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
二、 课程设计描述.....1
(一) 概述.....1
(二) 课程设计报告目录结构.....1
三、 课程设计内容.....3
(一) 编写基 2 FFT 函数.....3
(二) 直接进行 DFT 比较时间上的差异.....4
(三) 对连续信号进行频谱分析.....6
I) 采样率对 FFT 频谱的影响.....8
II) 采样点数对 FFT 频谱的影响.....9
III) 变换点数对 FFT 频谱的影响.....10
IV) 窗函数对 FFT 频谱的影响.....10
四、 课程设计小结.....11
附录一 代码编译与程序运行说明.....13
支持的编译工具.....13
如何构建.....13
Linux.....13
Windows.....14
如何运行测试程序.....14
注意事项.....15
附录二 库接口说明.....16
如何使用该库.....16
C++ 接口.....16
C 接口.....16

一、 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。

二、 课程设计描述

(一) 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”。

(二) 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
```

三、 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。

(一) 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 W_N^i。

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。

(二) 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 $x[n]=n+1=[1, 2, \ldots, 32768]$。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：

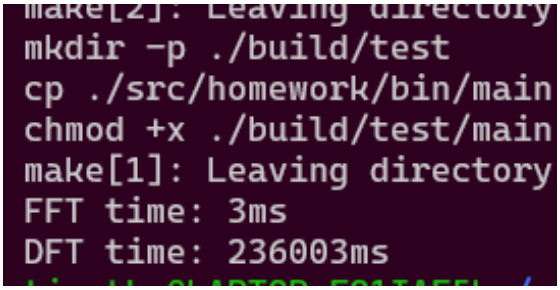
```
auto fft_begin = ::get_milliseconds();
auto r1 = dspfft::base_2_fft(v);
auto fft_end = ::get_milliseconds();
::std::cout << "FFT time: " << (fft_end - fft_begin) << "ms" <<
'\n';

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

(三) 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：

$$s(t) = 0.8 \times \sin(2\pi \times 103t) + \sin(2\pi \times 107t) + 0.1 \times \sin(2\pi \times 115t)$$

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：

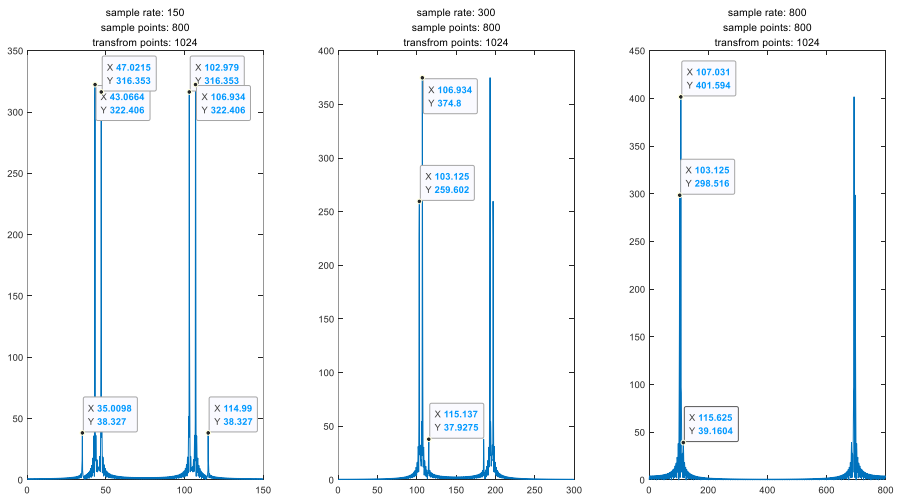
make build_and_test -j COMPILER=g++ CPP_STANDARD=-std=c++17 PREDEFINED_MACRO=-DTRANSFORM_TO_CRLF_NEWLINE && ./build/test/main.out --writing-matlab-code-to-file

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：

I) 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：



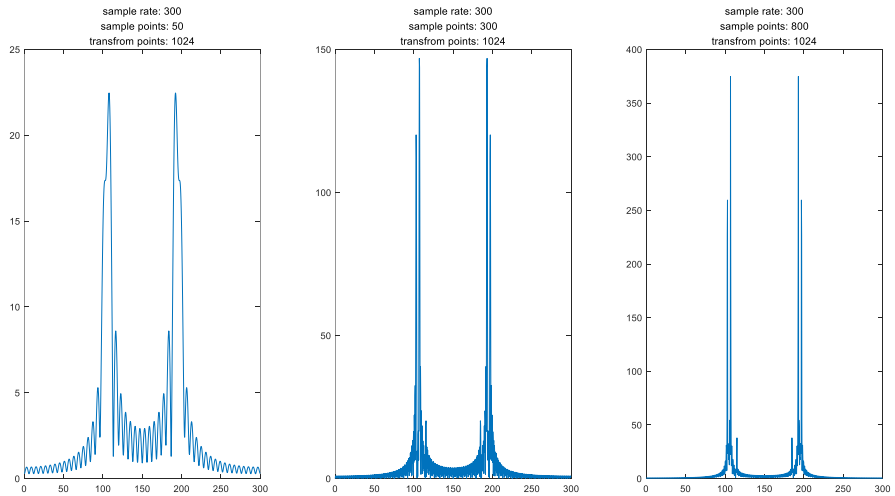
可以看到，采样率为 300Hz 和 800Hz 时，在原连续信号的频率 103Hz、107Hz 和 115Hz 附近均存在峰值，但是采样率为 150Hz 时这三个频点位于右半张图内，在频率较低的部分却出现了低频分量 35Hz、43Hz 和 47Hz。

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。

II) 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：



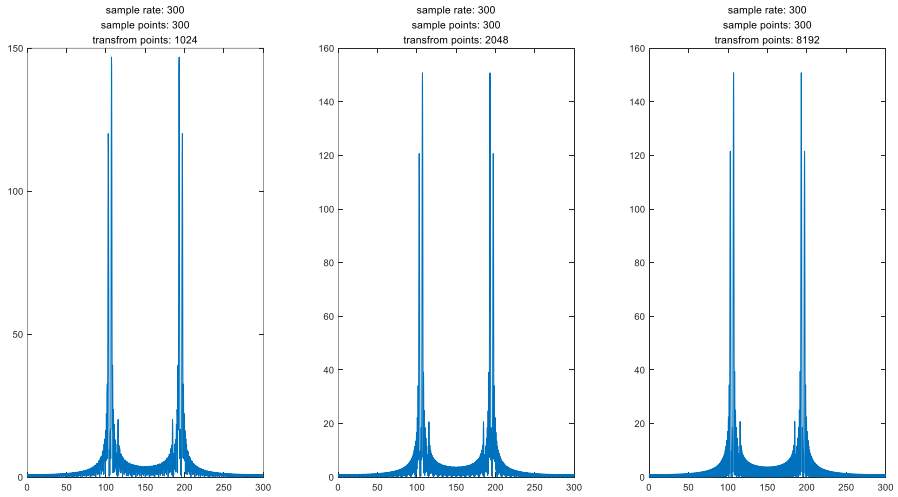
可以看到，随着采样点数增加，得到频谱的主瓣宽度变窄，主瓣高度增加。值得注意的是，当采样点数为 50 的时候，103Hz 与 107Hz 的峰发生了混叠。这是因为采样率为 300Hz，而当采样点数低于$\frac{300\text{Hz}}{(107-103)\text{Hz}}=75$时，其中一个频率的

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。

III) 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：

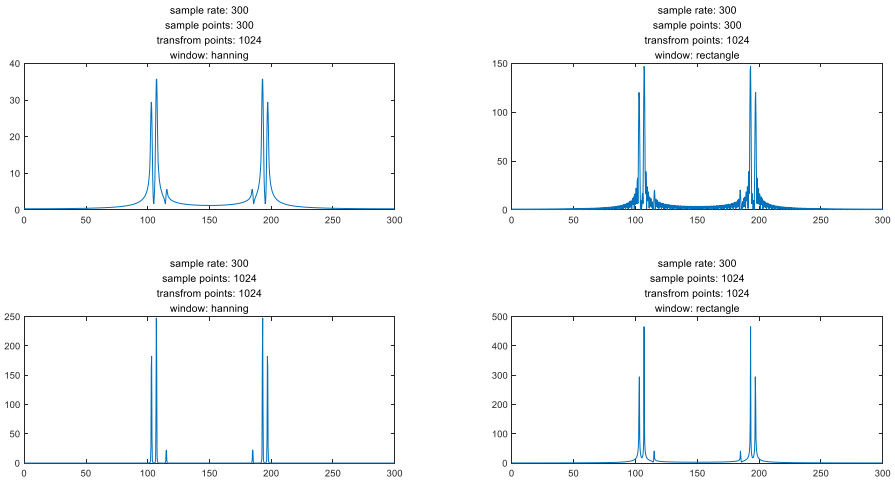


可以看到，三个频谱图的包络形状大致相同，但是变换点数越多越密集，说明变换点数增加，即在采样点数后补零只是使频域上的采样点数增多，但是并不能提高其频率分辨力。

IV) 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。

四、 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)`