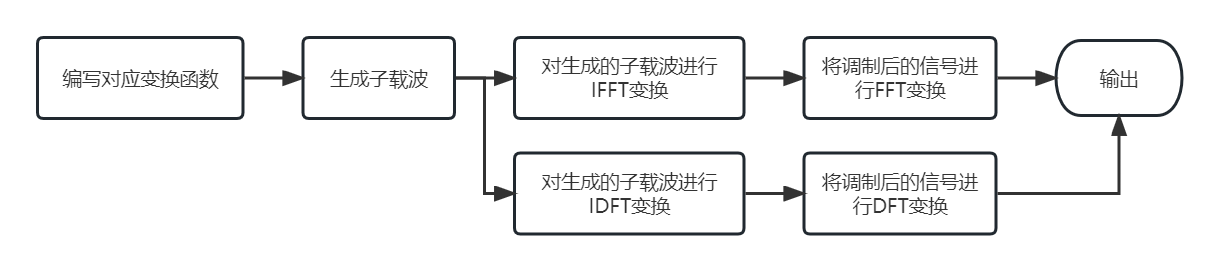
**频率分析与离散傅氏变换实验报告**

#### 1. 实验步骤



1. 根据FFT定义、IFFT定义以及DFT定义分别编写对应变换的函数；
2. 生成个子载波，其中（幅度可以一致，但是相位需要满足OFDM调制方案），利用MATLAB画出这个子载波的频域示意图，标注上频率和子载波频率间隔；
3. 对生成的个子载波进行IFFT变换，输出变换结果，并将结果在MATLAB中进行绘图展示；
4. 将调制后的信号s(t)进行FFT变换，输出变换结果，并将结果在MATLAB中进行绘图展示；
5. 上述步骤分别计算使用FFT以及DFT的运行，计算运行时间、占用内存等指标，比较两种算法的性能。
6. 利用MATLAB 自带的FFT函数进行验证，检查与自己编写的C/C++算法结果的差异。

#### 2. 实验设计思路

先分模块编写DFT、IDFT、FFT、IFFT、子载波生成的函数，再在main函数中调用。将生成的子载波通过IDFT，再将得到的IDFT结果通过DFT，将DFT结果和最初子载波对比即可知道函数编写是否错误。IFFT和FFT同理。

1、用C++语言编写DFT和IDFT函数如下：

//旋转因子的计算

complex\_t W(int k,int n,int N){

return complex\_t(cos(2\*PI\*k\*n/N),-sin(2\*PI\*k\*n/N));

}

//格式化 零

complex\_t format(complex\_t &c){

if(fabs(c.real())<eps) c.real(0);

if(fabs(c.imag())<eps) c.imag(0);

return c;

}

double format(double &c){

if(fabs(c)<eps) c=0;

return c;

}

//离散序列的DFT计算,只针对实数序列 ,完全按照DFT的公式来计算,O(n^2)的复杂度

void DFT(vector<double> &subcarriers,vector<complex\_t> &X\_k){

X\_k.clear();

int N=subcarriers.size();

for(int k=0;k<N;++k){

complex\_t t(0,0);

for(int n=0;n<N;++n){

t+=subcarriers[n]\*W(k,n,N);

}

X\_k.push\_back(format(t));

}

int cnt=0;

for(int i=0;i<N;++i){

if(cnt==(int)sqrt(N)){

cout<<endl;

cnt=0;

}

++cnt;

cout<<format(X\_k[i])<<" ";

}

cout<<endl;

}

//IDFT的计算,只针对实数序列

void IDFT(vector<complex\_t> &X\_k,vector<double> &subcarriers){

subcarriers.clear();

int N=X\_k.size();

for(int n=0;n<N;++n){

complex\_t t(0,0);

for(int k=0;k<N;++k){

t+=X\_k[k]\*W(k,-n,N);

}

subcarriers.push\_back(t.real()/N);//运算结果只剩下实部

}

int cnt=0;

for(int i=0;i<N;++i){

if(cnt==(int)sqrt(N)){

cout<<endl;

cnt=0;

}

++cnt;

cout<<format(subcarriers[i])<<" ";

}

cout<<endl;

}

2、用C++语言编写FFT和IFFT函数如下：

//保证N是2的n次幂

int bitlen(int N){

int n=0;

while((N&1)==0){

n++;

N>>=1;

}

return n;

}

int reverse\_bit(int n,int len){//bit反转

int tmp=0;

while(len--){

tmp+=((n&1)<<len);

n>>=1;

}

return tmp;

}

//序数重排

void resort(vector<complex\_t> &subcarriers,int N){

vector<complex\_t> v(subcarriers);

int len=bitlen(N);

for(int i=0;i<N;++i){

subcarriers[i]=v[reverse\_bit(i,len)];

}

}

//基2,FFT算法实现,O(nlogn)的复杂

void FFT(vector<complex\_t> &subcarriers){

int N=subcarriers.size();

int r=bitlen(N);

vector<complex\_t> W(N);

//预先计算旋转因子

for(int i=0;i<N;++i){

double angle=-i\*2\*PI/N;

W[i]=complex\_t(cos(angle),sin(angle));

}

for(int k=0;k<r;++k){//迭代次数

for(int j=0;j<(1<<k);++j){

int butterfly=1<<(r-k);

int p=j\*butterfly;

int s=p+butterfly/2;

for(int i=0;i<butterfly/2;++i){

complex\_t c=subcarriers[i+p]+subcarriers[i+s];

subcarriers[i+s]=(subcarriers[i+p]-subcarriers[i+s])\*W[i\*(1<<k)];

subcarriers[i+p]=c;

}

}

}

//次序重排

resort(subcarriers,N);

int cnt=0;

for(int i=0;i<N;++i){

if(cnt==(int)sqrt(N)){

cout<<endl;

cnt=0;

}

++cnt;

cout<<format(subcarriers[i])<<" ";

}

cout<<endl;

}

//IFFT,与FFT基本一致

void IFFT(vector<complex\_t> &subcarriers){

int N=subcarriers.size();

int r=bitlen(N);

vector<complex\_t> W(N);

//预先计算旋转因子

for(int i=0;i<N;++i){

double angle=i\*2\*PI/N;//IFFT的旋转因子与FFT的旋转因子差一个负号

W[i]=complex\_t(cos(angle),sin(angle));

}

for(int k=0;k<r;++k){//迭代次数

for(int j=0;j<(1<<k);++j){

int butterfly=1<<(r-k);

int p=j\*butterfly;

int s=p+butterfly/2;

for(int i=0;i<butterfly/2;++i){

complex\_t c=subcarriers[i+p]+subcarriers[i+s];

subcarriers[i+s]=(subcarriers[i+p]-subcarriers[i+s])\*W[i\*(1<<k)];

subcarriers[i+p]=c;

}

}

}

//次序重排

resort(subcarriers,N);

int cnt=0;

for(int i=0;i<N;++i){

if(cnt==(int)sqrt(N)){

cout<<endl;

cnt=0;

}

++cnt;

subcarriers[i]/=N;//IFFT与FFT还差一个系数

cout<<format(subcarriers[i])<<" ";

}

cout<<endl;

}

3、因为要求在OFDM场景下，故生成64个幅度均为1，频率间隔为1的子载波，代码如下：

vector<complex\_t> generate\_OFDM\_subcarriers(int num\_subcarriers, double subcarrier\_spacing) {

vector<complex\_t> subcarriers;

// 生成64个频率相隔等距的频率值

for (int i = 0; i < num\_subcarriers; ++i) {

double freq = i \* subcarrier\_spacing;

complex\_t subcarrier = polar(1.0, 0.0); // 设置幅度为1，相位为0的复数

subcarriers.push\_back(subcarrier);

// 打印每个子载波的实部和虚部

cout << "子载波 " << i << " 的实部和虚部：" << subcarrier.real() << ", " << subcarrier.imag() << endl;

}

return subcarriers;

}

4、在main函数中将已生成的子载波放入IDFT模块，再将IDFT得到的结果放入DFT模块，代码如下：

int main(){

int N=64;

vector<complex\_t> X\_k(N);

int num\_subcarriers = 64; // 子载波数量

double subcarrier\_spacing = 1.0; // 子载波频率间隔

// 生成OFDM子载波

vector<complex\_t> subcarriers = generate\_OFDM\_subcarriers(num\_subcarriers, subcarrier\_spacing);

cout<<endl;

vector<double> subcariers\_realpart = OFDM\_subcarriers\_real(num\_subcarriers, subcarrier\_spacing);

IDFT(subcarriers,subcariers\_realpart);

cout<<endl;

DFT(subcariers\_realpart,X\_k);

return 0;

}

5、在main函数中将已生成的子载波放入IFFT模块，再将IFFT得到的结果放入FFT模块，代码如下：

int main(){

int N=64;

vector<complex\_t> X\_k(N);

int num\_subcarriers = 64; // 子载波数量

double subcarrier\_spacing = 1.0; // 子载波频率间隔

// 生成OFDM子载波

vector<complex\_t> subcarriers = generate\_OFDM\_subcarriers(num\_subcarriers, subcarrier\_spacing);

vector<double> subcariers\_realpart = OFDM\_subcarriers\_real(num\_subcarriers, subcarrier\_spacing);

IFFT(subcarriers);

cout<<endl;

FFT(subcarriers);

return 0;

}

6、在matlab端将C++代码得出的各种结果绘图，并用matlab自带的FFT函数进行验证，代码如下：

% Define the recorded signal

x\_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 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1];

% Plot the original signal

subplot(3,2,1);

stem(x\_n);

title('Original Signal (x\_n)');

xlabel('Index');

ylabel('Amplitude');

% Perform IFFT

X\_k = ifft(x\_n);

%X\_k = [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 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];

% Plot the DFT-transformed signal

subplot(3,2,2);

stem(abs(X\_k));

title('IFFT Transformed Signal (|X\_k|)');

xlabel('Frequency Bin');

ylabel('Magnitude');

% Perform FFT

%X\_k\_fft = fft(X\_k)

X\_k\_fft = [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 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1];

% Plot the FFT-transformed signal

subplot(3,2,3);

stem(abs(X\_k\_fft));

title('FFT Transformed Signal (|X\_k\_fft|)');

xlabel('Frequency Bin');

ylabel('Magnitude');

% Perform IDFT to get back original signal from DFT

x\_n\_idft = [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 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];

% Plot the signal obtained after IDFT

subplot(3,2,4);

stem(x\_n\_idft);

title('Signal after IDFT');

xlabel('Index');

ylabel('Amplitude');

% Perform DFT to get back original signal from DFT

x\_n\_dft = [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 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1];

% Plot the signal obtained after IDFT

subplot(3,2,5);

stem(x\_n\_dft);

title('Signal after DFT');

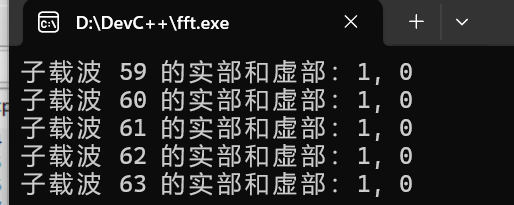
xlabel('Index');

ylabel('Amplitude');

#### 3. 实验结果分析

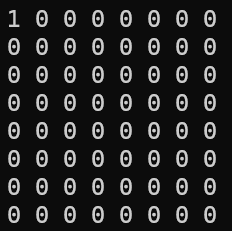
一、C++代码运行结果

1、调用生成子载波函数后得到结果如下：

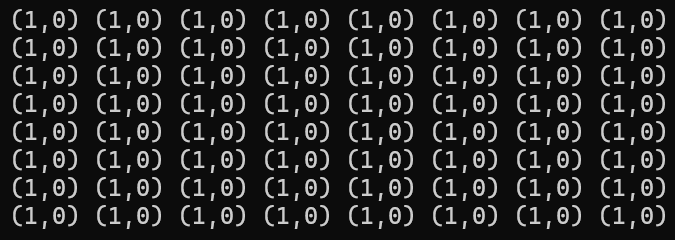


可以看出，该序列和我们希望得到的幅度为一，频率间隔为1，数量为64的子载波一致。

2、将该子载波通过IDFT得到的结果为：

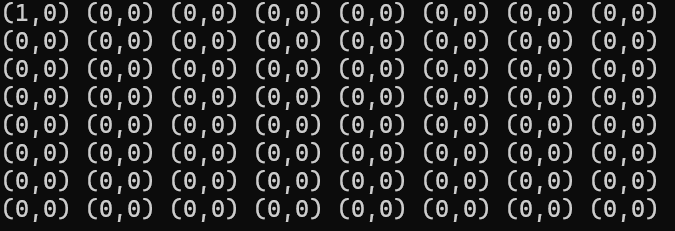


将IDFT结果再通过DFT得到：

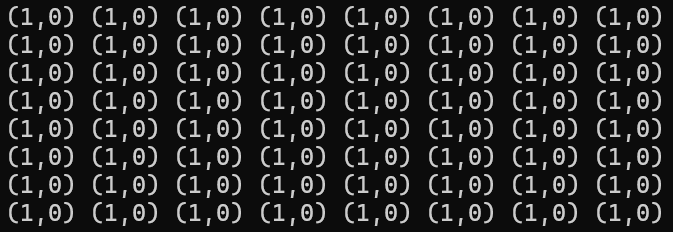


可以看到，最终得出的DFT结果与最初的子载波一致，验证了函数正确性。

3、将该子载波通过IFFT得到的结果为：

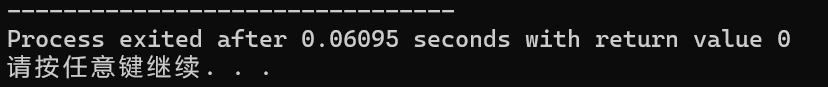


再将IFFT结果通过FFT得到：

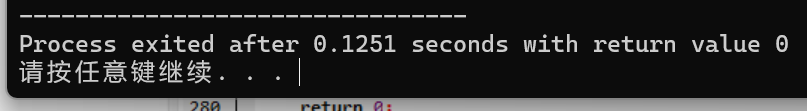


可以看到，最终FFT的结果与最初子载波一致，验证了FFT和IFFT函数的正确性。

4、将该子载波通过IFFT和FFT的时间为：

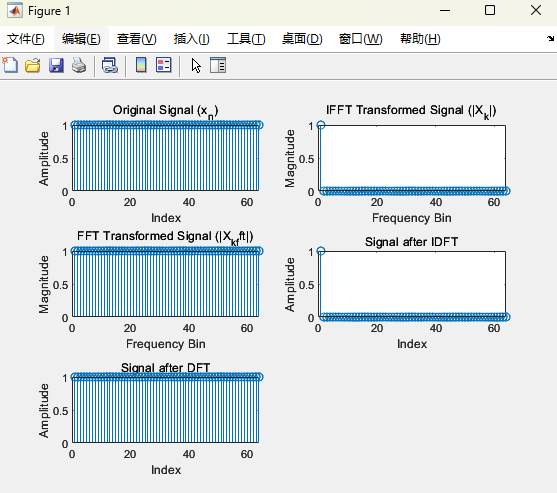


将该子载波通过IDFT和DFT的时间为：



可以看出，FFT算法要比DFT算法快一倍。

二、MATLAB代码运行结果



以上为原始子载波信号和经过MATLAB自带IFFT和FFT的结果图示，可以看出，得到的结果和C++代码得到的结果一致，再次验证了C++代码的正确性。