Efficient convolution using the Fast Fourier Transform, Application in C++

Jeremy Fix

April 26, 2013

Contents

1 Introduction

Convolution products are often encountered in image processing but also in other works such as evaluating a convolutive neural network. A naive implementation of a convolution product of signals of size N involves an order of N^2 operations. An efficient implementation of convolution product can be designed by making use of the Fast Fourier Transform and mathematical properties linking convolution products and Fourier transforms. This can boil down the complexity of computing a convolution product to an order of $N \log N$ operations. In this document, we briefly introduce the convolution product, Fourier transform, the use of Fourier transforms for performing convolution products and sample codes as well as benchmarks showing the performances of two libraries : GSL and FFTW.

All the benchmarks were done on an AMD Opteron 175 with GSL 1.13, FFTW 3.2.2 and gcc 4.4.3 under Ubuntu 10.04.

In the following we need to divide integers by two, sometimes takes its opposite. When writing $N/2$, we always consider it to be the largest integer smaller or equal to $N/2$ (e.g $3/2 = 1$). When writing $-N/2$, it is always the opposite of $N/2$ and not $-N$ divided by 2 (e.g. $-3/2 = -1$). Given this, the following results are used in the following: $N - \frac{N-1}{2} = \frac{N}{2} + 1, N - \frac{N+1}{2} = \frac{N}{2}$.

1.1 Convolution product : linear and circular

1.1.1 Definition

Consider two signals $f[n]$ of size N and $g[m]$ of size M. Suppose f is defined on $[0, N-1]$ and g is defined on $[0, M-1]$[1]. We use indices starting from 0 to be coherent with indices of C arrays. The discrete convolution product of these two sequences is defined as :

$$\forall k, h[k] = (f * g)[k] = \sum_{i=-\infty}^{\infty} f_\infty[i].g_\infty[k-i] = (g * f)[k] \tag{1}$$

where f_∞ and g_∞ are defined by :

$$\begin{aligned} f_\infty[n] &= \begin{cases} 0 & \text{if} \quad n < 0 \\ f[n] & \text{if} \quad 0 \leq n \leq N-1 \\ 0 & \text{if} \quad n > N \end{cases} \\ g_\infty[m] &= \begin{cases} 0 & \text{if} \quad m < 0 \\ g[m] & \text{if} \quad 0 \leq m \leq M-1 \\ 0 & \text{if} \quad m > M \end{cases} \end{aligned}$$

These signals correspond to the original signal f and g to which we added zeros on the left and right up to an infinite horizon. Obviously, the signals having a finite horizon (padded zeros up-to and starting from certain indices), the convolution is null up-to and starting from some indices (see fig. 1). Namely, the full convolution has only at most $N + M - 1$ non zero values :

$$h[k] = (f * g)[k] = \begin{cases} 0 & \text{if} \quad k < 0 \\ \sum_{i=\max(0,k-M+1)}^{\min(N-1,k)} f[i]g[k-i] & \text{if} \quad 0 \leq k \leq N+M-2 \\ 0 & \text{if} \quad k \geq N+M-1 \end{cases} \tag{2}$$

Also, some of the products involve padded values of f while others not. This leads to define, as in Matlab, three different results (see fig. 1):

- *Full* : The result with all the non-zero elements, of size $N + M - 1$

[1] Be careful, we use mathematical, also used for C arrays, indices running from 0 to $N-1$ for an array of size N, contrary to Matlab which uses indices running from 1 up-to N.

- *Same* : The result with the central components, of the same size N of f, all the indices for which the central element of g is multiplied by a non-padded value of f
- *Valid* : The result with the components that do not involve any padded zero of f, of size $N-M+1$

We get the following expressions for the three result types :

$$\textbf{Full}\ ,\forall k \in [0, N+M-2], \quad h_f[k] = \sum_{i=\max(0,k-M+1)}^{\min(N-1,k)} f[i]g[k-i] \tag{3}$$

$$\textbf{Same}\ ,\forall k \in [0, N-1], \quad h_s[k] = h_f[k+\frac{M}{2}] = \sum_{i=\max(0,k-\frac{M-1}{2})}^{\min(N-1,k+\frac{M}{2})} f[i]g[k+M/2-i] \tag{4}$$

$$\textbf{Valid}\ ,\forall k \in [0, N-M], \quad h_s[k] = h_f[k+M-1] = \sum_{i=k}^{k+M-1} f[i]g[k+M-1-i] \tag{5}$$

This corresponds to the **linear convolution**. As we saw previously, we may need to pad the f signal with zeros when required values of the signal are undefined. One may also define the **circular convolution**, denoted $*_N$, which does not require to pad the signal with zeros but wrap around the signal. This means that when an undefined value of the signal is required, e.g. $f[-1]$, we use the wrapped around value, i.e. $f[-1 \bmod N] = f[N-2]$. The circular convolution modulo N is depicted on figure 2 and defined formally as :

$$\begin{aligned}\forall k, (f \circledast_N g)[k] &= \sum_{i=-\infty}^{\infty} f_N[i].g_\infty[k-i] \\ &= \sum_{i=-\infty}^{\infty} f[i \bmod N].g_\infty[k-i] \end{aligned} \tag{6}$$

where f_N is the signal defined as repetitions of N, i.e. a periodic signal of period N. In the following we denote $\equiv$ the congruence relationship : $\forall a, b \in \mathbb{Z}^2, \forall n \in \mathbb{N}, a \equiv b[n] \Leftrightarrow \exists k \in \mathbb{Z}, a = b + nk$. The circular convolution is periodic of period N, therefore it is sufficient to compute it on one period. We can also restrict the infinite sum on the finite support of g (where $g \neq 0$):

$$\begin{aligned}\forall k \in [0, N-1], (f \circledast_N g)[k] &= \sum_{i=-\infty}^{\infty} f[i \bmod N].g_\infty[k-i] \\ &= \sum_{i=k-M+1}^{k} f[i \bmod N].g[k-i] \\ &= \sum_{i=0}^{M-1} f[(k-i) \bmod N].g[i] \end{aligned} \tag{7}$$

1.1.2 Computational complexity

Ajouter les comparaisons ou les modulos dans la complexit de la linaire ?
On n'est pas limit N ¡= M ; donc on peut refaire des benchs sur une grille
D'ailleurs, refaire les benchs, vu qu'on veut faire en taille N , M , on s'en fout d'images pour le moment

With the definitions of the standard convolutions given above, these operations roughly requires an order of $N \times M$ operations for a convolution of two signals of size N and M. More precisely :

- a linear convolution as defined by equation 4 requires approximately an order of $M \times N - M(M-1)/4$ products and $M \times N - M(M-1)/4 - 1$ sums, under the assumption that $M \leq N$ (which is usually the case in filtering),

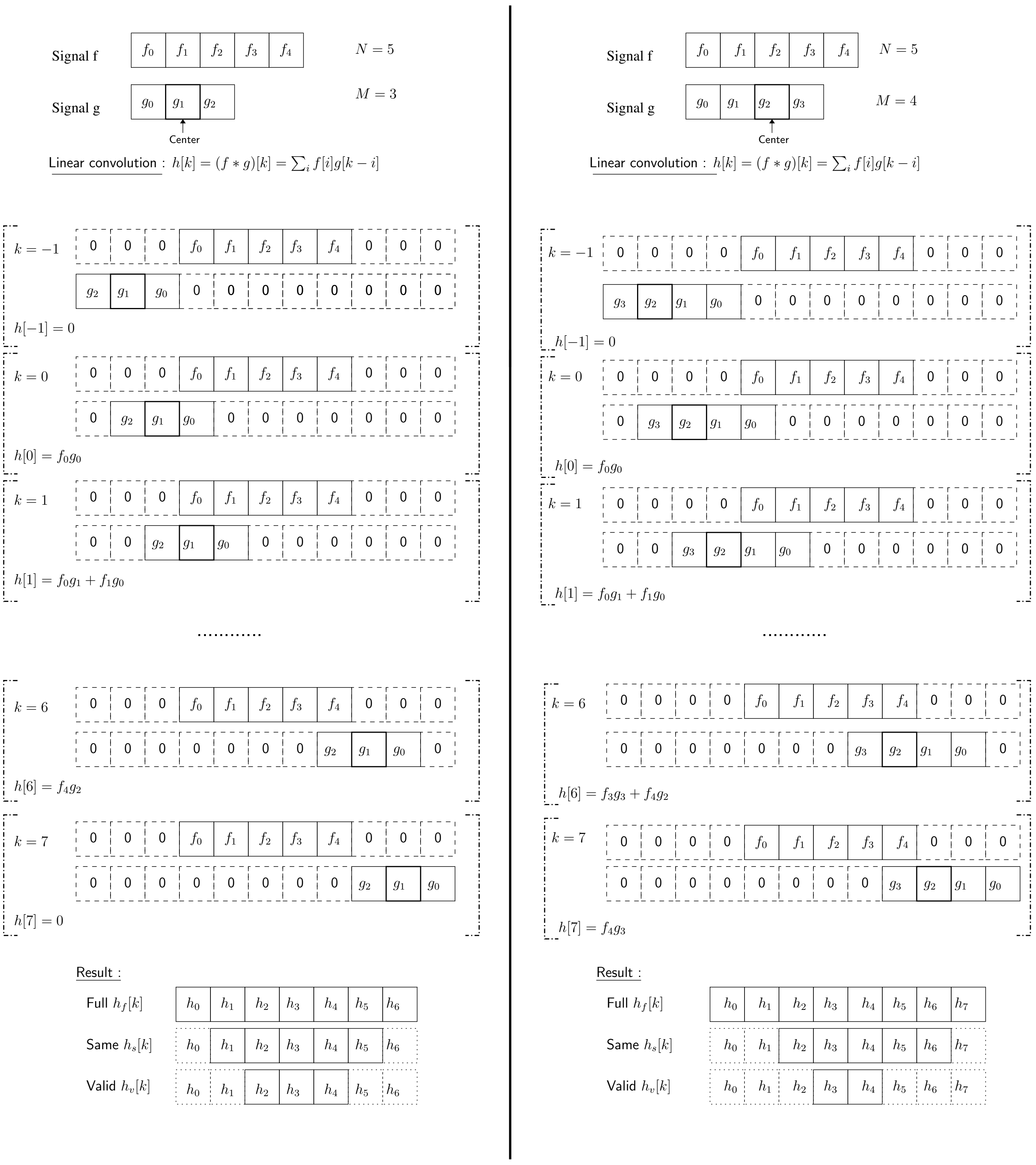

Figure 1: Schematic illustration of the linear convolution for an odd and even filter size

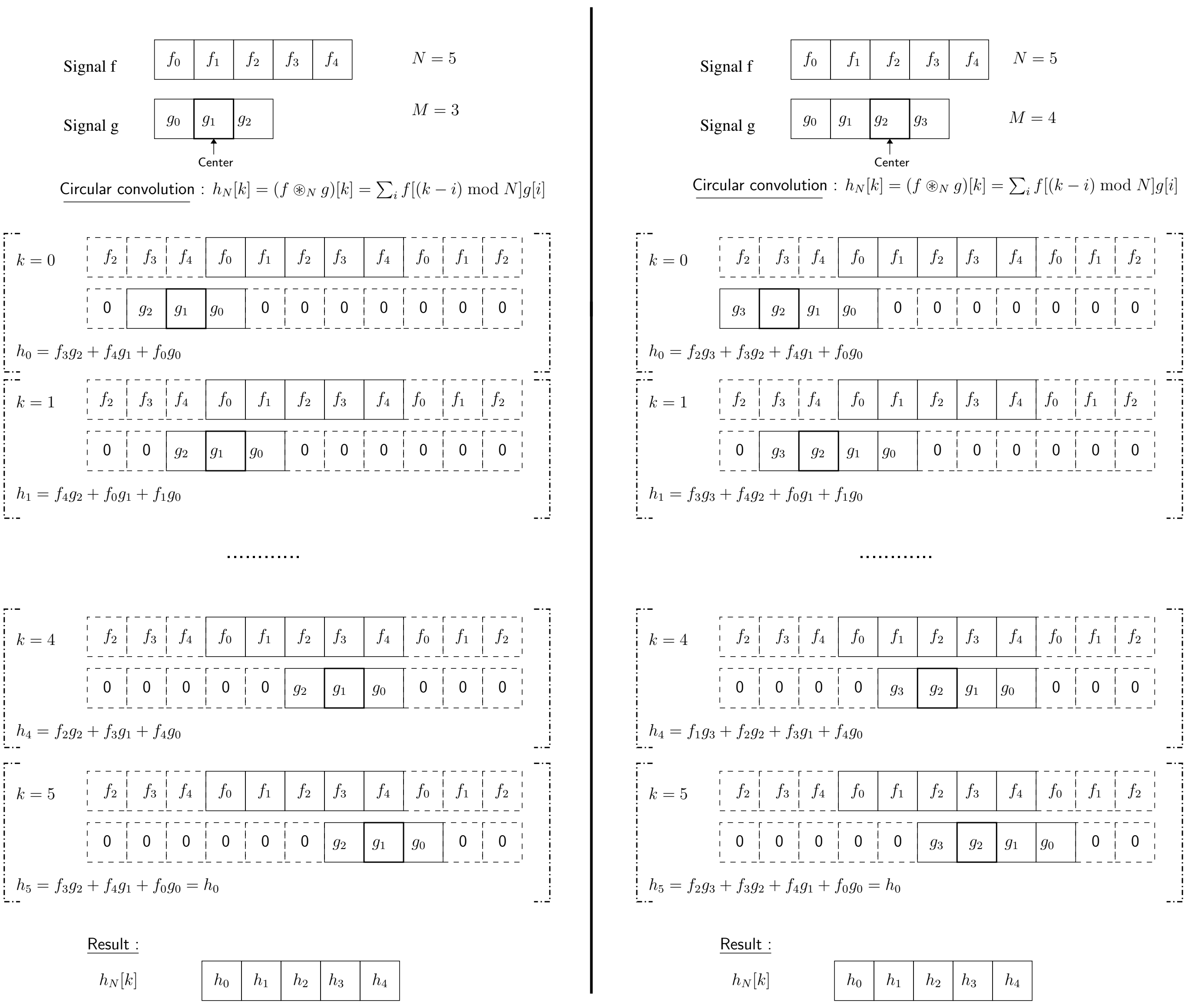
Signal f
f_0 f_1 f_2 f_3 f_4
$N = 5$
Signal g
g_0 g_1 g_2
$M = 3$
Center
Circular convolution : $h_N[k] = (f \circledast_N g)[k] = \sum_i f[(k-i) \bmod N]g[i]$
$k = 0$
$h_0 = f_3g_2 + f_4g_1 + f_0g_0$
$k = 1$
$h_1 = f_4g_2 + f_0g_1 + f_1g_0$
....
$k = 4$
$h_4 = f_2g_2 + f_3g_1 + f_4g_0$
$k = 5$
$h_5 = f_3g_2 + f_4g_1 + f_0g_0 = h_0$
Result :
$h_N[k]$
h_0 h_1 h_2 h_3 h_4
Signal f
f_0 f_1 f_2 f_3 f_4
$N = 5$
Signal g
g_0 g_1 g_2 g_3
$M = 4$
Center
Circular convolution : $h_N[k] = (f \circledast_N g)[k] = \sum_i f[(k-i) \bmod N]g[i]$
$k = 0$
$h_0 = f_2g_3 + f_3g_2 + f_4g_1 + f_0g_0$
$k = 1$
$h_1 = f_3g_3 + f_4g_2 + f_0g_1 + f_1g_0$
....
$k = 4$
$h_4 = f_1g_3 + f_2g_2 + f_3g_1 + f_4g_0$
$k = 5$
$h_5 = f_2g_3 + f_3g_2 + f_4g_1 + f_0g_0 = h_0$
Result :
$h_N[k]$
h_0 h_1 h_2 h_3 h_4

Figure 2: Schematic illustration of the circular convolution for an odd and even filter size

- a circular convolution as defined by equation 7 requires $N \times M$ products and $N \times M - 1$ sums and $N \times M$ modulo.

The figure 3 shows this execution time function of the signals' size and evaluated with the following script : std_convolution.cc.

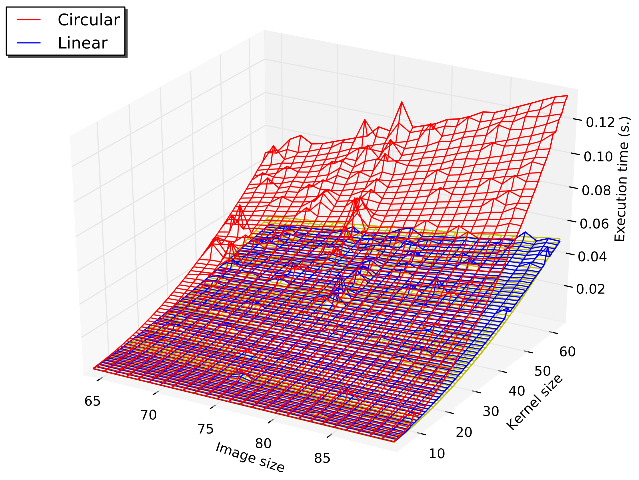

Figure 3: Execution time for computing a linear or circular convolution in C++, using nested for loops, of two signals of size N and M. The execution time for a circular convolution grows with $N \times M$. For the linear convolution, the sum is restricted on less indices, which leads to an execution time faster than $N \times M$

On the tested machine, these curves are well approximated by :

- $t = 5.8.10^{-9} N^{1.98} K^{1.93}$ for the circular convolution
- $t = 1.5.10^{-9} N^{2.38} K^{1.56}$ for the linear convolution

In the following sections, we introduce the Discrete Fourier Transform and *show* that it leads to an algorithm for computing a convolution product requiring only an order of $N \log N$ operations.

1.2 Discrete Fourier Transform

1.2.1 Introduction

The Discrete Fourier Transform of a discrete signal $f[n]$ of length N is defined as :

$$FT[f][k] = \hat{f}[k] = \sum_{n=0}^{N-1} f[n] e^{-\frac{2i\pi kn}{N}}$$

The discrete fourier transform of a discrete signal of length N is periodic of period N :

$$\begin{aligned}
\forall p \in \mathbb{Z}, \hat{f}[k+pN] &= \sum_{n=0}^{N-1} f[n] e^{-\frac{2i\pi(k+pN)n}{N}} \\
&= \sum_{n=0}^{N-1} f[n] e^{-\frac{2i\pi kn}{N}} e^{-\frac{2i\pi pNn}{N}} \\
&= \sum_{n=0}^{N-1} f[n] e^{-\frac{2i\pi kn}{N}} e^{-2i\pi pn} \\
&= \hat{f}(k)
\end{aligned}$$

The inverse discrete fourier transform of a discrete frequency signal $\hat{f}[k]$ is defined as :

$$IFT[\hat{f}][n] = f[n] = \frac{1}{N}\sum_{k=0}^{N-1} \hat{f}[k]e^{\frac{2i\pi kn}{N}}$$

The scaling factor $\frac{1}{N}$ is required to get the true inverse. To check this, let's compute $IFT[FT[f]][n]$ to check it equals to $f[n]$:

$$\begin{aligned}
IFT[FT[f]][n] &= \frac{1}{N}\sum_{k=0}^{N-1} FT[f][k]e^{\frac{2i\pi kn}{N}} \\
&= \frac{1}{N}\sum_{k=0}^{N-1}\sum_{m=0}^{N-1}\left(f[m]e^{-\frac{2i\pi km}{N}}\right)e^{\frac{2i\pi kn}{N}} \\
&= \frac{1}{N}\sum_{k=0}^{N-1}\sum_{m=0}^{N-1} f[m]e^{\frac{2i\pi k(n-m)}{N}} \\
&= \frac{1}{N}\sum_{m=0}^{N-1} f[m]\sum_{k=0}^{N-1} e^{\frac{2i\pi k(n-m)}{N}}
\end{aligned}$$

The last sum equals N when $n \equiv m[N]$ and 0 otherwise since it is a geometric serie of common ratio $e^{\frac{2i\pi(n-m)}{N}}$:
If $n \equiv m[N]$ (m is in [0, N-1]): $\sum_{k=0}^{N-1}(e^{\frac{2i\pi(n-m)}{N}})^k = \sum_{k=0}^{N-1} 1 = N$,
Otherwise : $\sum_{k=0}^{N-1}(e^{\frac{2i\pi(n-m)}{N}})^k = \frac{1-(e^{\frac{2i\pi(n-m)}{N}})^N}{1-e^{\frac{2i\pi(n-m)}{N}}} = \frac{1-e^{2i\pi(n-m)}}{1-e^{\frac{2i\pi(n-m)}{N}}} = 0$.
From the previous result, we get :

$$\begin{aligned}
IFT[FT[f]][n] &= \frac{1}{N}\sum_{m=0}^{N-1} f[m]\sum_{k=0}^{N-1} e^{\frac{2i\pi k(n-m)}{N}} \\
&= \frac{1}{N}(N.f[n \bmod N]) = f[n \bmod N]
\end{aligned}$$

Therefore : $IFT\ o\ FT = identity$ when restricted on the horizon [0,N-1]. In fact, we have built a periodic signal of period N which is repetitions of f.

In terms of complexity, we need an order of N^2 operations to compute the DFT of a signal of length N (N products for each of the N components). In the following sections, we first show how we can compute a n-dimensional DFT from only 1D DFT and then briefly introduces the Fast Fourier Transform which decreases the number of required operations down to an order of $N \log N$ for a performing a DFT of a signal of length N.

1.2.2 Computing a 2D Fourier transform from 1D Fourier transforms

A 2D Fourier transform can be computed from 1D fourier transforms. For sake of simplicity, let's write $w_P = e^{\frac{-2i\pi}{P}}$. The 2D Fourier Transform of a signal $s[n, m]$ of size (N, M) is defined as :

$$\begin{aligned}
S[k, j] &= \sum_{n=0}^{N-1}\sum_{m=0}^{M-1} s[n, m]e^{\frac{-2i\pi jm}{M}}e^{\frac{-2i\pi kn}{N}} && (8) \\
&= \sum_{n=0}^{N-1}\sum_{m=0}^{M-1} s[n, m]w_M^{jm}w_N^{kn} && (9) \\
&= \sum_{n=0}^{N-1}(\sum_{m=0}^{M-1} s[n, m]w_M^{jm})w_N^{kn} && (10)
\end{aligned}$$

It turns out that the 2D DFT can be computed by performing a 1D DFT on the rows of s (the inner sum is on n fixed) followed by a 1D DFT on the columns of the result. Let's denote $\hat{S}[n,j]$ the 2D signal for which the row n holds the 1D DFT of the line n of the original signal $s[n,m]$:

$$\hat{S}[n,j] = \sum_{m=0}^{M-1} s[n,m] w_M^{jm} \tag{11}$$

We can write :

$$S[k,j] = \sum_{n=0}^{N-1} \hat{S}[n,j] w_N^{kn} \tag{12}$$

The previous equation corresponds to a 1D DFT performed on the columns of $\hat{S}$. Therefore, to compute a 2D DFT of s you need to :

1. Perform a 1D DFT on the rows of s which leads to the 2D signal $\hat{S}$
2. Perform a 1D DFT on the columns of $\hat{S}$

The two previous operations can be done in the reverse order, no matter. This way of computing a 2D DFT from 1D DFTs is interesting because you can now compute DFT of n-dimensional signals involving only 1D-DFTs.

1.2.3 Computing 2 DFT at once

When we will apply the DFT to compute convolution products, we will need to compute two DFT : one for the image and one for the kernel. In fact, two DFTs can be simply computed with only one DFT. Given f and g are real signals, you can combine them to form a complex signal $h[n] = f[n] + ig[n]$. Given the DFT is linear : $FT[h][k] = FT[f][k] + iFT[g][k]$. The individual DFTs $FT[f]$ and $FT[g]$ are not the real and imaginary parts of $FT[h]$ given that the components of the DFTs are complex numbers. However, you know that :

$$f[n] = \frac{h[n] + (h[n])^*}{2}$$
$$g[n] = \frac{h[n] - (h[n])^*}{2i}$$

where c^* is the conjugate of the complex number c. Therefore, you can write :

$$FT[f][k] = \frac{FT[h][k] + FT[h^*][k]}{2}$$
$$FT[g][k] = \frac{FT[h][k] - FT[h^*][k]}{2i}$$

It is possible to express $FT[h^*][k]$ in terms of $FT[h][k]$ by simply observing that :

$$\begin{aligned} FT[h^*][k] &= \sum_{n=0}^{N-1} h^*[n] w_N^{kn} = \sum_{n=0}^{N-1} h^*[n] w_N^{kn} w_N^{-Nn} \\ &= \sum_{n=0}^{N-1} h^*[n] w_N^{-(N-k)n} = (\sum_{n=0}^{N-1} h[n] w_N^{(N-k)n})^* \\ &= (FT[h][N-k])^* \end{aligned}$$

We finally get :

$$\forall k \in [1, N-1] FT[f][k] = \frac{FT[h][k] + (FT[h][N-k])^*}{2}, FT[f][0] = \mathrm{Re}(FT[h][0])$$
$$\forall k \in [1, N-1] FT[g][k] = -i\frac{FT[h][k] - (FT[h][N-k])^*}{2}, FT[g][0] = \mathrm{Im}(FT[h][0])$$

This technique is not required if you want to convolve several images with the same kernel since, in that case, you can precompute the DFT of the kernel one for all.

1.2.4 Fast Fourier Transform

To compute a DFT according to the previous definition, we need an order of N^2 operations. There are some efficient algorithms that have been designed, which involve only an order of $N \log N$ operations. They are called Fast Fourier Transform algorithms. We do not provide here how they work but just some pointers to some of the FFT algorithms :

- Cooley-Tukey algorithm (Wikipedia) : breaks down a DFT of size $N = N_1 N_2$ into two DFT of size N_1 and N_2
- Prime-factor algorithm (Wikipedia) : breaks down a DFT of size $N = N_1 N_2$ into two DFT of size N_1 and N_2 when N_1 and N_2 have no common factors
- Rader's algorithm (Wikipedia)
- Winograd's algorithm

For the divide-and-conquer like algorithms (Cooley-Tukey, Prime-Factor), computing a DFT of a signal of size N can be done by computing fourier transforms of smaller sizes, in particular of sizes which correspond to the prime factors of N. Quite a lot of details can be found in the documentation of the GSL on FFT algorithms (take the archive online and browse in the doc directory to find the file fft_algorithms.texi) (see also the GSL documentation for some pointers). The FFTW library makes use of the Cooley-Tukey algorithm, the prime factor algorithm, Rader's algorithm for prime sizes, and a split-radix algorithm and they also propose a generator of adhoc FFT algorithms for specific sizes, which, according to their documentation "produces new algorithms that we do not completely understand".

1.2.5 Computational complexity

There are several librairies providing optimized methods for computing a DFT. To cite only two of them : the GSL (Gnu Scientific Library) and `FFTW3`. Others are cited on the FFTW website. The GSL proposes only 1D fourier transforms while FFTW3 proposes ND Fourier transforms. Therefore, for GSL, we will use the method presented in the previous section to compute a 2D Fourier Transform using only 1D Fourier transform. The scripts used to compute the Fourier Transform with the GSL and FFTW3 are `fft_gsl.cc` and `fft_fftw.cc`.

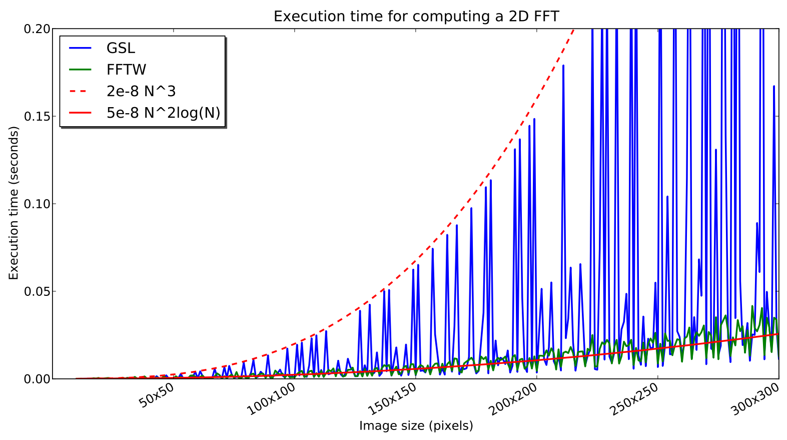

Figure 4: Execution time for computing a 2D FFT. With a signal of size NxN, the performancs of the FFTW is well approximated with a curve in $N^2 \log(N)$ while the performances of the GSL sometimes reach $(N^2)^{3/2}$.

The image 4 shows the time it takes, on an AMD Opteron 175, to compute the 2D-FFT for different signal sizes, the signals being randomly generated. The execution time fits with a $N \log(N)$ relationship. For some sizes, the GSL is performing worse than this relationship. The possible reason for this comes from how the GSL and the FFTW compute the FFT. As explained before, the algorithms rely on a factorization of the size of the signal. With the GSL, the implemented factors are $2, 3, 4, 5, 6, 7$ (see GSL website) and for FFTW the implemented factors are

$2, 3, 5, 7, 11, 13$ (see FFTW website). For the GSL, and apparently not the FFTW, if the size of the signal is not a product of only these factors, the performances can be worse than $N \log(N)$. In particular, this is the case for large prime numbers. If we look at the sizes where the performances are bad, they correspond to sizes like : 127x127, 131x131, .., 157x157, .. which are all prime numbers. It seems that FFTW handles successfully these large prime numbers.

2 2D Convolution product using the Discrete Fourier Transform

2.1 Introduction

In the previous sections, we introduced the linear and circular convolutions as well as the Fourier transform with its efficient implementation of the Fast Fourier Transform. Here, we detail the link between the fourier transform and the convolution product.

2.1.1 Convolution theorem : circular convolution and fourier transform

Consider we want to convolve an image of size (N, N) with a filter of the same size (N, N). As we saw in the introduction, a direct implementation of the convolution product with nested for loops requires an order of N^2N^2 operations. It is possible to speed up this computation by using the Fourier transform. This relies on the the convolution theorem : the circular convolution product of two signals equals the inverse Fourier transform of the product of the fourier transforms of the signals :

Theorem 1. *Let f and g be two discrete sequences of length N. The inverse fourier transform of the product of the fourier transform of f and g equals the circular convolution of f and g :*

$$\forall n \in [0, N-1], (f \circledast_N g)[n] = IFT[FT[f].FT[g]][n]$$

To show this, let's compute the fourier transform of the circular convolution of $f[n]$ and $g[n]$:

$$\begin{aligned}
FT[f \circledast_N g][k] &= \sum_{i=0}^{N-1} \left((f \circledast_N g)[i] w_N^{ki}\right) && (13)\\
&= \sum_{i=0}^{N-1} \left(\sum_{j=0}^{N-1} f[(i-j) \bmod N].g[j] w_N^{ki}\right) && (14)\\
&= \sum_{j=0}^{N-1} \left(g[j] \sum_{i=0}^{N-1} f[(i-j) \bmod N].w_N^{ki}\right) && (15)\\
&= \sum_{j=0}^{N-1} \left(g[j] w_N^{kj} \sum_{i=0}^{N-1} f[(i-j) \bmod N].w_N^{k(i-j)}\right) && (16)\\
& && (17)
\end{aligned}$$

The index of the inner sum can be changed to get rid of j since :

$$\forall j \in [0, N-1], \{(i-j) \bmod N, i = 0..(N-1)\} = \{i, i = 0..(N-1)\} \quad (18)$$

Therefore :

$$\begin{aligned}
FT[f \circledast_N g][k] &= \sum_{j=0}^{N-1} \left(g[j] w_N^{kj} \sum_{i=0}^{N-1} f[i].w_N^{ki}\right) && (19)\\
&= \left(\sum_{i=0}^{N-1} f[i].w_N^{ki}\right) \left(\sum_{j=0}^{N-1} g[j] w_N^{kj}\right) && (20)\\
&= FT[f][k].FT[g][k] && (21)
\end{aligned}$$

Using the inverse fourier transform, we finally get the convolution theorem. Computing a convolution product that way requires to compute 2 DFTs, their product, and an inverse DFT. As we saw previously, when one needs to compute 2 DFT of real signals, they can be computed using a single DFT. Also, computing the DFT of a signal of N requires an order of $N\log(N)$ operations. Adding the element-wise products, we get an algorithm for computing a convolution in $N(1 + \log(N))$. This is a significant improvement compared to the naive implementation which requires an order of N^2 operations.

2.1.2 Circular convolution of signals of different sizes

In the previous section, we saw that the circular convolution of two signals of same size can be efficiently computed with the Fourier Transform. What to do with the circular convolution of two signals of different size ?

Let's consider the circular convolution of two signals f and g of respectively size N and M if $N \neq M$. Let's first consider the case $N > M$ which is usually the case when one wants to filter an image for example. Denote g_∞ the signal g padded with zeros on both sides :

$$\forall n, g_\infty[n] = \left\{ \begin{array}{ccc} g[n] & \text{if} & 0 \leq n \leq M-1 \\ 0 & \text{otherwise} & \end{array} \right. \tag{22}$$

Let's compute the Fourier Transform of f, g_N and their pointwise product, as we did before but in the reverse order :

$$\begin{aligned}
\forall k \in [0, N-1], FT[f][k].FT[g_\infty][k] &= \left(\sum_{i=0}^{N-1} f[i] w_N^{ki}\right)\left(\sum_{j=0}^{N-1} g_\infty[j] w_N^{kj}\right) && (23)\\
&= \sum_{i=0}^{N-1}\left(\sum_{j=0}^{N-1} f[(i-j) \bmod N] g_\infty[j] w_N^{ki}\right) && (24)\\
&= \sum_{i=0}^{N-1}\left(\sum_{j=0}^{M-1} f[(i-j) \bmod N] g[j] w_N^{ki}\right) && (25)\\
&= \sum_{i=0}^{N-1} (f \circledast_N g)[i] w_N^{ki} && (26)\\
&= FT[f \circledast_N g][k] && (27)
\end{aligned}$$

Let's now consider the second case, when $M > N$. Let's rewrite sligthly the circular convolution in this case in order to get a circular convolution of two signals of the same size N :

$$\begin{aligned}
\forall k \in [0, N-1] (f \circledast_N g)[k] &= \sum_{i=0}^{M-1} f[(k-i) \bmod N] g[i] && (28)\\
&= \sum_{i=0}^{N-1} f[(k-i) \bmod N] \sum_{p=0}^{\lceil \frac{M}{N} \rceil} g_\infty[i+pN] && (29)
\end{aligned}$$

The $\lceil x \rceil$ denotes the smallest integer larger than x. Let's denote g_N the following signal, of size N :

$$\forall n \in [0, N-1], g_N[n] = \sum_{p=0}^{\lceil \frac{M}{N} \rceil} g_\infty[i+pN] \tag{30}$$

This signal is nothing more than slices of g, of size N that we add together (see fig. **??**). We can then write the circular convolution of f and g as a circular convolution of two signals of size N which we can compute using the Fourier Transform.

2.1.3 Circular convolution : summary

As a summary, we can now link the circular convolution as defined by equation 7 with the Fourier Transform :

Theorem 2. *Let f and g be two discrete sequences of respectively size N and M. The circular convolution of f and g, of size N, defined by $\forall k \in [0, N-1], (f \circledast g)[k] = \sum_{i=0}^{M-1} f[(k-i) \bmod N]g[i]$ can be computed using the Fourier Transform :*

$$\forall k \in [0, N-1], (f \circledast g)[k] = IFT[FT[f].FT[\tilde{g}]][k]$$

with :

$$\forall k \in [0, N-1] \tilde{g}[k] = \begin{cases} g_\infty[k] & \textit{if } M \leq N \\ \sum_{p=0}^{\lceil \frac{M}{N} \rceil} g_\infty[k+pN] \end{cases}$$

and

$$\forall k, g_\infty[k] = \left\{ \begin{array}{ccc} g[k] & \textit{if} & 0 \leq k \leq M-1 \\ 0 & \textit{otherwise} \end{array} \right.$$

2.1.4 Linear convolutions : particular cases of circular convolution

We now want to adress the question of computing a linear convolution with the Fourier Transform. Answering this question is quite easy. If we look at the illustrations of the linear and circular convolutions (figures 1 and 2), we see that the expression of the components of the two convolutions are the same if, in the case of a circular convolution instead of padding f with wrapped around values we padd it with zeros. More formally, let's suppose we want to linearly convolve two signals f and g of respectively size N and M. Let's now build the signal f_M, of size $N+M-1$ as:

$$\forall k \in [0, N+M-2], f_M[k] = \left\{ \begin{array}{ccc} f[k] & \text{if} & 0 \leq k \leq N-1 \\ 0 & \text{otherwise} \end{array} \right.$$

The f_M signal is somehow our f_∞ but here defined as a finite sequence. The circular convolution of f_M and g gives :

$$\forall k \in [0, N+M-2], (f_M \circledast_{N+M-1} g)[k] = \sum_{i=0}^{M-1} f_M[(k-i) \bmod (N+M-1)]g[i] \quad (31)$$

Given $i \in [0, M-1]$ and $k \in [0, N+M-2]$, then $k-i \in [-M+1, N+M-2]$. When $k-i$ is negative ($k-i \in [-M+1, -1]$), the wrap around leads to consider the indices $N \leq k-i+N+M-1 \leq N+M-2$ where f_M is null. Also, for $k \in [N; N+M-2]$, there are some indices $k-i$ which fall on parts where f_N is zero, in particular, when $k-i \in [N, N+M-2]$. Overall, the sum can be restricted to the indices i where $k-i \in [0, N-1]$ and $i \in [0, M-1]$, which means $i \in [0, M-1] \cap [k-N+1, k]$:

$$\begin{aligned} \forall k \in [0, N+M-2], (f_M \circledast_{N+M-1} g)[k] &= \sum_{i=\max(0,k-N+1)}^{\min(M-1,k)} f_M[(k-i) \bmod (N+M-1)]g[i] && (32) \\ &= \sum_{i=\max(0,k-N+1)}^{\min(M-1,k)} f[k-i]g[i] && (33) \end{aligned}$$

Now, we just need to change the summation index :

$$\begin{aligned} \max(0, k-N+1) \leq i \leq \min(M-1, k) &\iff k - \min(M-1, k) \leq k-i \leq k - \max(0, k-N+1) \\ &\iff \max(k-M+1, 0) \leq k-i \leq \min(k, N-1) \end{aligned}$$

Finally, we find the expression of the **Full** linear convolution (eq. 3):

$$\forall k \in [0, N+M-2], (f_M \circledast_{N+M-1} g)[k] = \sum_{i=\max(0,k-M+1)}^{\min(N-1,k)} f[i]g[k-i] = h_f[k]$$

If we want to compute the **Same** linear convolution (eq. 4), we need to pad the f signal with slightly less zeros. In fact we need to pad the signal with at least $\frac{M}{2}$ zeros on one end;

And restrict ???

2.2 Speeding-up by padding

Computing the DFT is faster for some specific sizes. There are some radix-2 algorithms working with images of sizes being a power of 2 but also mixed-radix algorithms which work with a factorization of the image's size. As we saw previously on figure 4, the performances can be really reduced if the image's size is non-optimal. In the two next paragaphs, we illustrate methods to change the size of the images to convolve in order to fall on optimal sizes for the FFT algorithms.

To determine if a size is optimal, we just factorize it and check if the factorization contains only factors for which sub-transforms are optimally computed. In GSL, this means only size that can be written $2^a * 3^b * 4^c * 5^d * 6^e * 7^f$. In addition, it seems that when the size is a multiple of $4.4.4.2 = 128$, the mixed-radii algorithm does not perform a FFT efficiently. Therefore, all the sizes multiple of 128 are considered non-optimal. In FFTW3, an optimal size is of the form $2^a * 3^b * 5^c * 7^d * 11^e * 13^f$, with $e + f$ equals 0 or 1. We also discard the sizes multiple of $4.4.4.2$ which also appeared to decrease the performances.

2.2.1 Linear convolution

To compute a linear convolution with the FFT, it is sufficient to pad the images with enough zeros, namely $\frac{N+1}{2}$. This bound is a lower bound. Therefore, we can pad the images with more than $\frac{N+1}{2}$ to get sizes that are optimally computed with the algorithm we use.

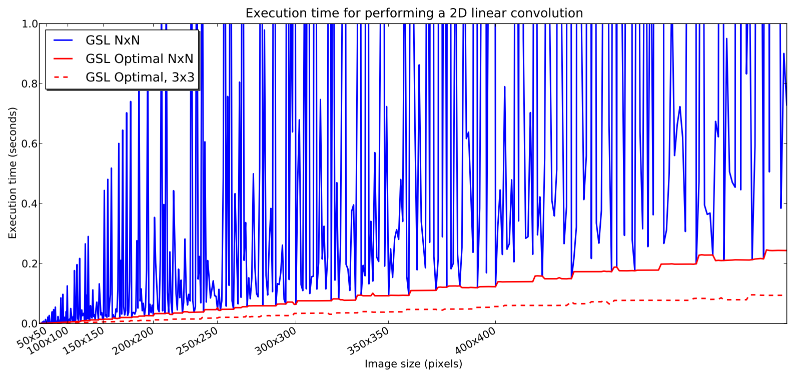

Figure 5: Execution time for performing a linear convolution with GSL. The times for both the unpadded and padded images are shown. Clearly, padding with zeros in order to get optimal sizes (in the sense of the factor decomposition of GSL) improves the performances.

The time it takes to compute a 2D linear convolution with or without padding, using the GSL (for which optimal sizes are of the form $2^a3^b4^c5^d6^e7^f$, see GSL website), is shown on the figure 5. It computes a 2D linear convolution of 2 randomly generated images of the same size. Padding clearly improves the processing time of the convolution. For this illustration, we used a small x-axis increment to better appreciate the effect of non-optimal sizes. The dashed line represents the execution time when the kernel size is small (actually 3x3). Since padding depends on the size of the kernel, the kernel size influences the size of the FFTs to compute. For convenience, only the times of the optimal version with a small kernel are plotted. This illustration was generated with the results produced by this script : linear_convolution_gsl_benchmark.cpp

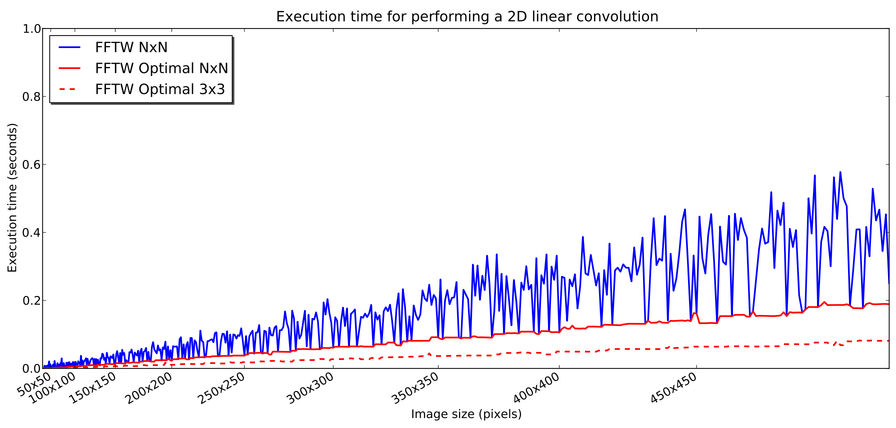

Figure 6: Execution time for performing a linear convolution with FFTW. The times for both the unpadded and padded images are shown. Clearly, padding with zeros in order to get optimal sizes (in the sense of the factor decomposition of FFTW) improves the performances.

The figure 6 shows the execution time for performing a 2D linear convolution with FFTW (for which optimal sizes are of the form $2, 3, 5, 7, 11, 13$,see FFTW website), with or without padding to get optimal sizes, using 2 randomly generated matrices of the same size. Again, we see that padding leads to better performances. This illustration was generated with the results produced by this script : linear_convolution_fftw_benchmark.cc.

The figure 7 compares the execution times with the GSL and FFTW implementations. The FFTW based implementation performs slightly better than the GSL implementation. It performs almost 20% better as soon as the image size is at least 200×200. The performances of the GSL are not as bad as we may have thought when we observed the bad performances of the GSL computing a FFT. It is simply because we are interested in a convolution product, therefore we are not obliged to stick to some specific frequencies and we can pad the signal as we want in order to get optimally computed sizes.

2.2.2 Circular convolution

We can also change the size of an image to optimally compute a circular convolution. For a circular convolution, instead of padding with zeros on both sides, we pad the image with wrapped around values. We then use the FFT to compute a circular convolution and extract the central part of the resulting image. The excluded regions, on the border of the result, will be meaningless since they combine the wrapped around pixel values with themselves but they are anyway discarded. In the code, it is a bit longer to write because we need a complex array in which the source image and the kernel are copied respectively in the real and imaginary parts. As we will see, padding does not necessarily improves the performances. This is because a circular convolution can be computed without padding. As soon as we want to pad the images to perform a circular convolution, we must pad with at least the size of the kernel (half the size on both sides). This can seriously increase the size of the images for the FFT.

On figure 8 (again involving 2 randomly generated images of the same size), padding does not always lead to better performances. When the size of the image is optimal, it is clear that the padded version must be worse. However, when the size is not optimal, padding can or cannot improves the performances. In this case, it is not clear how to define a rule deciding when we should pad or when we should not. We may define a simple rule : if the size is optimal, do not pad, otherwise pad. We may also perform tests before-hand to check which version is faster but this

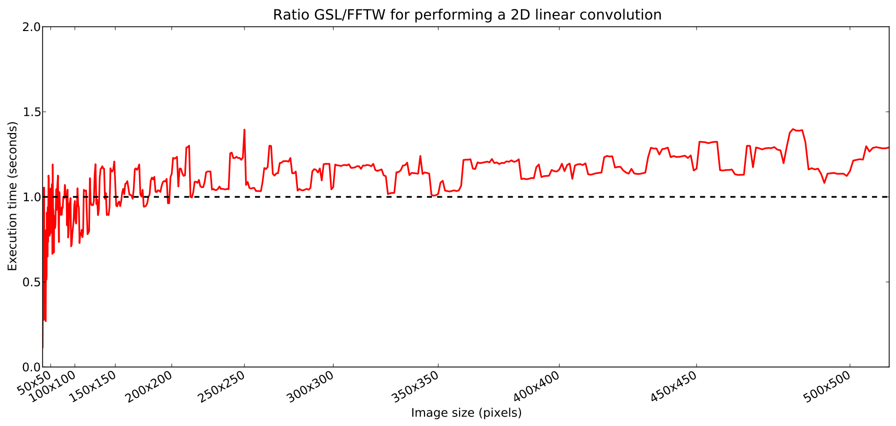

Figure 7: Comparison of the execution time for performing a linear convolution using optimal sizes with GSL or FFTW. The FFTW based implementation is slightly better, almost 20% faster for sizes at least 200×200.

is really dependent on the size of the images to convolve.

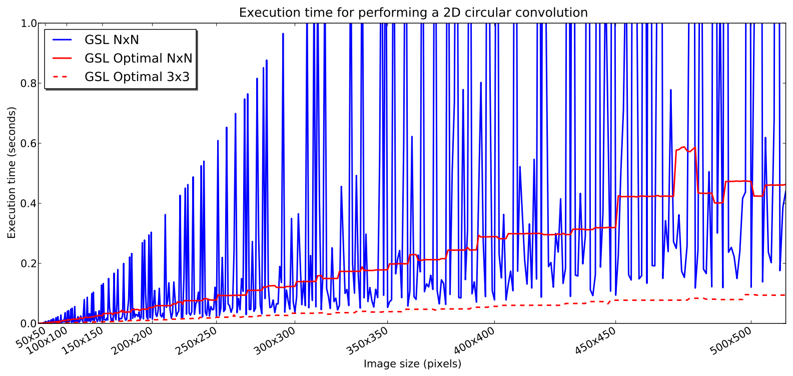

Figure 8: Execution time for performing a circular convolution with GSL. Two versions are plotted : with or without padding. The two curves of the padded version (solid and dashed) are the boundaries of the execution time, whatever the kernel size.

On the figure 9, we see that there is no benefits in padding to get optimal sizes with FFTW. Circular convolution is what is computed by default when using the product of the FFTs. Therefore, when we pad an image to get optimal sizes but still to compute a circular convolution, we cannot be better than the non-padded version for optimal sizes and it depends for non-optimal sizes on the performance of the algorithm for these sizes. It appears that FFTW has rather good performances even for sizes it is not optimized for. This illustration was generated with the following script : circular_convolution_fftw_benchmark.cc.

To compare the execution times, we took the smallest execution time of the padded or unpadded convolution with the GSL, and the unpadded convolution of FFTW, and computed their ratio. The results are shown on figure 10. The FFTW based version appears to be better than the GSL based implementation. The differences are higher than for the linear convolution, with performances almost 75% better, again for sizes at least 200×200.

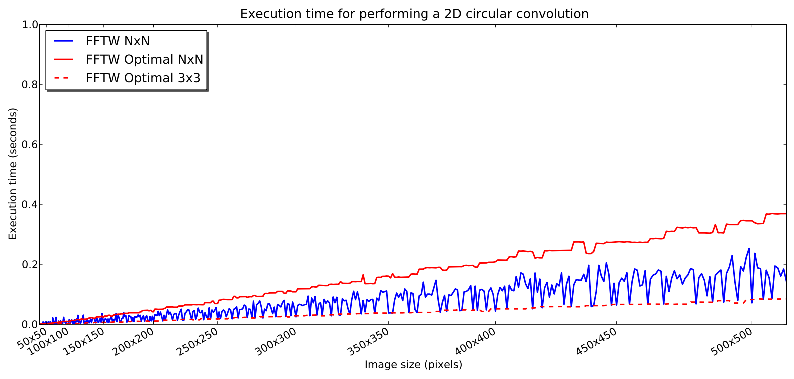

Figure 9: Execution time for performing a circular convolution with FFTW. Two versions are plotted : with and without padding with wrapped around values to get optimal sizes. For a circular convolution, padding makes the performances smoother but also worse !

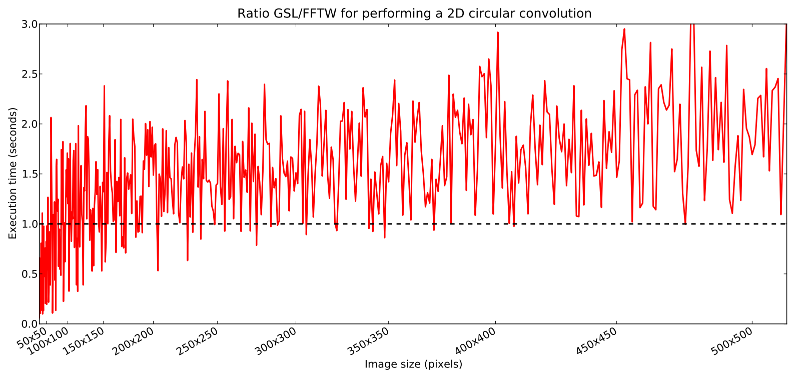

Figure 10: Comparison of the execution time for a circular convolution. For the GSL version, we used the minimum of the padded and unpadded times. For the FFTW version, we used the unpadded version. It seems that the FFTW version performs better.

2.3 Comparison between FFT based and standard convolution

The previous benchmarks allowed to detect which FFT based algorithm was faster and if padding improves or not the performances. However, it is not always beneficial to use the FFT to compute a convolution product, simply because, with the FFT, you need to "resize" the kernel to the image size and compute the products of the two FFT. When convolving an image of size $N \times N$ with a kernel of size $K \times K$, you work with FFT of size $N \times N$ and compute pointwise products of images of size $N \times N$ which may cost more time than convolving with nested *for loops*, especially if $K << N$.

To determine for which kernel size it is actually worth using the FFT, we ran again simulations of the FFT based convolution but now for different kernel sizes, to get a similar plot as the one for the standard convolution (fig. 3). We used the FFTW based implementation with the padded version for the linear convolution and the unpadded version for the circular convolution.

2.4 Putting everything together: 2D image Convolution using the FFT

The scripts given in the previous section were using randomly generated matrices to perform the benchmarks. To process images, we make use of the CImg library. You just need to get the header CImg.h and put it at the same place of the script you compile. The scripts allowing to convolve an image with a filter are given below. These scripts compute 2 DFT at once (the source image and the kernel) and pad them with zeros or wrapped around pixel values to get optimal sizes.

- 2D Linear convolution with GSL : linear_convolution_gsl.cc
- 2D Linear convolution with FFTW : linear_convolution_fftw.cc
- Circular convolution with GSL : circular_convolution_gsl.cc
- Circular convolution with FFTW : circular_convolution_fftw.cc

On the figures 11 and 12 we compare the time it takes to compute a linear or circular convolution for different image and kernel sizes using FFTW, GSL and the nested for loops. In the previous sections, the benchmarks were always done with 2 images of the same size. This is rather unusual. Usually, the filter size is much smaller than the image size. To get an idea of how well the convolution involving the FFT performs, the execution time of a standard convolution (involving for loops) is also given. The illustrations were generated with the results of linear_everything.cc and circular_everything.cc.

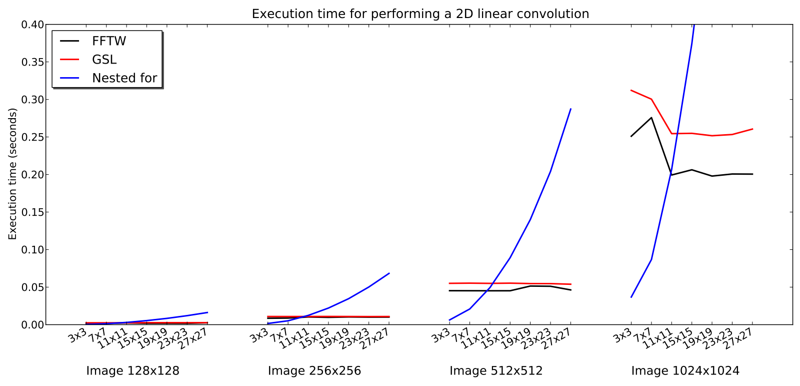

Figure 11: Comparison of the processing time of a linear convolution with different image and filter sizes. Using a FFT based implementation is beneficial only above a certain filter size.

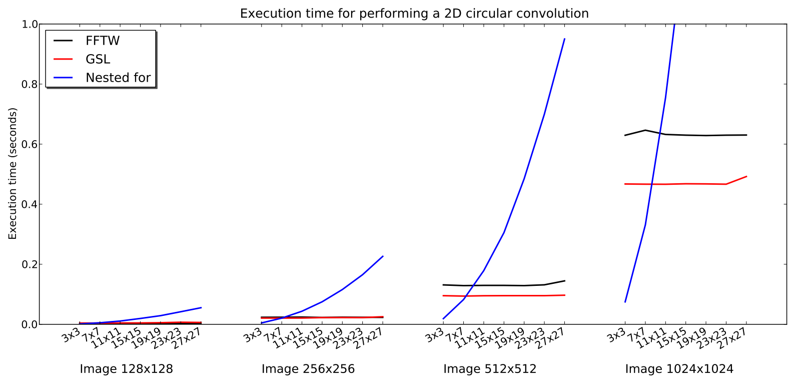

Figure 12: Comparison of the processing time of a circular convolution with different image and filter sizes. Using a FFT based implementation is beneficial only above a certain filter size.

3 C++ Implementations

Here, we just provide a list of the C++ scripts used through this document :

- Computing a 2D Fourier Transform :
  - with GSL : fft_gsl.cc
  - with FFTW : fft_fftw.cc
- Computing a linear convolution with nested for loops : std_convolution.cc
- Computing 2DFT at once with FFTW : 2DFT.cpp
- Computing a 2D linear convolution of randomly generated matrices with or without padding:
  - with GSL : linear_convolution_gsl_benchmark.cc
  - with FFTW : linear_convolution_fftw_benchmark.cc
- Computing a 2D circular convolution of randomly generated matrices with or without padding:
  - with GSL : circular_convolution_gsl_benchmark.cc
  - with FFTW : circular_convolution_fftw_benchmark.cc
- Computing a 2D linear convolution with images, using the optimal code :
  - with GSL : linear_convolution_gsl.cc
  - with FFTW : linear_convolution_fftw.cc
- Computing a 2D circular convolution with images, using the optimal code :
  - with GSL : circular_convolution_gsl.cc
  - with FFTW : circular_convolution_fftw.cc

4 References

- Discrete Fourier transform : definition and properties link
- Convolution product : definition and properties link
- Convolution theorem : link
- Fast Fourier transform with the GSL : link
- Fast Fourier transform with FFTW3 : link
- Book : Digital Signal Processing, J. Proakis, D. Manolakis. Chap 8 deals with computing 2 DFT at once : link