DFT.

z[0],z[4]; z[1],z[5]; z[2],z[6]; z[3],z[7];

v[0],v[1]; v[2],v[3]; v[4],v[5]; v[6],v[7];

准备

令输入为 x[s], x[s+N/2],输出为 y[s], y[s+N/2]
这个 N 绝对不是上面的8,这个 N 是当前颗粒的输入样本总量
对于 LayerI 而言 N 是2;对于 LayerII 而言 N 是4;对于 LayerIII 而言 N 是8

复数乘法:(a+j*b) * (c+j*d)
实部= a*c – bd;
虚部= ad + bc;

旋转因子:

实现(C 描述)

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
//#include "complex.h"

// ------------------------------------------------------------------
#define   N       8 //64
#define   M       3 //6       //2^m=N
#define   PI      3.1415926
// ------------------------------------------------------------------

float   twiddle[N/2] = {1.0, 0.707, 0.0, -0.707};
float   x_r[N] = {1, 1, 1, 1, 0, 0, 0, 0};
float   x_i[N];                                 //N=8
/*
float   twiddle[N/2] = {1,          0.9951, 0.9808, 0.9570, 0.9239, 0.8820, 0.8317, 0.7733,
                                    0.7075, 0.6349, 0.5561,     0.4721,     0.3835,     0.2912,     0.1961,
0.0991,
                                    0.0000,-0.0991,-0.1961,-0.2912,-0.3835,-0.4721,-0.5561,-0.6349,
                                   -0.7075,-0.7733, 0.8317,-0.8820,-0.9239,-0.9570,-0.9808,-0.9951};
//N=64
float   x_r[N]={1,1,1,1,1,1,1,1,
                        1,1,1,1,1,1,1,1,
                        1,1,1,1,1,1,1,1,
                        1,1,1,1,1,1,1,1,
```

```
                                        1,1,1,1,1,1,1,1,
                                        0,0,0,0,0,0,0,0,
                                        0,0,0,0,0,0,0,0,
                                        0,0,0,0,0,0,0,0,
                                        0,0,0,0,0,0,0,0,};
float    x_i[N];
*/
FILE *fp;

// ---------------------------------- func ----------------------------------
/**
 *初始化输出虚部
 */
static void fft_init(void )
{
    int i;
    for(i=0; i<N; i++)    x_i[i] = 0.0;
}

/**
 *反转算法.将时域信号重新排序.
 *这个算法有改进的空间
 */
static void bitrev(void )
{
    int     p=1, q, i;
    int     bit_rev[ N ]; //
    float    xx_r[ N ];     //

    bit_rev[ 0 ] = 0;
    while( p < N )
    {
        for(q=0; q<p; q++)
        {
            bit_rev[ q ]       = bit_rev[ q ] * 2;
            bit_rev[ q + p ] = bit_rev[ q ] + 1;
        }
        p *= 2;
    }

    for(i=0; i<N; i++)    xx_r[ i ] = x_r[ i ];

    for(i=0; i<N; i++)    x_r[i] = xx_r[ bit_rev[i] ];
}
```

```
/* ------------ add by sshc625 ------------ */
static void bitrev2(void )
{
    return ;
}


/* */
void display(void )
{
    printf("\n\n"`;
    int    i;
    for(i=0; i<N; i++)
        printf("%f\t%f\n", x_r[i], x_i[i]);
}

/**
 *
 */
void fft1(void )
{ fp = fopen("log1.txt","a+"`;
    int       L, i, b, j, p, k, tx1, tx2;
float    TR, TI, temp; //临时变量
float    tw1, tw2;

/*深 M.对层进行循环. L 为当前层,总层数为 M. */
for(L=1; L<=M; L++)
{
    fprintf(fp,"----------Layer=%d----------\n", L);
    /* b 的意义非常重大,b 表示当前层的颗粒具有的输入样本点数  */
    b = 1;
    i = L - 1;
    while(i > 0)
    {
        b *= 2;
        i--;
    }

// --------------是否外层对颗粒循环,内层对样本点循环逻辑性更强一些呢! ----------
    /*
     * outter 对参与 DFT 的样本点进行循环
     * L=1,循环了1次(4个颗粒,每个颗粒2个样本点)
     * L=2,循环了2次(2个颗粒,每个颗粒4个样本点)
```

```
 * L=3,循环了4次(1个颗粒,每个颗粒8个样本点)
 */
for(j=0; j<b; j++)
{
    /*求旋转因子 tw1 */
    p = 1;
    i = M - L; // M 是为总层数, L 为当前层.
    while(i > 0)
    {
        p = p*2;
        i--;
    }
    p    = p * j;
    tx1 = p % N;
    tx2 = tx1 + 3*N/4;
    tx2 = tx2 % N;
    // tw1是 cos 部分,实部; tw2是 sin 部分,虚数部分.
    tw1 = ( tx1>=N/2)? -twiddle[tx1-N/2]    : twiddle[ tx1 ];
    tw2 = ( tx2>=N/2)? -twiddle[tx2-(N/2)] : twiddle[tx2];

    /*
     * inner 对颗粒进行循环
     * L=1,循环了4次(4个颗粒,每个颗粒2个输入)
     * L=2,循环了2次(2个颗粒,每个颗粒4个输入)
     * L=3,循环了1次(1个颗粒,每个颗粒8个输入)
     */
    for(k=j; k<N; k=k+2*b)
    {
        TR = x_r[k]; // TR 就是 A, x_r[k+b]就是 B.
        TI = x_i[k];
        temp = x_r[k+b];
        /*
          *如果复习一下  (a+j*b)(c+j*d)两个复数相乘后的实部虚部分别是什么
          *就能理解为什么会如下运算了,只有在 L=1时候输入才是实数,之后层的
          *输入都是复数,为了让所有的层的输入都是复数,我们只好让 L=1时候的
          *输入虚部为0
          * x_i[k+b]*tw2是两个虚数相乘
         */
        fprintf(fp,"tw1=%f, tw2=%f\n", tw1, tw2);
        x_r[k]    = TR + x_r[k+b]*tw1 + x_i[k+b]*tw2;
        x_i[k]    = TI - x_r[k+b]*tw2 + x_i[k+b]*tw1;

        x_r[k+b] = TR - x_r[k+b]*tw1 - x_i[k+b]*tw2;
        x_i[k+b] = TI + temp*tw2        - x_i[k+b]*tw1;
```

```
                    fprintf(fp,"k=%d, x_r[k]=%f, x_i[k]=%f\n", k, x_r[k], x_i[k]);
                    fprintf(fp,"k=%d, x_r[k]=%f, x_i[k]=%f\n", k+b, x_r[k+b], x_i[k+b]);
                }//
            }//
        }//
    }

    /**
     * ------------ add by sshc625 ------------
     *该实现的流程为
     * for( Layer )
     *       for( Granule )
     *             for( Sample )
     *
     *
     *
     *
     */
    void fft2(void )
    {   fp = fopen("log2.txt","a+");
        int       cur_layer, gr_num, i, k, p;
        float    tmp_real, tmp_imag, temp;    //临时变量,记录实部
        float    tw1, tw2;//旋转因子,tw1为旋转因子的实部 cos 部分, tw2为旋转因子的虚部 sin 部分.

        int     step;       //步进
        int     sample_num;   //颗粒的样本总数(各层不同,因为各层颗粒的输入不同)

        /*对层循环 */
        for(cur_layer=1; cur_layer<=M; cur_layer++)
        {
            /*求当前层拥有多少个颗粒(gr_num) */
            gr_num = 1;
            i = M - cur_layer;
            while(i > 0)
            {
                i--;
                gr_num *= 2;
            }

            /*每个颗粒的输入样本数 N' */
            sample_num      = (int)pow(2, cur_layer);
            /*步进.步进是 N'/2 */
            step            = sample_num/2;
```

```
/* */
k = 0;

/*对颗粒进行循环  */
for(i=0; i<gr_num; i++)
{
        /*
          *对样本点进行循环,注意上限和步进
         */
for(p=0; p<sample_num/2; p++)
{
    //旋转因子,需要优化...
    tw1 = cos(2*PI*p/pow(2, cur_layer));
    tw2 = -sin(2*PI*p/pow(2, cur_layer));

    tmp_real = x_r[k+p];
    tmp_imag = x_i[k+p];
    temp = x_r[k+p+step];

    /*(tw1+jtw2)(x_r[k]+jx_i[k])
     *
     * real : tw1*x_r[k] - tw2*x_i[k]
     * imag : tw1*x_i[k] + tw2*x_r[k]
     *我想不抽象出一个
     * typedef struct {
     * double real; //实部
     * double imag; //虚部
     * } complex;以及针对 complex 的操作
     *来简化复数运算是否是因为效率上的考虑!
     */

/*蝶形算法  */
x_r[k+p]    = tmp_real + ( tw1*x_r[k+p+step] - tw2*x_i[k+p+step] );
x_i[k+p]    = tmp_imag + ( tw2*x_r[k+p+step] + tw1*x_i[k+p+step] );
/* X[k] = A(k)+WB(k)
 * X[k+N/2] = A(k)-WB(k)的性质可以优化这里*/
//旋转因子,需要优化...
tw1 = cos(2*PI*(p+step)/pow(2, cur_layer));
tw2 = -sin(2*PI*(p+step)/pow(2, cur_layer));
x_r[k+p+step] = tmp_real + ( tw1*temp - tw2*x_i[k+p+step] );
x_i[k+p+step] = tmp_imag + ( tw2*temp + tw1*x_i[k+p+step] );

printf("k=%d, x_r[k]=%f, x_i[k]=%f\n", k+p, x_r[k+p], x_i[k+p]);
```

```
                printf("k=%d, x_r[k]=%f, x_i[k]=%f\n", k+p+step, x_r[k+p+step], x_i[k+p+step]);
            }
            /*开跳!:) */
            k += 2*step;
        }
    }
}
/*
 *后记:
 *究竟是颗粒在外层循环还是样本输入在外层,好象也差不多,复杂度完全一样.
 *但以我资质愚钝花费了不少时间才弄明白这数十行代码.
 *从中我发现一个于我非常有帮助的教训,很久以前我写过一部分算法,其中绝大多数都是递归.
 *将数据量减少,减少再减少,用归纳的方式来找出数据量加大代码的规律
 *比如 FFT
 * 1.先写死 LayerI 的代码;然后再把 LayerI 的输出作为 LayerII 的输入,又写死代码; .....
 *      大约3层就可以统计出规律来.这和递归也是一样,先写死一两层,自然就出来了!
 * 2.有的功能可以写伪代码,不急于求出结果,降低复杂性,把逻辑结果定出来后再添加.
 *      比如旋转因子就可以写死,就写1.0.流程出来后再写旋转因子.
 *寥寥数语,我可真是流了不少汗! Happy!
 */


void dft(void )
{
    int     i, n, k, tx1, tx2;
    float   tw1,tw2;
    float   xx_r[N],xx_i[N];

    /*
     * clear any data in Real and Imaginary result arrays prior to DFT
     */
    for(k=0; k<=N-1; k++)
        xx_r[k] = xx_i[k] = x_i[k] = 0.0;

    // caculate the DFT
    for(k=0; k<=(N-1); k++)
    {
        for(n=0; n<=(N-1); n++)
        {
            tx1 = (n*k);
            tx2 = tx1+(3*N)/4;
            tx1 = tx1%(N);
            tx2 = tx2%(N);
            if(tx1 >= (N/2))
```

```
            tw1 = -twiddle[tx1-(N/2)];
        else
            tw1 = twiddle[tx1];
        if(tx2 >= (N/2))
            tw2 = -twiddle[tx2-(N/2)];
        else
            tw2 = twiddle[tx2];
        xx_r[k] = xx_r[k]+x_r[n]*tw1;
        xx_i[k] = xx_i[k]+x_r[n]*tw2;
            }
            xx_i[k] = -xx_i[k];
        }
        // display
        for(i=0; i<N; i++)
            printf("%f\t%f\n", xx_r[i], xx_i[i]);
}

// ------------------------------------------------------------------------
int main(void )
{
    fft_init( );
    bitrev( );
    // bitrev2( );
    //fft1( );
    fft2( );
    display( );

    system("pause" );
    // dft();
    return 1;
}
```