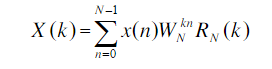
**基2的DIT-FFT算法实验报告**

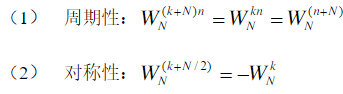
1. **算法原理**

本实验实现基2的DIT-FFT算法，基于分治法的思想，将时域上的快速傅里叶变换用C++语言实现。

对于DFT变换，表达式为

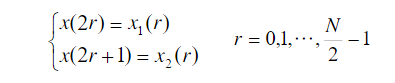


计算一个X（K）需要N次复数乘法和N-1次复数加法。算出全部的N点X（k）共需要N^2次复数乘法和N（N-1）次复数加法。即计算量是O（N^2）。如果可以将DFT分解成若干组合，利用蝶型运算，可以达到分而治之的目的。我们利用旋转因子W的若干性质：

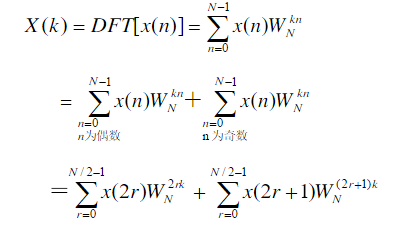


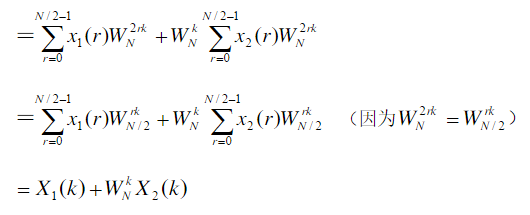
可以化简运算。

对于时间抽取的FFT即DIT-FFT将输入序列x（n）按奇偶分为两组。



那么DFT也可以相应地表示为：

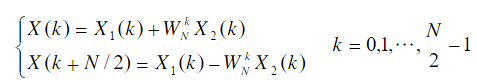




同理，对于后半部分，有

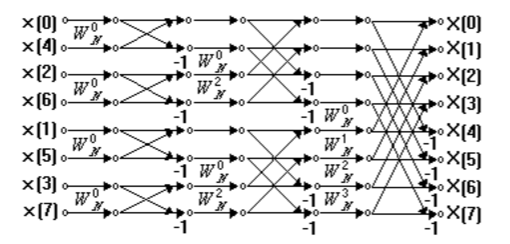


整理可以得到



这个式子体现出了分治法的思想，即通过蝶型运算来简化复数运算过程。这里面的复数乘法只涉及到了旋转因子与X的复数乘法。

最后可以用蝶形图如下表示



这里x（n）为输入的时域序列，为复数。X（n）为输出的频域序列，也为复数。

1. **问题描述**

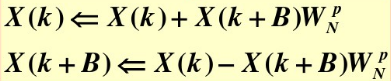
本实验要解决的问题是：给定一组复数输入x（n），通过DIT-FFT算法，输出X（n）。请注意，这里面涉及到如下几个问题：

1. **输入顺序问题：**从蝶形图上不难发现，对于DIT-FFT算法，输出是按序输出，但输入并不是有序的，我们需要根据二进制码的倒读原则来调整输入顺序，稍后我们介绍这个算法。
2. **旋转因子的系数问题：**注意到，旋转因子的底参数为输入的宽度，而指数根据蝶型运算的层级不同而不同，确定旋转因子的参数是算法的关键。
3. **输入对齐问题：**注意到蝶形图要求输入为2的整数次方，若不满足这个条件，可以通过补0来达到这个条件。为了精简程序，这个过程在程序中没有体现出来。
4. **算法过程分析**

FFT的每级输入都是N个复数，两两构成蝶形，运算得到另外N个复数（从上面的蝶形图很容易看出）。这表明，我们可以反复利用同一个宽度为N的数组，两两复数蝶型运算之后，再保存到同样的数据位当中，以便下一级的运算。

算法的关键是确定每一个蝶形运算中旋转因子的参数，完成参数的确定之后，只需要进行复数的乘法、加法运算即可完成蝶形运算。

定义步长B可以通过蝶型运算的级数L来确定，B表示了本级参与蝶型运算中两个运算单元的间距。因此我们可以将运算模块表示为



完成了一共M级（N = 2^M）的蝶型运算后，x序列中的复数值即为输出序列X。

算法流程图可以简单的表示如下：

1. **算法复杂度分析**

对于每一个蝶形运算单元，完成这次运算共需要两次复数加法和一次复数乘法（旋转因子可以复用）。观察蝶形图可以看到，运算共有M级，每一级一定会有N/2个蝶形运算单元。

不难计算，共需要( NlogN ) / 2 次复数乘法和NlogN次复数加法，则算法的时间复杂度为O（NlogN）。

算法的空间复杂度分析也很简单，由于复用了寄存器（宽度为N的数组），空间复杂度为O（N）。

对比原来的DFT算法的O（N^2），利用分治法的计算速度提升较多。

1. **可能的改进**

本实验的算法在下面几个部分仍存在缺陷

1. **输入对齐问题：**如上所述，虽然这个问题不难解决，但确实存在，且根据采样点多少不同，这个问题在某些时候需要考虑进去。
2. **缺少合理的输入样本：**虽然本算法实现的背景是信号的采集，但是为了重点分析算法过程，本实验的输入采用了随机数的方式输入，这在实际的信号应用中也是不合理的。
3. **算法实现**
4. **数据结构和函数**

struct cpx // 复数结构

{

double re;

double im;

};

typedef struct cpx CPX;

复数结构包含了实数与虚数，模拟信号的采样。

CPX add\_cpx(CPX x,CPX y) // 复数加法

CPX sub\_cpx(CPX x,CPX y) // 复数减法

CPX mul\_with\_rtt(CPX x,int exp\_W,int width\_W) // 由于采用FFT，只需要和旋转因子W做复数乘法

void print\_cpx(void) // 打印函数

void inv\_input(int len) // 由于是DIT-FFT 需要使用算法对输入进行倒置译码

void DIT\_FFT(int len,int log\_len) // DIT-FFT过程

1. **算法核心流程**

这里重点讲DIT\_FFT函数的实现，首先对于蝶型运算单元，模拟代码如下

add2 = mul\_with\_rtt(X[k+step],exp\_w,MAX\_N);

add1 = X[k];

X[k] = add\_cpx(add1,add2); // 蝶型运算 ，步进为step，实质上是奇偶分治

X[k+step] = sub\_cpx(add1,add2);

这里用步长确定了参与蝶型运算的二元组，用函数来实现复数的乘法和加法。

函数有三重循环，第一重循环

for(int L = 1;L<=log\_len;L++)

是对级数的步进，并根据级数计算步长step；

第二重循环

for(int j = 0;j<=step - 1;j++)

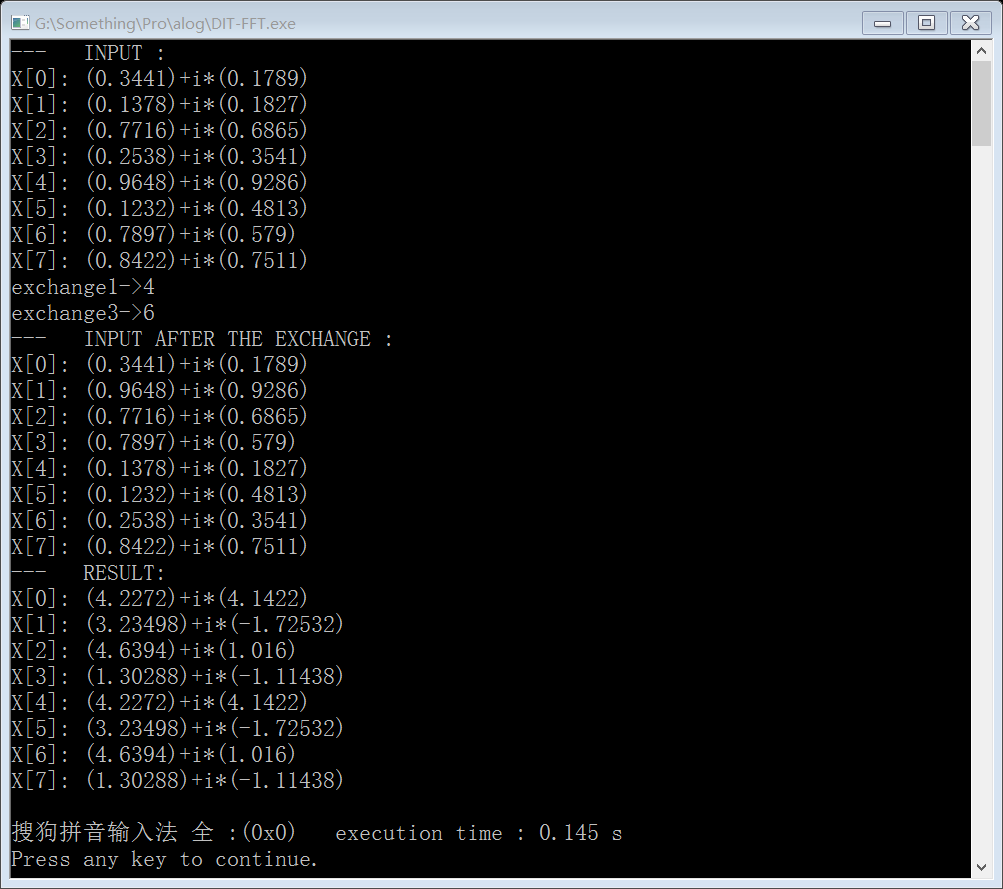
是对步长的步进，并求出旋转因子的参数；

第三重循环

for(int k = j ; k<=MAX\_N -1;k+=2\*step)

用k来索引数组X，旨在两两一对遍历X数组。

1. **输出结果示例与说明**



这是在默认MAX\_M = 3的情况下的实验结果。输入为8个复数，即采样序列，输出同样为8个复数。注意到在使用DIT-FFT之前，倒序算法发挥了其应有的功效，该算法处理了输入顺序，将X[1]和X[4]互换，X[3]和X[6]互换。

请注意，这里的输入均为随机数实现，无任何实际的物理意义。

1. **总结和反思**

本次编码是对一个不了解的领域：信号处理，做了研究和探讨。本次实验大量的功夫都花在在网上找资料、学习资料的过程上，这个过程让我受益匪浅，从本质上了解了另一个领域上的算法应用。本次实验设计到的两个算法：FFT和二进制倒置算法都有一定的难度和新奇特点，尤其是FFT本身，对于旋转因子W的处理，如果没有良好的复数计算功底，很难顺利地推导出来（我就推导了很长时间）。不过欣慰的是，当弄透彻了FFT原理之后，算法的编码就不再困难。

其实，最重要的是这次实验强化了我的自学能力。虽然算法本质是分治法，但是算法本身几乎是完全陌生的，这也就需要你利用资源整理、分析和解决问题。这个过程是之前不曾历练到的，之前的算法练习大多是一个基本熟悉的问题，然后在此基础上进行编码和分析。

需要说明的是，由于理论部分借鉴了互联网的资料，本实验报告的算法原理部分非原创，除此之外均为独立完成。

#include<iostream>

#include<cmath>

#include<cstdlib>

#include<ctime>

using namespace std;

#define MAX\_M 3

#define MAX\_N 8

#define RAND 9999

#define PI 3.141593

struct cpx // 复数结构

{

double re;

double im;

};

typedef struct cpx CPX;

CPX X[MAX\_N]; // 寄存器数组，既是输入也是输出

CPX add\_cpx(CPX x,CPX y) // 复数加法

{

CPX temp;

temp.im = x.im + y.im;

temp.re = x.re + y.re;

return temp;

}

CPX sub\_cpx(CPX x,CPX y) // 复数减法

{

CPX temp;

temp.im = x.im - y.im;

temp.re = x.re - y.re;

}

CPX mul\_with\_rtt(CPX x,int exp\_W,int width\_W) // 由于采用FFT，只需要和旋转因子W做复数乘法

{

CPX temp;

temp.re = x.re \* cos(2.0 \* PI \* exp\_W / MAX\_N) + x.im \* sin(2.0 \* PI \* exp\_W / MAX\_N);

temp.im = x.im \* cos(2.0 \* PI \* exp\_W / MAX\_N) - x.re \* sin(2.0 \* PI \* exp\_W / MAX\_N);

return temp;

}

void print\_cpx(void) // 打印函数

{

for(int i = 0; i <= MAX\_N - 1 ; i++)

cout<<"X["<<i<<"]: "<<"("<<X[i].re<<")"<<"+"<<"i\*("<<X[i].im<<")"<<endl;

}

void inv\_input(int len) // 由于是DIT-FFT 需要使用算法对输入进行倒置译码

{

CPX temp;

int bit\_w = MAX\_N / 2;

int step;

for(int i = 1 ; i<= len-2 ; i++)

{

if( i < bit\_w )

{

temp = X[i];

X[i] = X[bit\_w];

X[bit\_w] = temp;

cout<<"exchange"<<i<<"->"<<bit\_w<<endl;

}

step = MAX\_N / 2;

jump:

if(bit\_w < step)

{

bit\_w += step;

}

else

{

bit\_w -= step;

step /= 2;

goto jump;

}

}

}

void DIT\_FFT(int len,int log\_len) // DIT-FFT过程

{

int step; // 当前步进

CPX add1,add2; // 蝶型运算的两个加法元

int exp\_w; // 旋转因子W的指数

for(int L = 1;L<=log\_len;L++)

{

step = (int)pow(2,L-1);

for(int j = 0;j<=step - 1;j++)

{

exp\_w = j \* (int)pow(2,log\_len-L);

for(int k = j ; k<=MAX\_N -1;k+=2\*step)

{

add2 = mul\_with\_rtt(X[k+step],exp\_w,MAX\_N);

add1 = X[k];

X[k] = add\_cpx(add1,add2); // 蝶型运算 ，步进为step，实质上是奇偶分治

X[k+step] = sub\_cpx(add1,add2);

}

}

}

}

int main(void)

{

srand(time(NULL));

cout<<"--- INPUT :"<<endl;

for(int i = 0 ;i<=MAX\_N - 1 ;i++) // 随机产生采样值

{

X[i].re = rand()%(RAND + 1) / (double)(RAND + 1);

X[i].im = rand()%(RAND + 1) / (double)(RAND + 1);

}

print\_cpx();

inv\_input(MAX\_N); // 倒置译码

cout<<"--- INPUT AFTER THE EXCHANGE :"<<endl;

print\_cpx();

DIT\_FFT(MAX\_N,MAX\_M); // 基2的DIT-FFT

cout<<"--- RESULT: "<<endl;

print\_cpx();

return 0;

}