第一部分、FFTW 介绍

一、FFTW 介绍

FFTW 由麻省理工学院计算机科学实验室超级计算技术组开发的一套离散傅立叶变换(DFT)的计算库，开源、高效和标准 C 语言编写的代码使其得到了非常广泛的应用，Intel 的数学库和 Scilib(类似于 Matlab 的科学计算软件)都使用 **FFTW** 做 **FFT** 计算。

FFTW 是计算离散 Fourier 变换(**DFT**)的快速 C 程序的一个完整集合。

1、它可计算一维或多维、实和复数据以及任意规模的 DFT；甚至包括正弦/余弦变换和离散哈特莱变换(DHT)。

2、FFTW 输入数据长度任意。

3、FFTW 支持任意多维数据。

4、FFTW 支持 SSE、SSE2、 Altivec 和 MIPS 指令集。

5、FFTW 还包含对共享和分布式存储系统的并行变换。

二、FFTW 的基本结构：

FFTW 不是采用固定算法计算变换，而是根据具体硬件和变换参数来调整使用不同算法，以期达到最佳效果。因此，变换被分成两个阶段。首先，FFTW 规划针对目标计算机的最快变换的计算途径，并生成一个包含此信息的数据结构。然后，对输入数据进行变换。该规划可以被多次使用。在一个典型的高性能应用中，总是在执行相同参数条件的任务，因而，相对复杂但结果可被重复使用的初始化是值得的。另一方面，当你需要某一参数的单次变换，初始化就显得过于费时。基于此 FFTW 提供基于启发式和先例的快速初始化。总的来说，FFTW 的一个显著特点就是，对某一参数类型的单次变换优势不大，但对于参数相同的多次变换具有更快的平均速度。

FFTW 为了加快用户的使用集成速度，提供了三种不同层次的接口。

1. 连续数据的单一变换的基本接口。
2. 计算多重和步进阵列数据的高级接口。
3. 支持通用数据布局、多重和步进的顶级接口。

大部分的用户使用基本接口就可以满足需要，顶级接口需要更小心使用以避免出错，因此需要花更多的时间去理解掌握。FFTW 不仅提供了数据自适应，还提供了高级用户定制功能。例如，由于代码空间不足，可以去掉用户不需要的功能代码。相反，FFTW 还可以拓展数据结构。

第二部分、FFTW 的数据类型

2.1、数据

任何调用 FFTW 的程序都要包含它的头文件 fftw3.h 及其库(在 windows 下共享库 fftw-3.2.1.dll)。

2.1.1、复数类型

FFTW 使用包含两个元素的 double 型数据来表示一个复数，第一个元素表示实部，第二个元素是虚部。

```
typedef double fftw_complex[2];
```

2.1.2、精度

FFTW 默认使用双精度浮点进行运算，用户可以使用 float 和 long double 来改变运算精度，虽然更高精度的数据进行运算理论上会得到更高精度的结果，但是由于软件系统和硬件的限制往往适得其反，因此用户要结合软硬件 系统来选择 FFTW 的数据类型。

2.1.3、分配存储空间

```
void *fftw_malloc(size_t n);
void fftw_free(void *p);
```

FFTW 一般只是简单调用系统的函数来分配存储空间，通常也不必担心有重大的内存开销，但我们强烈建议使用该函数来为用户的数据分配存储空间。

第三部分、FFTW 基本接口介绍

3.1、基本接口

3.1.1. 一维复数 DFT

方案：选择最佳的变换算法获得变换结果。

使用 FFTW 计算一个大小为 N 的一维 DFT 很简单，步骤如下：

```
#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);

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

    ...
    fftw_execute(p); /* repeat as needed */

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

}
```

首先为输入输出数组分配空间，你可以使用任何方式进行分配，但我们建议使用 **fftw_malloc**,因之不但具有一般 malloc 功能还在支持 SIMD 指令的情况下能够提供数组对齐功能。数据是一个 **fftw_complex** 型数组。分配空间后，就可以创建方案了，该方案包括了执行变换的所有需要准备的信息。下面的函数就是用来创建方案的：

```
fftw_plan fftw_plan_dft_1d(int n,   fftw_complex *in,   fftw_complex *out,   int sign,
                           unsigned flags);
```

第一个参数 n 是你要进行变换的数据大小，n 可以是任何大小的正整数。接下来的两个参数是输入输出数组的指针。这两个指针可以相同，表示就地转换以节省空间。

第四个参数 sign，可以是 **FFTW_FORWARD(-1)**或 **FFTW_BACKWARD (+1)**，指出变换的方向，从技术角度讲，它是变换中的指数标志。

参数 flags 通常可以是 **FFTW_MEASURE** 或 **FFTW_ESTIMATE**，**FFTW_MEASURE** 指示 FFTW 计算几种 FFT 算法执行这个 n 大小的变换所花费的时间，从中找出最优算法。处理器所花费的时间决定于硬件性能和 n 大小。**FFTW_ESTIMATE** 则相反，它只是建立一个 FFTW 认为合理的方案（也许不是最优的）。说白了，如果你想使用同一大小相同类型的输入数据执行多次变换，那么初始化建立方案的时间被均摊而变得不那么重要了，选择 **FFTW_MEASURE**；否则选择 **FFTW_ESTIMATE**。在选择 **FFTW_MEASURE** 创建方案的过程中，输入输出数据的数据将被覆盖，因此用户应该在初始化使用变换数据前完成变换方案的创建。

一旦变换方案创建完成你就可以刷新使用你的输入/输出数组而执行多次变换，实际的变换计算工作由 **fftw_execute(plan)**来完成：

```
void fftw_execute(const fftw_plan    plan);
```

如果你想对相同大小不同数据值的数组进行变换，你可以使用 **fftw_plan_dft_1d** 创建一个新的变换方案，FFTW 将会自动使用之前的方案数据以节约时间。

完成变换后调用 **fftw_destroy_plan(plan)**销毁方案：

```
void fftw_destroy_plan(fftw_plan plan);
```

使用 **fftw_malloc** 创建的数组需要调用 **fftw_free** 释放，而不是使用普通的内存释放函数（或者禁用、删除）。DFT 结果存放在输出数组里，零频率直流响应放在 **out[0]**中。如果输入和输出不是同一数组，则输出不会引起输入数据的修改（反之变换结果将覆盖输入数据）。

用户需要注意的是 FFTW 不是使用通常的 DFT 算法。而是计算一个正向接着一个逆向变换(反之亦然)而得出结果。可以参考 What FFTW Really Computes。如果你有一个支持 C99 的编译器如 GCC，你可以包含**#include <complex.h>**，使用双精度复数类型来进行运算。另外，除了 **FFTW_ESTIMATE** 和 **FFTW_MEASURE** 标志，还有许多其它的标志存在，例如，如果可以忍受更长的时间，你可以使用 **FFTW_PATIENT** 标志来生成一个更有效率的方案（见 FFTW Reference）。你甚至可以将方案保存起来留待以后的应用，见 Words of Wisdom-Saving Plans。

3.1.2. 一维实数 DFT

在许多应用场合，输入数据是实数序列，而输出满足“**Hermitian**”冗余：**out[i]**是 **out[n-i]** 的共轭。可以利用这个特点来改善速度和内存使用量。

为了换取速度和空间的优势，用户需要牺牲一些简单的 FFTW 复数变换内容。首先，输入和输出的数组大小会不同：输入是大小为 n 的实数组，输出是 n/2+1 的复数组(去掉了冗余的输出)；此外还要求输入数据的就地转化需要填充。第二点，逆变换(复数到实数)默认情况下会破坏输入数组。这些不便对于用户来说可能不是个严重的问题，但是，了解这些问题很有必要。

实数的变换程序和复数几乎是相同的：分配空间(最好使用 **fftw_malloc)**，创建一个 **fftw_plan** 并可使用 **fftw_execute(plan)**多次执行该方案，使用 **fftw_destroy_plan(plan) (and fftw_free)**清理和销毁方案。唯一的不同是输入的数据类型为双精度实数型，并且使用一个不同的程序来创建方案。一维：

```
fftw_plan fftw_plan_dft_r2c_1d(int n,    double *in,    fftw_complex *out,    unsigned flags);
fftw_plan fftw_plan_dft_c2r_1d(int n,    fftw_complex *in,    double *out,    unsigned flags);
```

实数输入复数输出(r2c)和复数输入实数输出(c2r)不同于复数 DFT，没有 sign 参数。r2c DFTs 总是 **FFTW_FORWARD**，而 c2r DFTs 总是 **FFTW_BACKWARD**.。这里 n 是逻辑大小，而不需要数组的物理大小。需要特别说明的是实数组有 n 个元素，复数组有 n/2+1 个元素。对于

就地转换来说，输入和输出数组放在一起，需要保证数组足够大，因此实数组的长度因该是2(n /2+1)。

综上，c2r 变换即使在非就地转换的情况下也会破坏输入数组的数据内容，如果有必要可以使用 **FFTW_PRESERVE_INPUT** 这个 flag 来避免，缺陷是会牺牲一定的性能。该标志目前还不支持多维实数 DFT。

熟悉实数 DFT 的读者知道，输出复数组的第 0 个元素(直流)和第 n/2 个元素(当 n 为偶数时叫“Nyquist”频率)完全是实数。因此一些实现为了使输入和输出数据长度一致，将“Nyquist”元素放到了直流元素的虚部里。然而这种实现不能够很好的概括多维变换，而且节省下来的空间有限，因此 FFTW 不支持。

一维 r2c 和 c2r DFTs 的另一种接口在 r2r 中体现(见:The Halfcomplex-format DFT)，那种半复数格式的输出与输入数据的类型和长度相同。那种接口对于多维实数变换也不常用，某些情况下可能会产生较好的效果。

第四部分、FFTW 创建方案(计划)函数的 flags 参数说明

4.1、方案的 flags 参数

FFTW 中所有的方案程序都支持 flags 参数，该参数采用位或的方法提供 0 到多参数的配置。这些参数严格(且有时效限制)的控制方案的生成过程，并可以加入和取消一些支持的算法限制。

注意：在创建方案之后一定要初始话输入数组，除非使用一个之前保存的方案，因为在方案的生成过程中，输入数组有被改写。唯一的例外是 **FFTW_ESTIMATE** 和 **FFTW_WISDOM_ONLY flag**，下面会提到。

在所有情况下，如果之前保存过相同或者更大规模的方案，可以导入之。例如在 **FFTW_ESTIMATE** 模式下可以导入所有的已存方案，而在 **FFTW_PATIENT** 模式下只有 patient 或 exhaustive 模式下创建的方案可以使用。详见 Words of Wisdom-Saving Plans。

4.2、方案设计方式 flags

FTW_ESTIMATE：规定根据输入数据长度等信息直接挑选一个算法(可能不是最优的)来，而不是去计算比较各种算法得到，因而方案创建的比较快，且不会覆盖输入输出阵列。

FFTW_MEASURE：告诉 FFTW 通过几种算法计算输入阵列实际所用的时间来确定最优算法。根据你的机器硬件配置情况，可能需要点时间(通常为几秒钟)，该参数是默认选项。

FFTW_PATIENT：类似 FFTW_MEASURE，但进行更广泛的算法筛选来确定更理想的方案（特别是对于大规模的变换任务），同时也会带来方案的创建时间成倍增加。

FFTW_EXHAUSTIVE：类似 FFTW_PATIENT，但进行更为广泛的算法筛选，即使是通常认为并不理想的算法也不放过，其结果相较于后者可能更优秀但耗时无疑会更长。

FFTW_WISDOM_ONLY：一个特殊的方案设计模式，该模式的方案仅当任务支持 wisdom 时才会被创建，否则返回 NULL。该模式通常与其他模式一起工作，如“**FFTW_WISDOM_ONLY|FFTW_PATIENT**”将会在支持 wisdom 的情况下创建一个基于 FFTW_PATIENT 模式的方案。在用户需要判断任务是否支持 wisdom 时 使用 FFTW_WISDOM_ONLY，比如，用户的任务在不支持 wisdom 的情况下可以重新创建一个方案以避免用户的数据被覆盖。

4.3、算法限制 flags

FFTW_DESTROY_INPUT：允许就地转换(out-of-place)的变换可以覆盖输入阵列为任意内容，该模式有时候会得到更优秀的方案。

FFTW_PRESERVE_INPUT：规定就地转换(out-of-place)的变换不能更改输入阵列。通常情

况下，特别是 r2c 和 hc2r(复数 到实数)，**FFTW_DESTROY_INPUT** 是默认模式。其余的情况，将通过使用 FFTW_PRESERVE_INPUT 使得变换方案放弃那些会破坏 输入阵列的算法。对于二维 c2r 变换，方案将返回 NULL，即该任务不支持 **FFTW_PRESERVE_INPUT** 模式。

FFTW_UNALIGNED：禁止算法发布任何对输入输出阵列进行对齐的请求(SIMD 禁用)。通常情况下该模式很少使用，因为方案会自动检测失调阵列。唯一使用此模式的是当你用一个老的方案去对新的不同长度的阵列进行变换时的用来对齐(用 fftw_malloc 分配空间的数据不必使用该模式)。

4.4、方案限时

```
extern void fftw_set_timelimit(double seconds);
```

该函数在方案中规定大约最大用时多少秒。如果参数 seconds ==FFTW_NO_TIMELIMIT(默认值，负数)，则不设限时。反之，创建方案过程中会对不同的方案模式挨个测试，直到测试完成或超时，然后从中挑出 最佳方案。例如，使用 FFTW_PATIENT 模式，如果加上限时，FFTW 会先测试 FTW_ESTIMATE 模式，然后 FFTW_MEASURE 模式，如果时间允许最后是 FFTW_PATIENT 模式。如果设定的是 FFTW_EXHAUSTIVE 模式，FFTW 最终还要测试 FFTW_EXHAUSTIVE 模式。

注意，seconds 参数只是一个粗略的限制；实际上，可能由于方案创建过程被放在了一个不能被打断的操作中间，导致花费更多的时间而产生超时。不过起码方案会工作在 FFTW_ESTIMATE 模式下(相当于超时时间设定为 0)。

第五部分、FFTW 使用方案(创建计划)

5.1、使用方案

所有的变换类型方案信息存放在 **fftw_plan**(一个不透明的指针类型)，该类型存放着包括输入输出数据指针的所有变换所需信息。

创建方案不仅仅收集数据信息，还包括对输入数据的分析，从多种不同特点的算法中选择最适合的一种或几种并进行数据预处理，稍后的执行变换仅按照方案提供的信息执行变换即可，FFTW 将变换分成方案和执行两部分对于相同类型方案，不同输入数据的多次变换，可以省略前面的步骤，重复调用执行变换即可，从而在不损失变换精度和速度的前提下令平均变换效率大大提高。

```
void fftw_execute(const fftw_plan plan);
```

在不改变 fftw_plan 的前提下，上面的函数可以多次执行，因而可以处理连续数据。

fftw_execute 是 FFTW 中唯一线程安全的函数。

```
void fftw_destroy_plan(fftw_plan plan);
```

上面的函数用来释放方案包括其相关的数据。