# A short tutorial on the basic usage of the package **FFTW3**.

Tutorial version 1.0.

Jordi-Lluís Figueras

figueras@maia.ub.es

22th March 2010

# About this document

This tutorial is designed as a short introduction to the basic usage of the library `FFTW3`. The purpose of it is to save some time to the reader while he is introduced to the basics of this library. By no means it is intended to substitute the great tutorial that can be found in [3], so we encourage to go through it in order to master it.

For any suggestions, comments, remarks or whatever, you can send me an email to the address that appears in the front page.

The author of this tutorial kindly aknowledge FFTW team and its authors. Visit `www.fftw.org` or read [7] for more details.

# Contents

# 1 Introduction

In this tutorial we will introduce the C-library FFTW3, [3], which is used in order to compute **F**ast **F**ourier **T**ransforms, **FFT**. Before that, let's introduce some basic facts and notations.

Let $f\colon \mathbb{T} \to \mathbb{C}$, where $\mathbb{T} = \mathbb{R}/\mathbb{Z}$, be a function. We define its *Fourier series* as

$$\sum_{k=-\infty}^{\infty} c_k e^{2\pi k i \theta}, \tag{1}$$

where the coefficients $c_k$ are determined by the integrals

$$c_k = \int_{\mathbb{T}} e^{-2\pi k i \theta} f(\theta) d\theta. \tag{2}$$

*Remark* 1.1. If the function $f$ is quite regular, for example $\mathcal{C}^1(\mathbb{T})$, then the series (1) converges uniformly, so (1) represents the function $f$. In this tutorial we will not cover the wide area of convergence of these series. For a deeper exposition of these facts the reader can consult [1] and [2].

In order to compute the coefficients $c_k$ of (1) in a fast way we use the algorithms implemented in the library FFTW3. This library has implemented the computation of these coefficients using Fast Fourier Transform (FFT) routines, so...

**What is FFT?**

**Answer** FFT is an algorithm to compute in a fast way **D**iscrete **F**ourier **T**ransforms, **DFT**, so...

**What is DFT?**

**Answer** DFT is simply the computation of the coefficients $c_k$, the integrals (2), using the trapezoidal integration formula, that is, if $x_0, x_1, \ldots, x_{N-1}$ are $N$ complex numbers that represents $f(n/N) = x_n$, then

$$c_k = \frac{1}{N} \sum_{n=0}^{N-1} x_n e^{-ik\frac{2\pi n}{N}}. \tag{3}$$

*Remark* 1.2. A quick reference of DFT and FFT can be found in Wikipedia, [4], [5].

Now that we know what we will compute, let's see two applications of the DFT:

- Given $N$ points $x_0, x_1, \ldots, x_{N-1}$, compute the trigonometric polynomial

$$c_0 + c_1 e^{2\pi i\theta} + \cdots + c_{N-1} e^{2(N-1)\pi i\theta} \tag{4}$$

that interpolates the points $(0, x_0)$, $(\frac{1}{N}, x_1)$, $(\frac{2}{N}, x_2)$, ..., $(\frac{N-1}{N}, x_{N-1})$.

- Evaluate the trigonometric polynomial (4) in all the abscissae $0, \frac{1}{N}, \cdots, \frac{N-1}{N}$.

Now, a question comes in mind...

**Now that I know what is a DFT, can you repeat, please, what is a FFT?**

**Answer** A FFT is a fast way to compute DFT. If we analyse the computational cost of computing $c_0, c_1, \ldots, c_{N-1}$, via the direct implementation of (3), we see that it is $O(N^2)$ but the computational cost of FFT is, in the best cases, of order $O(NlogN)$ and, in the worst cases, the same as the previous one.

*Remark* 1.3. The rule of thumb about the computational time of the FFT is that it works better when the number $N$ is a power of 2 and it works bad when $N$ is a prime or it has as divisors big primes. For more details see [5].

One thing that the reader must have in mind is that the library `FFTW3` does not compute expressions like (3) but like the following ones:

- (No normalized) forward DFT:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-ik\frac{2\pi n}{N}}. \tag{5}$$

- (No normalized) backward DFT:

$$x_k = \sum_{n=0}^{N-1} X_n e^{ik\frac{2\pi n}{N}}. \tag{6}$$

*Remark* 1.4. Expression (5) differs from (3) by the constant $\frac{1}{N}$ and expression (6) is the inverse of (5) up to constant $\frac{1}{N}$.

## 1.1 Getting and installing FFTW3

In the webpage `www.fftw.org` can be found the source code of `FFTW3`. There it is explained how can be installed this package but, in most `Linux` environments, the following works:

1. Download the source code `fftw-X.X.X.tar.gz` from

   `ftp://ftp.fftw.org/pub/fftw/fftw-X.X.X.tar.gz`

2. Decompress it:

   ```
   tar -xvvf fftw-X.X.X.tar.gz
   ```

3. Enter the directory `fftw-X.X.X`:

   ```
   cd fftw-X.X.X
   ```

4. Now, proceed to configure, compile and install the package:

   ```
   ./configure && make && make install
   ```

   If the above does not work, read carefully the documentation that is in `www.fftw.org`.

# 2 Examples

In this section we expose some example codes where I try to show the basic usage of FFTW3 library. The text below and the codes associated to it are intimately related, so in order to understand the text one needs to check the codes (sorry for this!). The codes of the examples can be found in my webpage www.maia.ub.es/~figueras. Feel free to download these codes.

## 2.1 Example 1: Basic usage

An example of a C code, EXAMPLE1_basicusage.c, that use the library fftw3 must be like

```
1   #include<fftw3.h>
    int main(void)
    {
2       int N;
3       fftw_complex *in, *out;
4       fftw_plan my_plan;

5       in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*N);
6       out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*N);
7       my_plan = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD, FFTW_ESTIMATE);

8       fftw_execute(my_plan);

10      fftw_destroy_plan(my_plan);
11      fftw_free(in);
12      fftw_free(out);

13      return 0;
    }
```

To compile this code with gcc we just type in a console

```
gcc EXAMPLE1_basicusage.c -lfftw3 -o EXAMPLE1_basicusage
```

Let's explain in detail the above code:

- Line 1 includes the header file fftw3.h needed in order to use the package.

- Line 2 contains an integer N which has the dimension of the input and output data of the FFT.
- Line 3 declares two pointers of type fftw_complex, in and out, which will contain the input and output of the FFT. Note that to allocate memory we use the function fftw_malloc instead of the stdlib.h function malloc.
- Line 4 declares a variable of type fftw_plan, a *plan*, which will store the type of FFT that we want to perform.
- Lines 5 and 6 allocates memory for the pointers in and out. Note that it must be specified that they are of type fftw_complex.
- Line 7 declares the type of plan which we want to perform via the function fftw_plan_dft_1d which has as arguments
  1. int N: the dimension of the pointers in and out.
  2. fftw_complex *in: the pointer that stores the input data.
  3. fftw_complex *out: the pointer that stores the output data.
  4. int FFTW_FORWARD: FFTW_FORWARD is an integer constant of the package that tells to the function that the FFT to perform must be the forward one. To perform the bacward one, we will introduce FFTW_BACKWARD. To see the differences between these two, the reader can refer to section 1.
  5. unsigned FFTW_ESTIMATE: FFTW_ESTIMATE is a flag that tells to the function how well must be optimized, with respect to the computational time, the FFT algorithm. If we are getting started to the package we will use this flag. For other values of this flag the reader can consult appendix A.
- Line 8 performs the FFT stored in my_plan.
- Lines 10, 11 and 12 deallocate the memory stored by the plan and the pointers. Note that for the pointers we use fftw_free and not the stdlib.h function free.

## 2.2 Example: Basic usage expanded

This example, which can be found in the code EXAMPLE2_transform.c, performs two explicit FFT of an input data of dimension 10 of the form $x_k = (k+1)+(3k-1)i$, $0 \leq k < 10$ for the first FFT and $x_k = e^{-k}$ for the second FFT. The code is very similar to the one explained in section 2.1.

To compile EXAMPLE2_transform.c use

```
gcc EXAMPLE2_transform.c -Wall -lfftw3 -lm -o EXAMPLE2_transform
```

*Remark* 2.1. The `gcc` command has more libraries, `-lm`. Note that the order of the flags `-lfftw3` and `-lm` is intended: in some `Unix` systems, if we transpose this two it can happen that the compilation produce an error and does not produce the executable file.

One of the differences between this code and the one in section 2.1 is that in the header files we have included the library `complex.h`. This is because, although `fftw3.h` has implemented its own complex routines, if we declare `complex.h` before it then we can use the, more common, syntax of it. So, for example, when we introduce the input data in the pointer `in`, we do

```
in[i] = (i+1.)+(3.*i-1.)*I;
```

*Remark* 2.2. `I` is the imaginary part in the library `complex.h`.

Note that we have performed two FFT by simply executing the line

```
fftw_execute(plan);
```

*Remark* 2.3. This tell us that, if we do some other code, not this one, where we are planning to do a lot of FFT of the same type with the same dimension and input and output pointers, we will change the flag `FFTW_ESTIMATE` in the planning creator function `fftw_plan_dft_1d` for a more appropiate one. See appendix A for more details.

## 2.3 Example: Real trigonometric polynomial

This example, associated to the code `EXAMPLE3_cosine_sine.c`, generates aleatory the real coefficients of a real a trigonometric polynomial

$$A_0 + \sum_{k=1}^{2} A_k \cos(2\pi k\theta) + \sum_{k=1}^{2} B_k \sin(2\pi k\theta).$$

and, using the command-line input a positive integer `Npoints`, it evaluates this polynomial (using non-FFT algorithm) in a `Npoints` equidistant grid of $[0, 1]$. This evaluation is stored in a pointer and then performed a forward FFT. Then, the ouput is transformed in order to obtain the coefficients of the polynomial in real trigonometric form.

### 2.4 Example: Product of two trigonometric polynomials

This example, `EXAMPLE4_product.c` is the most ambitious of this tutorial. It creates two $N$ degree random polynomials and performs, via FFT, their product, obtaining a $2N$ degree polynomial. To see the theoretical argument of this, the reader can consult [6]. Basically, the program performs the $2N$ FFT of the two polynomials, multiply the points obtained and recover the product polynomial by the inverse FFT.

One thing to comment of the code of this program is that at the end of the it it appears the function `fftw_cleanup()`. Although we have freed the memory using the functions `fftw_free` and `fftw_destroy_plan`, at the end of the execution the library has not freed all the memory. This function does this.

## 3 Arrived at this point...

Arrived at this point, this short tutorial ends. There ar a lot of things that the author has not explained. Some of them are:

- Special FFT of real input, output or both data.
- Other fast algorithms that perform special transformations.
- FFT of $d$ dimensional data.
- The usage of `wisdom`. A feature that helps to optimize the computation of FFT by checking, via some tests, several FFT.

Nevertheless, thank you very much for reading this tutorial!!!

# A Planner flags

There are different planner flags. For a full discussion of them the reader can consult the documentation that appears in the website [3].

Basically, the planner flags tell to `FFTW3` package how much we want to spend computing the FFT. If we demand to omptimize the computational time, we must pay the cost that the optimization takes some time so, in practice, we will demand more optimization when we want to perform more FFT of the same time.

The flags are ordered in increasing optimization scale:

1. `FFTW_ESTIMATE`.
2. `FFTW_MEASURE`.
3. `FFTW_PATIENT`.
4. `FFTW_EXHAUSTIVE`.

# References

[1] *Fourier series*, Wikipedia
`http://en.wikipedia.org/wiki/Fourier_series`.

[2] Y. Katznelson, *An introduction to harmonic analysis*, Cambridge University Press, 2004.

[3] *FFTW library* ,
`http://www.fftw.org`.

[4] *Discrete Fourier Transform*, Wikipedia
`http://en.wikipedia.org/wiki/Discrete_Fourier_transform`.

[5] *Fast Fourier Transform*, Wikipedia
`http://en.wikipedia.org/wiki/Fast_Fourier_transform`.

[6] *Convolution Theorem*, Wikipedia
`http://en.wikipedia.org/wiki/Convolution_theorem`.

[7] Matteo Frigo and Steven G. Johnson, *The design and implementation of FFTW3,* Proc. IEEE 93 (2), 216231 (2005).