数字信号处理第一次课程设计

课程设计报告

无92 刘雪枫

2019011008

目录

[一、 课程设计目的 1](#_Toc87202637)

[二、 课程设计描述 1](#_Toc87202638)

[(一) 概述 1](#_Toc87202639)

[(二) 课程设计报告目录结构 1](#_Toc87202640)

[三、 课程设计内容 3](#_Toc87202641)

[(一) 编写基2 FFT函数 3](#_Toc87202642)

[(二) 直接进行DFT比较时间上的差异 4](#_Toc87202643)

[(三) 对连续信号进行频谱分析 6](#_Toc87202644)

[I) 采样率对FFT频谱的影响 8](#_Toc87202645)

[II) 采样点数对FFT频谱的影响 9](#_Toc87202646)

[III) 变换点数对FFT频谱的影响 10](#_Toc87202647)

[IV) 窗函数对FFT频谱的影响 10](#_Toc87202648)

[四、 课程设计小结 11](#_Toc87202649)

[附录一 代码编译与程序运行说明 13](#_Toc87202650)

[支持的编译工具 13](#_Toc87202651)

[如何构建 13](#_Toc87202652)

[Linux 13](#_Toc87202653)

[Windows 14](#_Toc87202654)

[如何运行测试程序 14](#_Toc87202655)

[注意事项 15](#_Toc87202656)

[附录二 库接口说明 16](#_Toc87202657)

[如何使用该库 16](#_Toc87202658)

[C++ 接口 16](#_Toc87202659)

[C 接口 16](#_Toc87202660)

# 课程设计目的

1. 使用C/C++编写任意2的整数次幂点数的基2 FFT程序；
2. 将FFT与直接计算DFT进行比较，分析运行时间上的差异；
3. 利用所编写的FFT程序对给定连续信号进行频谱分析。

# 课程设计描述

## 概述

本次课程设计主要采用C++语言进行编写。目标生成Linux和Windows平台上可供C语言和C++语言进行调用的、含有计算任意2的整数次幂点数的基2 FFT的函数的静态链接库；编写可执行程序，用于比较FFT和DFT在时间上的差异，并对连续信号进行采样，生成用于显示其FFT的MATLAB脚本，最后用MATLAB进行频谱分析。

Linux平台编译工具使用GCC和Clang，支持C++11、C++14与C++17标准，源代码附件中已经提供相应的Makefile文件；Windows平台编译工具使用MSVC，支持C++14与C++17标准，源代码附件中已经提供相应的.sln与.vcxproj文件；MacOS平台暂未进行测试。

静态库与可执行文件的具体构建方式参见“附录”。

## 课程设计报告目录结构

课程设计报告目录结构如下：

2019011008\_刘雪枫\_数字信号处理第一次课程设计

|

| 课程设计报告.pdf

|

\---code

| .gitattributes

| .gitignore

| Makefile

| README.md

| README\_zh\_CN.md

|

\---src

| .gitignore

| clean\_publish.cmd

| DSPFFT.sln

| publish.cmd

|

+---dspfft

| | dspfft.vcxproj

| | dspfft.vcxproj.filters

| | Makefile

| |

| +---cpp

| | dspfft.cpp

| | dspfft\_for\_c.cpp

| |

| \---include

| dspfft.h

| dspfft.hpp

| dspfft\_decl.h

|

+---homework

| .gitignore

| format\_complex.hpp

| homework.vcxproj

| homework.vcxproj.filters

| main.cpp

| Makefile

|

\---prior

| Makefile

| prior.vcxproj

| prior.vcxproj.filters

|

+---cpp

| bool\_arg\_parser.cpp

| raw\_bool\_arg\_parser.cpp

|

\---include

| bool\_arg\_parser.hpp

| compiler.h

| declaration.h

| namespace.h

| numbers.hpp

| prior.h

| raw\_bool\_arg\_parser.h

|

\---details

compiler\_clang.i

compiler\_gcc.i

compiler\_msvc.i

# 课程设计内容

本次课程设计所选用的实验平台为Ubuntu 20.04，使用GCC 11.1（g++）进行编译，语言标准采用C++17，并开启O2优化。

## 编写基2 FFT函数

实验采用按频率抽取的基2 FFT算法，采用循环实现，算法的C++代码如下：

auto stage\_inc = static\_cast<size\_type>(1);

auto stage\_dec = input.size();

for (; stage\_dec != 1; stage\_dec >>= 1, stage\_inc <<= 1)

{

// stage\_inc: group\_num; stage\_dec: per group

auto half\_of\_group = stage\_dec / 2;

for (size\_type group = 0; group < stage\_inc; ++group)

{

auto offset = group \* stage\_dec;

for (size\_type idx = offset; idx < offset + half\_of\_group; ++idx)

{

result[idx] = input[idx] + input[idx + half\_of\_group];

result[idx + half\_of\_group] = (input[idx] - input[idx + half\_of\_group]) \* w\_n[(idx - offset) \* stage\_inc];

}

}

::std::copy(result.begin(), result.end(), input.begin());

}

for (size\_type i = 0; i < input.size(); ++i)

{

result[i] = input[reverse\_bit(i) >> ((sizeof(i) \* 8) - (get\_log\_2\_of\_base\_2(input.size()) - 0))];

}

其中，input为输入的序列，result为最终输出的DFT序列，get\_log\_2\_of\_base\_2，为获取以2为底的对数值，w\_n[i]即为DFT中的*W­Ni*。

此段代码被封装为C++函数模板base\_2\_fft<typename>（声明在头文件src/dspfft/include/dspfft.hpp中，定义在代码文件src/dspfft/cpp/dspfft.cpp中）和C函数base\_2\_fftf、base\_2\_fftl和base\_2\_fftll（声明在头文件dspfft/include/dspfft.h中，分别适用于float、double与long double类型）。在C++中均位于命名空间dspfft。

## 直接进行DFT比较时间上的差异

根据DFT的算法，编写直接计算DFT的函数。C++相关代码如下：

using complex\_type = ::std::complex<floating\_type>;

for (size\_type k = 0; k < x.size(); ++k)

{

result[k] = ::std::inner\_product(x.begin(), x.end(), w\_n.begin(), complex\_type{0}, ::std::plus<complex\_type>{},

[k](complex\_type x\_n, complex\_type w\_n) { return x\_n \* static\_cast<complex\_type>(::std::pow(w\_n, k)); });

}

封装为命名空间dspfft中的函数模板dft<typename>以及C函数dftf、dftl与dftll。

同时编写测试程序对两种算法进行测试。为了更好地测试性能，决定采用32768点序列进行测试，测试序列为*x*[*n*]=*n*+1=[1, 2, …, 32768]。对其分别做基2的FFT和DFT，并分别计算运行时间。

为了进行计时，先编写函数获取当前时间戳（单位：毫秒）：

PRIOR\_NODISCARD PRIOR\_FORCED\_INLINE static

typename ::std::chrono::milliseconds::rep

get\_milliseconds()

{

return ::std::chrono::duration\_cast<::std::chrono::milliseconds>(::std::chrono::system\_clock::now().time\_since\_epoch()).count();

}

然后比较两个算法的所需时间：

auto fft\_begin = ::get\_milliseconds();

auto r1 = dspfft::base\_2\_fft(v);

auto fft\_end = ::get\_milliseconds();

::std::cout << "FFT time: " << (fft\_end - fft\_begin) << "ms" << '\n';

auto dft\_begin = ::get\_milliseconds();

auto r2 = ::dspfft::dft(v);

auto dft\_end = ::get\_milliseconds();

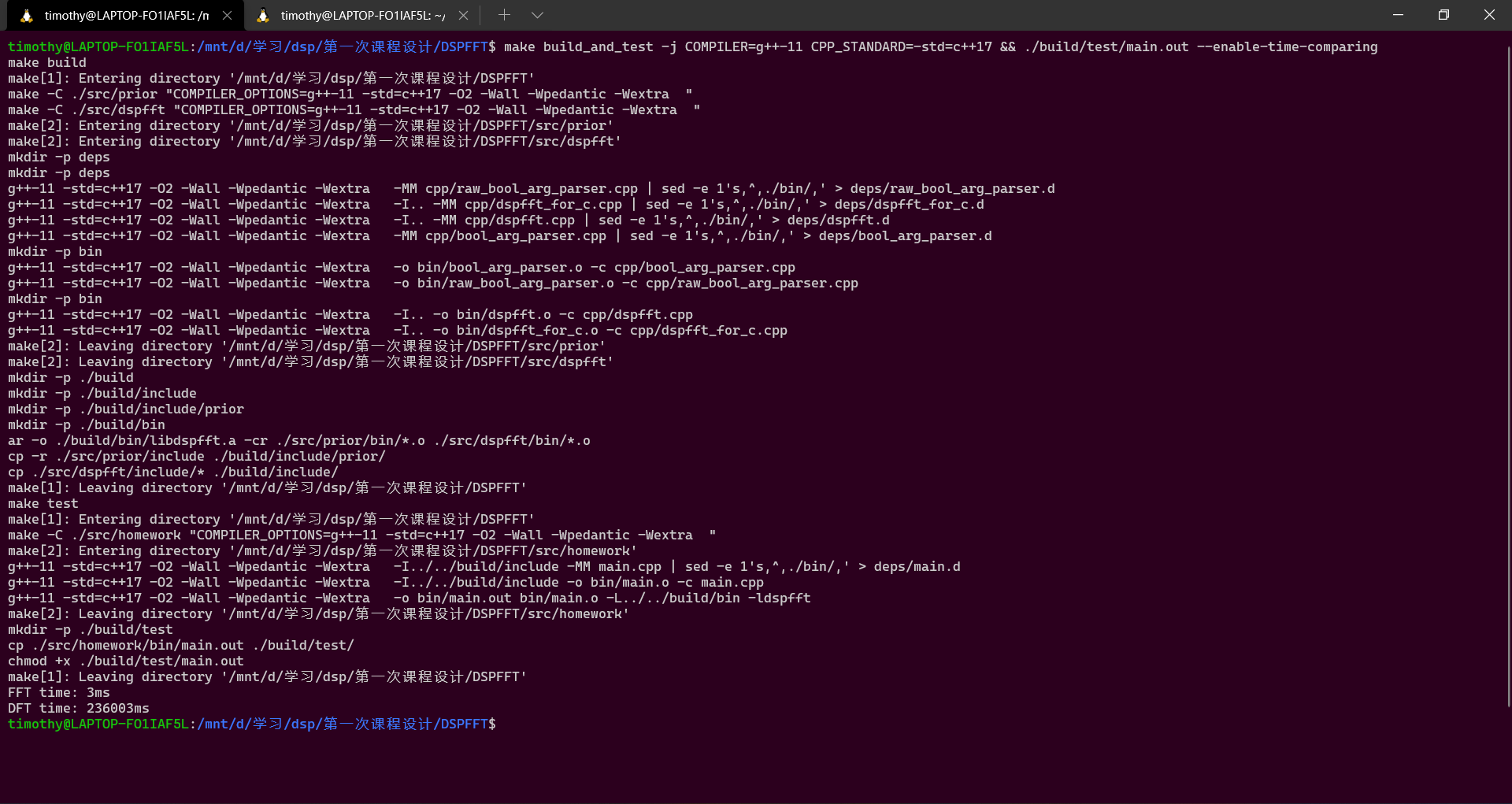
::std::cout << "DFT time: " << (dft\_end - dft\_begin) << "ms" << '\n';

::std::cout.flush();

由于本次作业有多个问题，因此用命令行参数来指定运行哪个问题的代码。本问题需指定--enable-time-comparing参数运行。因此，编写完毕后，在主目录（即最外层的Makefile文件所在目录）内输入：

make build\_and\_test -j COMPILER=g++-11 CPP\_STANDARD=-std=c++17 && ./build/test/main.out --enable-time-comparing

开始运行，程序运行完毕后结果如下：



从结果中可以看到，FFT需要的时间是3毫秒，而直接进行DFT所需的时间达到了236秒，即将近4分钟的时间。因此，我们看到，FFT比DFT大大节省了时间。

本问题代码位于文件src/homework/main.cpp中

## 对连续信号进行频谱分析

接下来利用编写的基2 FFT程序对下面的连续信号进行频谱分析：

根据连续信号的表达式，信号由103Hz、107Hz、115Hz的单频正弦波叠加而成，因此频谱应当在这三个频点附近存在峰值。

在本问题中采用C++来生成MATLAB脚本文件result.m，然后使用MATLAB运行脚本，观察绘制出的频谱图。

首先需要对连续信号进行采样得到离散序列，C++伪代码如下：

::std::vector<::std::complex<double>>

v(total\_points, ::std::complex<double>(0, 0));

point\_t i = 0;

::std::generate\_n(v.begin(), sample\_points,

[i, sample\_rate]() mutable

{

double t = (double)i++ / sample\_rate;

return 0.8 \* ::std::sin(2.0 \* pi\_v \* 103.0 \* t)

+ ::std::sin(2.0 \* pi\_v \* 107.0 \* t)

+ 0.1 \* ::std::sin(2.0 \* pi\_v \* 115.0 \* t);

}

);

其中，sample\_rate为采样率，v为最终得到的采样序列。

若要对信号进行加窗，则需要生成窗函数。本次实验决定采用矩形窗和汉宁窗。生成汉宁窗的函数如下：

auto get\_hanning(::std::size\_t N)

-> ::std::vector<::std::complex<double>>

{

::std::vector<::std::complex<floating\_type>> res;

res.reserve(N);

::std::size\_t i = 0;

::std::generate\_n(::std::back\_inserter(res), N, [i, N]() mutable

{

constexpr auto pi = ::prior::numbers::pi\_t<floating\_type>::value;

return static\_cast<floating\_type>(0.5L \* (1.0L - ::std::cos(2 \* pi \* i++ / (N - 1))));

});

return res;

}

得到汉宁窗后，与采样序列相乘。为了提高效率，可以在采样时对其进行加窗：

::std::vector<::std::complex<double>>

v(total\_points, ::std::complex<double>(0, 0));

point\_t i = 0;

auto hanning = ::get\_hanning<double>(total\_points);

::std::generate\_n(v.begin(), sample\_points,

[i, sample\_rate, &hanning]() mutable

{

constexpr auto pi\_v = ::prior::numbers::pi\_t<double>::value;

double t = (double)i / sample\_rate;

return hanning[i++] \*

(0.8 \* ::std::sin(2.0 \* pi\_v \* 103.0 \* t)

+ ::std::sin(2.0 \* pi\_v \* 107.0 \* t)

+ 0.1 \* ::std::sin(2.0 \* pi\_v \* 115.0 \* t));

}

);

然后对得到的序列计算FFT，并将结果及绘图代码写入到MATLAB脚本文件result.m中。

本问题全部代码位于src/homework/main.cpp中。

注意到，本题C++代码使用Linux平台运行，而MATLAB使用Windows平台运行，因此需要解决行尾不一致的问题，为此设计预定义宏TRANSFORM\_TO\_CRLF\_NEWLINE控制是否生成行尾为CRLF的文件。输入如下命令运行：

make build\_and\_test -j COMPILER=g++ CPP\_STANDARD=-std=c++17 PREDEFINED\_MACRO=-DTRANSFORM\_TO\_CRLF\_NEWLINE && ./build/test/main.out --writing-matlab-code-to-file

即可在当前工作目录生成MATLAB脚本文件result.m。

使用MATLAB运行该脚本文件，得到下面的图像并进行分析：

### 采样率对FFT频谱的影响

先分析采样率对得到的FFT频谱的影响。本问题控制总采样点数为800，变换点数为1024，并且使用矩形窗，采样率分别为150Hz、300Hz、800Hz进行对照，得到的图像如下图所示：



可以看到，采样率为300Hz和800Hz时，在原连续信号的频率103Hz、107Hz和115Hz附近均存在峰值，但是采样率为150Hz时这三个频点位于右半张图内，在频率较低的部分却出现了低频分量35Hz、43Hz和47Hz。

究其原因，由于离散信号的频谱是周期的且输入信号是实的，故信号的FFT是厄米对称的，所以对于采样率为300Hz和800Hz来说，一定在高频处存在与低频处对称的频谱。而对于根据奈奎斯特采样定理，对于带限信号，至少需要以高于2倍频率的采样率采样才能恢复原信号。本题中，原连续正弦信号的频率分别为103Hz、107Hz和115Hz，因此采样率应当分别至少为206Hz、214Hz和230Hz才能正确得到频谱，否则频谱发生混叠，出现其他的低频分量。因此采样率为150Hz时无法得到正确的频率分量。

### 采样点数对FFT频谱的影响

下面分析采样点数对FFT频谱的影响。

控制采样率为300Hz，变换点数为1024，使用矩形窗，以变换点数分别为50、300和800的条件进行分析，得到图像如下：



可以看到，随着采样点数增加，得到频谱的主瓣宽度变窄，主瓣高度增加。值得注意的是，当采样点数为50的时候，103Hz与107Hz的峰发生了混叠。这是因为采样率为300Hz，而当采样点数低于时，其中一个频率的主瓣峰值恰好落在另一个频率的主瓣末尾处，因此采样点数为50时两个频率的主瓣发生交叠，难以分辨。因此增加采样点数可以提高频率分辨力。

### 变换点数对FFT频谱的影响

下面讨论变换点数的影响。

控制采样率为300Hz，采样点数300，使用矩形窗，分别进行1024、2048和8192点的FFT，得到频谱如下：



可以看到，三个频谱图的包络形状大致相同，但是变换点数越多越密集，说明变换点数增加，即在采样点数后补零只是使频域上的采样点数增多，但是并不能提高其频率分辨力。

### 窗函数对FFT频谱的影响

之前的问题中用到的都是矩形窗，现在考虑使用汉宁窗。

仍然使用300Hz的采样率，在采样点数分别为300和1024的条件下，比较矩形窗和汉宁窗。绘制频谱图如下：



上图中，第一行采样点数300，第二行采样点数1024；第一列使用汉宁窗，第二列使用矩形窗。可以看到，使用汉宁窗频谱的旁瓣明显减弱，频谱也更加平滑。

# 课程设计小结

总体上来讲，本次课程设计还是比较顺利的，我基本上独立完成了本次课程设计，使用C++和MATLAB完成了频谱分析。

但是，课程设计当中，我还是遇到了一些困难。

首先，在进行代码编写的过程中，由于我对C++语言标准还不是很熟悉，因此出现过一些编译错误；此外我对Makefile的编写也不是非常了解，所以也发生了一些构建错误。基于此，我在网上查找了很多资料，最终解决了问题。

第二，我对FFT算法的理解还欠佳，因此在编写FFT函数的时候，也遇到了一些困难。起初想要使用递归完成，但是为了提高效率决定使用循环来计算。但是循环计算FFT看起来不是那么显然，我也是自己静心思考了一段时间才写出了代码。

第三，我对数字信号频谱和连续信号频谱之间的关系还不是很熟练。因此，在最后绘制频谱图的时候，我绘制的横轴（频率轴）出现了问题，然后我重新推导了一次数字频率与模拟频率的关系，才将其正确画出。

此外，我认为课程设计还存在一些不足之处。例如，目前绘制频谱时，如果要修改采样率等参数，还需要将代码重新构建，改为命令行传递参数也许更加合适；最后对连续信号进行频谱分析的时候，还是比较粗略，讨论不够深入；窗函数目前只讨论了矩形窗和汉宁窗，对于其他的窗函数还没有进行过多讨论，等等。

在此次课程设计中，我受益匪浅，不仅更加熟练了编程技巧，更是对连续信号与离散信号的关系有了更深的理解，同时对FFT算法有了更亲身的体会。在此感谢老师的耐心授课与助教的辛勤付出以及周围同学的鼓励，没有老师和助教的讲授与努力，我也很难顺利完成本次课程设计。

# 附录一 代码编译与程序运行说明

## 支持的编译工具

| **编译工具** | **平台** | **目标平台** | **C++11** | **C++14** | **C++17** |
| --- | --- | --- | --- | --- | --- |
| GCC 7 ~ GCC 11 | Linux x86-64 | -m64 | Perfectly supported | Perfectly supported | Perfectly supported |
| Clang 12 | Linux x86-64 | -m64 | Perfectly supported | Perfectly supported | Perfectly supported |
| MSVC 19 | Windows (64-bit) | x64 | Not supported | Supported | Supported |
| MSVC 19 | Windows (64-bit) | x86 | Not supported | Compiler warnings | Supported |

## 如何构建

### Linux

* 构建静态库

make build [options]

options 可以是：

* + COMPILER：指定构建所用的编译工具。值可以是 g++ 或 clang++，默认为g++
  + CPP\_STANDARD：指定 C++ 语言标准，值可以是 -std=c++11、-std=c++14、-std=c++17、-std=gnu++11、-std=gnu++14、-std=gnu++17 等。默认为 -std=c++11
  + OPTIMIZATION：指定优化等级。值可以是 -O0、-O1、-Os 等，默认为 -O2
  + WARNING\_LEVEL：指定警告等级。默认值为 -Wall -Wpedantic -Wextra
  + PREDEFINED\_MACRO：指定预定义宏，例如 -DUNICODE，默认为空

例如：

make build CPP\_STANDARD=-std=c++17 COMPILER=clang++

然后，生成的库将会位于 build 文件夹内。二进制库文件为 build/bin/libdspfft.a，头文件位于 build/include 文件夹内。

* 构建测试程序

make test [options]

options 与构建静态库相同。此外，当 PREDEFINED\_MACRO 包含 -DTRANSFORM\_TO\_CRLF\_NEWLINE 选项时，程序会生成行尾为 CRLF 的 MATLAB 代码（用来支持 Windows 平台的 MATLAB），否则生成的行尾是 LF。

可执行文件为 build/test/main.out。

* 全部构建

make build\_and\_test [options]

此命令与下面的命令等价：

make build [options] && make test [options]

* 清理

make clean

将会清理所有构建过程中产生的文件。

### Windows

进入文件夹 src，使用 Visual Studio 2019 或更高版本打开 DSPFFT.sln，并选择一个目标平台（建议为 Release | x64），再生成解决方案。然后运行 publish.cmd 来发布库文件和测试程序。二进制库文件为 publish\bin\dspfft.lib，其头文件位于 publish\publish\include 文件夹内，可执行文件为 publish\test\homework.exe。需要注意，publish.cmd 默认为 Release | x64 平台，如果要改变平台，请编辑该脚本，修改变量 BINARY\_DIR 的值。

如果要清理发布的文件，可以运行 clean\_publish.cmd。

## 如何运行测试程序

在 Linux 上，可执行文件为 build/test/main.out；在 Windows 上，可执行文件为 homework.exe。运行程序时可以传递下面的参数：

* --enable-time-comparing：比较 FFT 与直接运行 DFT 的时间
* --writing-matlab-code-to-file：生成 MATLAB 代码，并写入 MATLAB 脚本文件 result.m

## 注意事项

* 由于 dspfft 库是用 C++ 编写的，因此当使用 C 语言调用该库时，应当链接 C++ 运行时
* 如果要在 Linux 上运行测试程序，并用 Windows 平台的 MATLAB 运行生成的脚本，需要确保在构建时定义了 TRANSFORM\_TO\_CRLF\_NEWLINE 宏，以保证生成的脚本行尾是 CRLF

# 附录二 库接口说明

## 如何使用该库

C++ 语言应当包含头文件 dspfft.hpp，C 语言应当包含 dspfft.h。在 Linux 上需链接 libdspfft.a，在 Windows 上需链接 dspfft.lib。

库的接口如下：

### C++ 接口

接口在 <dspfft.hpp> 中声明，所有的接口都在命名空间 dspfft 中。

* template <typename floating\_type> ::std::vector<::std::complex<floating\_type>> base\_2\_fft(const ::std::vector<::std::complex<floating\_type>>& x)
* template <typename floating\_type> ::std::vector<::std::complex<floating\_type>> dft(const ::std::vector<::std::complex<floating\_type>>& x)

### C 接口

接口在 <dspfft.h> 中生命。当使用 C++ 语言包含时，所有接口都定义在 dspfft 命名空间中，所有函数都被声明为具有 C 语言链接（extern "C"）。

* complexf, complexl, complexll：实部和虚部分别为float, double and long double 的复数结构体
* complexf \*base\_2\_fftf(complexf \*, size\_t), complexl \*base\_2\_fftl(complexl \*, size\_t), complexll \*base\_2\_fftll(complexll \*, size\_t)
* complexf \*dftf(complexf \*, size\_t), complexl \*dftl(complexl \*, size\_t), complexll \*dftll(complexll \*, size\_t)