# FFTW

## 1 简介

FFTW (Faster Fourier Transform in the West) 是一个快速计算离散傅里叶变换的标准C语言程序集，其由MIT的M.Frigo 和S. Johnson 开发。可计算一维或多维实和复数据以及任意规模的DFT。

在使用FFTW之前，首先要对其内部结构有个基本的认识，我们要明确的一点是：FFTW不是在使用一个固定的算法来计算傅里叶变换。它会根据底层硬件的信息，自适应的采用DFT算法以便达到最大的性能。因此，FFTW计算傅里叶变换分两步进行：第一步，FFTW中的planner会根据机器的情况，“学习”到一个最快的方式来计算变换，在这一过程Planner会产生一个名为plan的数据结构来包含相关信息。第二步就是执行这个plan来对输入数据做变换。

## 2 安装

1.下载fftw-3.3.8.tar.gz (www.fftw.org)

2.tar zxvf fftw-3.3.8.tar.gz

3../configure

4.make

5.make install

在第3步可以加一些选项，比如：

–prefix=/usr/local/fftw(设定安装目录) –disable-fortran(忽略Fortran调用的机制) –enable-shared=yes(为了生成动态库文件) –enable-threads(多线程FFTW) –enable-mpi(mpi做并行变换)

# 3 FFTW的使用

## 3.1 一维复数据DFT

使用FFTW来计算长度为N的复数序列的DFT，通常情况下代码具有如下形式：

```
#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);
}
```

首先是一个新的数据类型，fftw_complex。在默认情况下FFTW的所有浮点数都是双精度double，fftw_complex实际上就是由两个double组成的：

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

[0]元素存储实部，[1]元素存储虚部。但如果你在inclued$\langle fftw3.h\rangle$之前先include$\langle complex.h\rangle$，那么fftw_complex的定义就是native complex type(c99)，这样你就可以进行一些常规的运算操作，例如：x=y*(3+4*I),其中，x与y都是fftw_complex，I是虚数单位。

FFTW使用fftw_malloc给输入输出数组分配内存，在最后使用fftw_free来释放内存。这里也可以使用malloc来分配内存，但最后必须使用free来释放，不能混用。这里推荐使用fftw_malloc，这样传递给FFTW的数据在内存中是

对齐的，保证了在一些处理器上可以进行SIMD，在某些情况下可以起到加速作用。

接下来调用fftw_plan_dft_1d创建一个plan：

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

第一个参数n，表示数组长度。第二和第三个参数是指向输入数组与输出数组的指针。第四个参数sign用来指示傅里叶变换的“方向”，正变换FFTW_FORWARD(-1)，逆变换FFTW_BACKWARD(+1)，指数的符号。最后一个参数flags，它可以有多种选择，我们这里列举若干个：

FFTW_ESTIMATE，在创建plan的过程中不会实际测量不同算法的执行时间，仅通过简单的启发式的方法来快速选择一个plan(有可能是次优的),这种flag在创建plan的过程中不会重写输入或输出数组。

FFTW_MEASURE，通过实际计算几个fft并测量它们的执行时间来找到一个优化的计划，这可能花费一些时间(通常几秒钟，取决于你的机器)。

FFTW_PATIENT，与FFTW_MEASURE类似，但搜索的范围更大(会尝试更多fft算法)，同时花费的时间也更多。

FFTW_EXHAUSTIVE,与FFTW_PATIENT相似，搜索范围更广，耗时更多。

需要注意的一点是，上述这些flags，除了FFTW_ESTIMATE，其余的都会在创建plan的过程中覆盖输入数组，因此我们最好在创建plan后再初始化输入。

当计划创建完成后，我们调用fftw_execute执行plan，最后释放资源。

## 3.2 多维复数据的DFT

```
fftw_plan fftw_plan_dft_2d(int n0, int n1,
                           fftw_complex *in, fftw_complex *out,
                           int sign, unsigned flags);
fftw_plan fftw_plan_dft_3d(int n0, int n1, int n2,
                           fftw_complex *in, fftw_complex *out,
                           int sign, unsigned flags);
fftw_plan fftw_plan_dft(int rank, const int *n,
                        fftw_complex *in, fftw_complex *out,
                        int sign, unsigned flags);
```

多维复数据DFT与一维复数据DFT的流程基本相同：使用fftw_malloc分配内存，创建fftw_plan，执行计划fftw_execute，最后cleanup.不同之处在于plan的创建。

### 3.3 一维实数据的DFT

在许多实际应用中，输入数据in[i]都是实数，在这种情况下，输出满足out[i]与out[n-i]是共轭的。FFTW利用这些特性，在速度和内存使用上实现了一些改善。

与一维复数据不同，在对于实数据DFT，输入与输出数组的类型(type)和大小(size)是不同的：输入是n个实数，而输出是n/2+1个复数。

但是一维使数据DFT的算法流程与复数据基本相同：为数组分配内存(double或fftw_complex)，创建fftw_plan，执行fftw_execute(plan)，最后cleanup。

```
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);
```

### 3.4 多维实数据的DFT

```
fftw_plan fftw_plan_dft_r2c_2d(int n0, int n1,
                               double *in, fftw_complex *out,
                               unsigned flags);
fftw_plan fftw_plan_dft_r2c_3d(int n0, int n1, int n2,
                               double *in, fftw_complex *out,
                               unsigned flags);
fftw_plan fftw_plan_dft_r2c(int rank, const int *n,
                            double *in, fftw_complex *out,
                            unsigned flags);
```

# 4 FFTW关于DFT的定义

## 4.1 1d DFT

正向(FFTW_FORWARD)离散傅里叶变换(DFT)，对于一维的复数组X(size n):

$$Y_k = \sum_{j=0}^{n-1} X_j e^{-2\pi jk\sqrt{-1}/n} .$$

逆变换(FFTW_BACKWARD)DFT:

$$Y_k = \sum_{j=0}^{n-1} X_j e^{2\pi jk\sqrt{-1}/n} .$$

FFTW计算的是未标准化的变换，在DFT的求和号前面没有系数。换句话说，对输入X应用正变换和逆变换后将输入X将乘以n。

## 4.2 1d real-data DFT

在FFTW中，real-input(r2c)DFT，对于一维的实数组X(size n)，做forward transform，定义与之前相同。假设输出数组为Y，则Y具有“Hermitian” symmetry，即Y[k]与Y[n-k]共轭。由于这种对称性，输出的Y有一半是冗余的，因此1d r2c transform只输出Y[0]到Y[n/2]，除法向下取整。

对于一个具有“Hermitian” symmetry的复数组X(size n)，c2r transform计算backward DFT，定义与之前相同。在这种情况下，输出的数组Y是一个实数组。

同样的，这里的变换也没有标准化，对输入X应用正变换和逆变换后将输入X将乘以n。

# 5 保存plans(wisdom)

FFTW实现了将plan保存到磁盘并且可以重新加载的方法，这种机制被称为wisdom。当通过FFTW_MEASURE，FFTW_PATIENT，或者FFTW_EXHAUSTIVE这些选项来创建plan时，我们所创建的plan会具有比较好的性能，但是这会花费一些时间，因为FFTW必须测量许多可能的计划的运行时间，并选择最佳的计划。对于FFTW_ESTIMATE，虽然初始化的时间短，但做变换时的效率并不高。而wisdom机制允许我们只计算好的plan(创建时耗时，但做变换时的效率高)一次，将它保存到磁盘中，之后根据需要可以重新加载若干次。

每当创建一个plan，FFTW的planer就会累计wisdom(一些信息)。在完成创建后，可以将这些信息保存到磁盘中：

```
int fftw_export_wisdom_to_filename(const char *filename);
```

在下一次跑其他程序的时候，我们可以重新加载之前保存的wisdom，然后重新创建plan，但要保证创建的plan与之前使用相同的flags:

```
int fftw_import_wisdom_from_filename(const char *filename);
```

Wisdom可以自动的用于任何合适的size，也就是说在创建plan时参数n可以改变，但flags不能比之前“强”。如果我们是在FFTW_MEASURE选项下保存的wisdom，那么之后新创建的plan必须是FFTW_ESTIMATE或者FFTW_MEASURE，这样wisdom才会被使用。

Wisdom所需的存储空间并不大，但如果想释放与其相关的内存，可以调用：

```
void fftw_forget_wisdom(void);
```

# 6 并行FFTW

## 6.1 多线程FFTW

多线程FFTW和之前相比只多了两步操作，第一就是线程的初始化：

```
int fftw_init_threads(void);
```

第二就是在创建一个plan之前，需要调用：

```
void fftw_plan_with_nthreads(int nthreads);
```

这里的参数nthreads是用来指定我们希望FFTW使用的线程数。

## 6.2 FFTW with MPI

对于MPI程序，首先要使用MPI_Init进行初始化，之后我们要调用fftw_mpi_init()。这步操作要在调用任何"fftw_mpi_"程序之前进行。

我们使用fftw_mpi_plan_dft_2d来创建一个plan，相比之前，这里多了一个参数MPI_COMM_WORLD,这个参数用来指示哪些进程将参与计算(MPI_COMM_WORLD指示所有进程都将参与计算)。

在FFTW MPI程序中，使用ptrdiff_t来代替int.

最重要的一点是，我们使用fftw_mpi_local_size_2d来给数组分配内存，这个函数会找出每个进程上包含了哪些数据，并且需要分配多少空间。在这个例子中，每个进程上都包含了整个数组的一个切片(local_n0×N1),并且索引是从local_0_start开始的。

例如，如果我们想执行一个分布在4个进程上的100×200二维复DFT，每个进程将得到25×200的切片。也就是说，进程0将得到0到24行的数据，进程1将得到行25至49，进程2将得到行50至74，进程3将得到行75至99。

```
#include <fftw3-mpi.h>

int main(int argc, char **argv)
{
    const ptrdiff_t N0 = ..., N1 = ...;
    fftw_plan plan;
    fftw_complex *data;
    ptrdiff_t alloc_local, local_n0, local_0_start, i, j;

    MPI_Init(&argc, &argv);
    fftw_mpi_init();

    /* get local data size and allocate */
    alloc_local = fftw_mpi_local_size_2d(N0, N1, MPI_COMM_WORLD,
                                         &local_n0, &local_0_start);
    data = fftw_alloc_complex(alloc_local);

    /* create plan for in-place forward DFT */


  plan = fftw_mpi_plan_dft_2d(N0, N1, data, data, MPI_COMM_WORLD,
                              FFTW_FORWARD, FFTW_ESTIMATE);

    /* initialize data to some function my_function(x,y) */
    for (i = 0; i < local_n0; ++i) for (j = 0; j < N1; ++j)
       data[i*N1 + j] = my_function(local_0_start + i, j);

    /* compute transforms, in-place, as many times as desired */
    fftw_execute(plan);

    fftw_destroy_plan(plan);

    MPI_Finalize();
}
```