

FFTW 使用手册

Ver 1.0

熊金水

xjs.xjtu@gmail.com

2011-5-11

一、FFTW 是什么

FFTW—Fastest Fourier Transform in the West, 一个颇为古怪的名字, 是由 MIT 的 Matteo Frigo 博士和 Steven G. Johnson 博士开发的一个完全免费的软件包。FFTW 最初的 release 版本于 1997 年发布, 最新的 release 版本 3.2.2 于 2009 年 7 月发布。它是一个 C 语言开发的库, 支持任意大小的、任意维数的数据的离散傅里叶变换(DFT), 并且还支持离散余弦变换(DCT)、离散正弦变换(DST)和离散哈特莱变换(DHT)。

根据两位博士的测试, FFTW 在计算速度上远远优于目前其他计算 DFT 的免费的库, 甚至可以和收费的 DFT 库相媲美。但是, 和收费的 DFT 库相比, FFTW 能够轻松的在不同的平台上移植。我们熟悉的 MATLAB, 就是调用了 FFTW 来实现 DFT/IDFT 变换的。

另外, 值得一提的是, FFTW 还获得了 1999 年的 J. H. Wilkinson Prize for Numerical Software 奖。J. H. Wilkinson Prize for Numerical Software 每四年颁发一次, 用于奖励那些“最好的解决了高质量的数值计算问题的软件设计”。

二、FFTW 有哪些优越的性能

- **高速**——远优于目前其他的免费 DFT 库。
- 支持**任意维度**的变换。
- 支持**任意大小**的变换——FFTW 对 $N = 2^a * 3^b * 5^c * 7^d * 11^e * 13^f$ 的变换处理的最好, 其中 $e + f = 0$ 或 1, 其他指数可以为任意值。
- 支持快速的**输入为实数**的 DFT 变换。
- 支持 **DCT(I-IV)**和 **DST(I-IV)**。
- 支持**多线程**。
- 支持**并行处理**。
- **可移植性**——任意包含 C 编译器的平台都可以使用 FFTW。
- 同时包含 **C 和 Fortran 接口**。
- **完全免费**——如果您在您的系统里使用了 FFTW, 请您尊重两位博士的辛勤劳作, 在您的参考文献(Reference)中添加下面这篇文章。Matteo Frigo and Steven G. Johnson, "The Design and Implementation of FFTW3," Proceedings of the IEEE 93 (2), 216–231 (2005).

三、如何在 windows 上安装 FFTW

第一步, 在 <http://www.fftw.org/install/windows.html> 上下载 32 位/64 位版本的 windows 安装程序 [fftw-3.2.2.pl1-dll32.zip](#) 或 [fftw-3.2.2-dll64.zip](#), 并解压。

第二步, 打开 windows 命令行窗口, 把命令行的目录转到第一步中解压的目录下。运行一下三条指令:

- `lib /machine:ix86 /def:libfftw3-3.def`
- `lib /machine:ix86 /def:libfftw3f-3.def`
- `lib /machine:ix86 /def:libfftw3l-3.def`

这样会在您的当前目录下生成三个 lib 文件和三个相应的 dll 文件, 其中文件名包含 `fftw3-3` 的为 double 精度, `fftw3f-3` 的为 float 精度, `fftw3l-3` 的为 long double 精度——在 32 的机器中, long double 精度与 double 没有什么区别。

第三步, 对于 lib 文件, 在您的 VS 开发工具里添加 lib 文件的目录, 并在 VS 工程里添加您需要的精度所对应的 lib 文件。

对于相应的 dll 文件, 您可以把第二步中的目录设为环境变量的目录, 然后注销重启就可以使用了, 也可以拷贝到 `system32` 目录下, 或者拷贝到您的工程目录下。

对于头文件 `fftw3.h`, 您可以在您的 VS 工具下添加 `fftw3.h` 的目录, 也可以直接把 `fftw3.h` 拷贝到您的工程目录下, 我推荐使用后者, 因为您在开发过程中, 可能需要修改头文件, 如果您再别的工程里再次使用该库时, 就难以找到原始的头文件了。

四、第一次使用 FFTW

第一次使用 FFTW 编程, 可以使用下面的结构:

```
#include <fftw3.h>
...
{
    fftw_complex *in, *out;
    fftw_plan p;
    ...
    in = (fftw_complex*)
fftw_malloc(sizeof(fftw_complex) * N);
    out = (fftw_complex*)
fftw_malloc(sizeof(fftw_complex) * N);

    // 输入数据in 赋值

    p = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD,
                        FFTW_ESTIMATE);

    fftw_execute(p); // 执行变换

    ...
    fftw_destroy_plan(p);
    fftw_free(in);
    fftw_free(out);
}
```

总的来说就是先输入, 然后构造策略 `plan`, 最后执行 `plan` 就可以了, 还是很简单的。下面, 我们一步一步来解析这个程序, 并剖析每一步中需要注意些什么。

1、内存空间的申请与释放

以 `in` 的分配为例。

在这里, 空间分配有三种可以选择的方式:

第一, 直接 `in[N]`;

第二, 使用 ANSI C 或者 C++ 语言中的 `malloc`, `new` 等动态分配;

第三, 使用 FFTW 提供的 `fftw_malloc` 函数动态分配。

第一种方法的问题在于, 这种方法申请的空间由编译器在栈内存里分配, 众所周知, 栈内存是非常有限的, 在 windows 上为 2M 大小, 所以对于大的数据容易导致 `Stack Overflow` 的致命性错误。当然, 你可以在编译器设置里边它调大一些, 但是换了个环境, 你是不是就愣了~

第二种方法虽然解决了第一种方法存在的问题, 但是, 由于不同的编译器内存分配策略的不同, 可能使分配的数组并没有内存对齐, 而内存没有对齐, 对 FFTW 性能上的影响将是

显著的。当然，你可以使用一些语言的特性让内存对齐，但是，有下面更好的方法，为什么非要用这种方法呢~

第三种方法则是由 FFTW 提供的内存分配接口，它在堆内存上申请空间，并且能够保证内存对齐，同时可以在不同的平台之间顺利的移植，何乐而不为~

空间的分配问题解决了，空间的释放问题当然也就水到渠成了，如果我们使用了 `fftw_malloc()` 来分配空间，那么使用 `fftw_free()` 释放相应的空间就行了。

最后，在内存方面值得注意的问题是，`fftw_plan` 是一种声明了一个变量就需要使用 `fftw_destroy_plan()` 来销毁的类型。

2、实数 FFT 中内存的复用

对于实数输入的序列，其 DFT 输出是一个有共轭对称性质的复数序列，在 FFTW 中，两位博士利用了这一点来减少内存空间的使用。

对于一维的情况，假设输入序列为 $x(n), n=0 \sim N-1$ ，对应的 DFT 为 $X[k], k=0 \sim N-1$ 。我们知道， $X(n)$ 有共轭对称性，即：

$$X[k] \leftarrow X[N-k]$$

在 FFTW 中的 $X[k]$ 的存储策略如下图所示。

对于 N 为偶数的情况——以 $N = 4$ 为例：

0	1	2	3
---	---	---	---

0r	0i	1r	1i	2r	2i
----	----	----	----	----	----

变换以后，内存空间多了两个单元。

对于 N 为奇数的情况——以 $N = 5$ 为例：

0	1	2	3	4
---	---	---	---	---

0R	0I	1R	1I	2R	2I
----	----	----	----	----	----

变换以后，内存空间多了一个单元。

更一般的，我们可以总结一下，对于输入为 N 的序列，我们需要 $2 * (\text{floor}(N/2) + 1)$ 个存储单元。 N 为奇数时，增加一个单元； N 为偶数时，增加两个存储单元。

这里有两个问题，当 `in` 和 `out` 不是一块内存时，即使我们为 `out` 申请了 N 个 `fftw_complex` 类型的内存单元，但是后面会有一部分的内存空间不被操作，如下图所示，FFT 的 `output` 的后两个存储单元都没有操作（全是 0 是因为我在程序中手动赋值为 0）。

```
FFT input is:
1
2
3
4
5

FFT output is :
15 + i 0
-2.5 + i 3.44095
-2.5 + i 0.812299
0 + i 0
0 + i 0
```

对于二维的情况，对称性显得稍微复杂一些。假设输入为 $x[m][n], m = 0 \sim M-1, n = 0 \sim N-1$ ，对应的 DFT 序列为 $X[u][v], u = 0 \sim M-1, v = 0 \sim N-1$ 。从二维傅里叶变换的公式中，我们可以很轻

松的得到 X 有下面的共轭对称性质:

$$X[u][v] \leftrightarrow X[M-u][N-v]$$

$u=0$ 时, $X[0][v] \leftrightarrow X[M][N-v] = X[0][N-v]$;

$v=0$ 时, $X[u][0] \leftrightarrow X[M-u][N] = X[M-u][0]$;

$u \neq 0$ 且 $v \neq 0$ 时, $X[u][v] \leftrightarrow X[M-u][N-v]$, 二者的对称中心在 $(M/2, N/2)$ 处。存储策略和一维的情况有些相似。

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

0	0	1	1	2	2
4	4	5	5	6	6
8	8	9	9	10	10
12	12	13	13	14	14

以上是列数为偶数的情况, 以下是列数为奇数的情况。

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

0	0	1	1	2	2
5	5	6	6	7	7
10	10	11	11	12	12
15	15	16	16	17	17
20	20	21	21	22	22

3、FFTW 中提供了哪些策略, 各有什么不同

FFTW 中, 针对不同的情况, 提供了下面一些生成策略(plan)的函数接口。

```
fftw_plan fftw_plan_dft_1d(int n, fftw_complex *in,
fftw_complex *out, int sign, unsigned flags); // 一维复数据的DFT、IDFT
```

```

fftw_plan fftw_plan_dft_2d(int n0, int n1,fftw_complex
*in, fftw_complex *out,int sign, unsigned flags);//二维
复数据DFT、IDFT

fftw_plan fftw_plan_dft_3d(int n0, int n1, int
n2,fftw_complex *in, fftw_complex *out,int sign, unsigned
flags);//三维复数据DFT、IDFT

fftw_plan fftw_plan_dft(int rank, const int
*n,fftw_complex *in, fftw_complex *out,int sign, unsigned
flags);//rank维复数据DFT、IDFT

fftw_plan fftw_plan_dft_r2c_1d(int n, double *in,
fftw_complex *out,
unsigned flags);//一维实数DFT

fftw_plan fftw_plan_dft_c2r_1d(int n, fftw_complex *in,
double *out,
unsigned flags);//一维实数IDFT

fftw_plan fftw_plan_dft_r2c_2d(int n0, int n1,double *in,
fftw_complex *out,
unsigned flags);//二维实数DFT

fftw_plan fftw_plan_dft_r2c_3d(int n0, int n1, int
n2,double *in, fftw_complex *out,unsigned flags);//三维
实数DFT

fftw_plan fftw_plan_dft_r2c(int rank, const int *n,double
*in, fftw_complex *out,unsigned flags);//rank维实数DFT

```

在复数据的 DFT 中，sign 表示了是做 DFT 变换(FFTW_FORWARD)还是 IDFT 变换(FFTW_BACKWARD)。另外，在每一个函数中，我们都能找到一个变量叫 flags，这个变量就是我们要重点讨论的策略生成方案。

flags 参数一般情况下常用的为 FFTW_MEASURE 或 FFTW_ESTIMATE。FFTW_MEASURE 表示 FFTW 会先计算一些 FFT 并测量所用的时间，以便为大小为 n 的变换寻找最优的计算方法。依据机器配置和变换的大小 (n)，这个过程耗费约数秒 (时钟 clock 精度)。FFTW_ESTIMATE 则相反，它直接构造一个合理的但可能是次最优的方案。总体来说，如果你的程序需要进行大量相同大小的 FFT，并且初始化时间不重要，可以使用 FFTW_MEASURE，否则应使用 FFTW_ESTIMATE。FFTW_MEASURE 模式下 in 和 out 数组中的值会被覆盖，所以该方式应该在用户初始化输入数据 in 之前完成。

下面我们测试一下对一个长度为 70000 的实数序列做 DFT，两种策略的效率有什么不同。

当我们只做一次 DFT 变换时：(注意，这里的时间不仅包括 fftw_execute()的时间，也包括生成策略 fftw_plan_dft_r2c_1d()的时间)

<pre>1.FFTW_MEASURE for ONE time. 2.FFTW_ESTIMATE for ONE time. 3.FFTW_MEASURE for 100 times. 4.FFTW_ESTIMATE for 100 times. 11.exit 1 FFTW_MEASURE for ONE time: 3937 2 FFTW_ESTIMATE for ONE time: 0 1 FFTW_MEASURE for ONE time: 0 2 FFTW_ESTIMATE for ONE time: 0</pre>	<pre>1.FFTW_MEASURE for ONE time. 2.FFTW_ESTIMATE for ONE time. 3.FFTW_MEASURE for 100 times. 4.FFTW_ESTIMATE for 100 times. 11.exit 2 FFTW_ESTIMATE for ONE time: 16 1 FFTW_MEASURE for ONE time: 3891 2 FFTW_ESTIMATE for ONE time: 0 1 FFTW_MEASURE for ONE time: 0</pre>
---	--

从上面的数据可以看出来，仅执行一次时，毫无疑问要选择 FFTW_ESTIMATE。而且在如果已经构造好了 FFTW_MEASURE 策略的情况下，FFTW_ESTIMATE 的执行效率能更高。

对于执行 100 次 DFT 的情况，我们测试结果如下：注意，这里我们去掉了生成策略的时间，因为从上面的测试中我们看到生成策略的时间是相当长的，而这段时间我们是可以通过后面的 wisdom 方法避免的。

```

1.FFTW_MEASURE for ONE time.
2.FFTW_ESTIMATE for ONE time.
3.FFTW_MEASURE for 1000 times.
4.FFTW_ESTIMATE for 1000 times.
11.exit

4
FFTW_ESTIMATE for 1000 times: 1016

3
FFTW_MEASURE for 1000 times: 937

4
FFTW_ESTIMATE for 1000 times: 922

3
FFTW_MEASURE for 1000 times: 922

```

从上面的测试中我们发现，对于执行多次的情况，我们可以使用 `FFTW_MEASURE` 来提高运行效率。

4、对 plan 的重用

从上面的测试中我们看到，生成一个 plan 有多么不容易，多么费时间，那么，我们可不可以生成一个 plan，然后在其他的情况下重用这个 plan 呢？答案当然是肯定的。

当然，重用也是有条件的：

- 输入输出数据的大小相等。
- 输入输出的数据对齐不变。按照前面所说，都是用 `fftw_malloc()` 就不会有问题了。
- 变换类型、是否原位变换不变。

具体函数接口如下所列。

```

void fftw_execute_dft(const fftw_plan p, fftw_complex *in,
    fftw_complex *out);

void fftw_execute_dft_r2c(const fftw_plan p, double *in,
    fftw_complex *out);

void fftw_execute_dft_c2r(const fftw_plan p, fftw_complex
    *in, double *out);

```

五、接触 wisdom

1、wisdom 是什么

在上面的测试中，我们已经发现，生成一个策略有多么不容易，多么费时间，那么，除了重用 plan 这样的方法能够减少生成策略的次数，还有没有别的办法来减少生成策略所花费的时间呢？答案当然也是肯定的。这就是我们这里需要介绍的 wisdom。

wisdom 的大体思路就是把生成好的策略相关的配置信息存储在磁盘里，然后在下次重新运行程序的时候，把策略相关的配置信息重新载入到内存中，这样在重新生成 plan 的时候就可以节约大量的时间。

2、如何使用 wisdom

由于 FFTW 起初是在 linux 环境下开发的,当我们需要在 windows 下使用 wisdom 的时候,我们需要在头文件 `fftw3.h` 中添加以下代码。

```
static void my_fftw_write_char(char c, void *f)
{ fputc(c, (FILE *)f);}

#define fftw_export_wisdom_to_file(f)
    fftw_export_wisdom(my_fftw_write_char, (void*)(f))

static int my_fftw_read_char(void *f)
{ return fgetc((FILE *)f);}

#define fftw_import_wisdom_from_file(f)
    fftw_import_wisdom(my_fftw_read_char, (void*)(f))
```

如果您需要使用其他的精度,则添加代码的方法类似。

wisdom 存储起来的不是 plan 本身,而是和 plan 相关的配置信息,例如内存、寄存器等。所以我们在把 wisdom 载入到内存中后,我们还是需要调用生成 plan 的函数的。使用 wisdom 的基本代码框架如下所示。

```
/* 导出 wisdom */

fftw_complex *in = NULL, *out = NULL;
fftw_plan p;
in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)
                                * row * col);
out = (fftw_complex*)fftw_malloc(sizeof(fftw_complex)
                                * row * col);

p = fftw_plan_dft_2d(row, col, in, out, FFTW_FORWARD,
                    FFTW_MEASURE);

FILE *fp = fopen("fftw_plan_dft_2d.txt", "w");
fftw_export_wisdom_to_file(fp);
fclose(fp);

fftw_destroy_plan(p);
fftw_free(in);
fftw_free(out);
```

```

/* 导入wisdom */

Fftw_complex *in = NULL, *out = NULL;
fftw_plan p;

in = (fftw_complex*)fftw_malloc(sizeof(fftw_complex)
    * row * col);
out = (fftw_complex*)fftw_malloc(sizeof(fftw_complex)
    * row * col);

FILE *fp = fopen("fftw_plan_dft_2d.txt", "r");
fftw_import_wisdom_from_file(fp);
fclose(fp);

p = fftw_plan_dft_2d(row, col, in, out, FFTW_FORWARD,
    FFTW_MEASURE);
for(int i=0; i<row*col; i++)
{
    in[i][0] = (double)i*i + 1;
    in[i][1] = (double)i/3;
}
fftw_execute(p);

fftw_destroy_plan(p);
fftw_free(in);
fftw_free(out);

```

3、在使用 wisdom 的时候需要尤其注意的几点

- 不管是 `fftw_import_wisdom_from_file(FILE *fp)` 还是 `fftw_export_wisdom_to_file(FILE *fp)`，里面的文件指针 `fp` 在调用函数前应该是打开的，在调用函数之后也是打开的，需要我们调用函数将它关闭。
- 每一个 **wisdom** 都是针对某一个确定的 **processor** 而言的。每次更换了运行的硬件环境，都应该重新生成 **wisdom**。
- 每一个 **wisdom** 都是针对某一个确定的程序而言的。每次修改了程序，即最后的二进制文件不同，都需要重新生成 **wisdom**。如果不重新生成 **wisdom**，相对于硬件环境的改变引起的效率下降，这里的效率下降没有那么明显。
- 对相同的 **processor** 和相同的 **program binary**，在每次运行的时候，也会因为虚拟内存使用的不同而导致程序执行效率的改变。这一条对于性能要求特别严格的情况，开发者需要考虑。

六、其他

FFTW 不仅支持 DFT 变换, 还支持 DCT(I-IV)和 DST(I-IV)以及 DHT 变换, 由于时间所限, 这部分内容不再赘述, 如果有需要, 可以参看英文版得使用手册。

另外, FFTW 可以多线程执行 DFT 变换, 但是多线程存在线程同步问题, 这可能会降低性能。所以除非问题规模非常大, 一般并不能从多线程中获益。

七、总结

FFTW 是当今世界上公认的最快的 FFT 算法, 已经受到越来越多的科学研究和工程计算工作者的普遍青睐, 并为量子物理、光谱分析、音视频流信号处理、石油勘探、地震预报、天气预报、概率论、编码理论、医学断层诊断等领域提供切实可行的大规模 FFT 计算。

本文并不是英文原版的官方使用手册的翻译, 而是笔者根据自己学习和使用 FFTW 的心得和体会, 自己设计的手册的章节顺序, 以利于完全没有接触过 FFTW 的中国学生学习和使用。另外, 由于不是官方使用手册的翻译, 所以, 一来难免有错误, 希望读者与笔者取得联系; 二来读者在阅读的过程中, 如果遇到笔者没有完全阐述清楚的问题, 可以查阅英文原版官方的手册, 也希望读者与笔者取得联系, 以完善该手册。

最后, 感谢您阅读本手册, 期待您的更多的好的建议。

参考文献:

- [1] Matteo Frigo and Steven G. Johnson, "The Design and Implementation of FFTW3," Proceedings of the IEEE 93 (2), 216–231 (2005)
- [2] <http://www.fftw.org/>
- [3] http://www.fftw.org/fftw3_doc/
- [4] <http://en.wikipedia.org/wiki/FFTW>
- [5] [http://en.wikipedia.org/wiki/J. H. Wilkinson Prize for Numerical Software](http://en.wikipedia.org/wiki/J._H._Wilkinson_Prize_for_Numerical_Software)
- [6] <http://www.docin.com/p-71818494.html>