TME n° 4

Ryan Lahfa

22 décembre 2019

Table des matières

Table des figures

Exercice 5

Q1. On applique le cours en prenant garde si $\omega = 1$ ou si $n = 1$ (dimension du vecteur).

```
function res = fft_rec(A, w)
    n = length(A);
    res = A;
    if w ~= 1 && n > 1
        B = fft_rec(A(1:2:n), w^2);
        C = fft_rec(A(2:2:n), w^2);
```

```
        res = zeros(n, 1);
        for j=1:n/2
            res(j) = B(j) + w^(j - 1) * C(j);
            res(j + n/2) = B(j) - w^(j - 1) * C(j);
        end
    end
end
```

Q2. On fait attention à respecter le circuit itératif en renversant l'ordre des bits (la difficulté technique):

```
function res = fft_iter(A)
    n = length(A);
    q = log2(n);
    res = A;

    if n > 1
        res = zeros(n, 1);
        [~, indexes] = bitrevorder(0:(n-1));
        for j=1:n
            res(indexes(j)) = A(j);
        end
        bsize = 1;

        for i=1:q
            bsize = bsize * 2;
            w = exp(1i*2*pi/bsize);

            for j=1:bsize:n
                k = j + bsize/2;
                W = 1;

                for p=0:(bsize/2 - 1)
                    c_even = res(j + p);
                    c_odd = res(k + p);

                    res(j + p) = c_even + W*c_odd;
                    res(k + p) = c_even - W*c_odd;
                end
            end
        end
    end
end
```

Q3/4. On traite les deux questions simultanément en faisant des tests sur $\{2^k \mid k \in [[1, 30]]\}$, le résultat est la figure 1.

On y observe donc que la FFT récursive est nettement plus lente que les autres, c'est attendu, MATLAB n'est pas un langage qui effectue des optimisations de récursion terminale (d'après https://stackoverflow.com/questions/5326749/does-matlab-perform-tail-call-optimization), on conclut facilement que tout algorithme récursif sera toujours plus lent que sa version itérative.

Ensuite, la version itérative est nettement plus lente que la version MATLAB, et cela est à nouveau normal, il est fort probable que la version de MATLAB est écrite en C alors que les fonctions MATLAB quant à eux sont interprétés par le runtime MATLAB.

Le résultat est donc conforme à nos attentes.

Exercice 6

Q1. On décide d'employer `fft2` et `ifft2` puisqu'à l'exercice 5, on a déjà écrit une fonction de FFT et on a comparé les performances, pour le coup, il vaut mieux employer ceux de MATLAB.

L'algorithme de compression prend cette forme:

```
function res = compresser_image(X, eps)
    Y = fft2(X);
    for i=1:size(Y,1)
        for j=1:size(Y,2)
            if norm(Y(i,j)) <= eps
                Y(i,j) = 0;
            end
        end
    end
    % taux_compression calcule le rapport d'éléments non nuls sur le nombre d'éléments tota
    % fprintf("Taux de compression: %f %% (%d)\n", taux_compression(X, Y)*100, nnz(Y));
    res = Y;
end
```

Où `eps` est un paramètre de compression, plus `eps` est grand, plus l'image est dégradée mais devient légère.

Q2. On constate trois valeurs `eps` intéressantes: $\varepsilon_k = 5 \times 10^k, k \in \{1, 2, 3\}$.

Voici les images et le taux de parcimonie (avec un tracé de la structure de l'image) pour ces valeurs avec l'algorithme de compression aux figures 2, 3, 4, 5, 6.

Si on interprète rapidement, on constate peu voire aucune différence de la figure 2 à la figure 3.

Cependant, à la figure 4, on voit une forme de bruit apparaître dans l'image, elle paraît nettement plus bruitée, on est à 77 % de taux de compression, mais l'image reste tout à fait encore visible et assez proche de son originale.

À la figure 5, l'image devient floutée de façon globale mais reste reconnaissable, nous sommes alors à 95 % de taux de compression, ce qui est impressionnant.

À la figure 6, à 99 % taux de compression, l'image n'est plus du tout reconnaissable, on ne voit que sa silhouette vague en regardant attentivement sachant qu'on sait ce que nous sommes censés voir [1].

On conclut que dans le domaine de Fourier, les 3098 éléments non nuls dominent la représentation et la structure de l'image. Ce n'est pas surprenant, la théorie de l'acquisition comprimée démontre que ce genre de représentation sont possibles sur beaucoup plus de cas que l'on croit.

Enfin, l'on peut quand même se demander ce que ferait la FFT sur des images muni d'une couleur cette fois-ci, comment varie le taux de compression en fonction de l'espace de couleurs choisis (RGB, HSV, HSL par exemple).

Exercice 7

Q1. On réutilisera `fft` directement à nouveau en faisant du produit terme à terme, tout en veillant à construire les bonnes tailles de vecteurs à l'avance [2]

```
function res = fft_prod(P, Q)
    N = 2*max(length(P), length(Q));
    P_ = fft([P zeros(1, N - length(P))]);
    Q_ = fft([Q zeros(1, N - length(Q))]);

    R_ = P_ .* Q_;

    res = ifft(R_);
end
```

Q2. On compare `conv` de MATLAB, l'algorithme naïve quadratique et notre produit par FFT, voici les résultats à la figure 7.

Sans surprise, l'algorithme naïf suit une croissance en $O(n^2)$, alors que la produit par FFT et la convolution de MATLAB croissent très lentement.

Si on zoomait sur la convolution et le produit par FFT, on observait que lorsque le degré est élevé, la FFT devient beaucoup plus lente et sa vitesse varie énormément d'un degré à un autre. Tandis que la convolution est très stable.

Une explication qu'on peut avancer c'est encore une fois le caractère interprété de l'algorithme combiné au fait qu'on est pas sûr qu'on fait un bon usage des caches CPU dans notre algorithme alors qu'il est très fort probable que la convolution de MATLAB doit être optimisé à fond de ce côté là.

1. Ce qui est de la triche.
2. Il est possible de faire plus efficace sur les tailles et en préallouant, mais ce n'est pas très important pour la comparaison que nous allons effectuer.

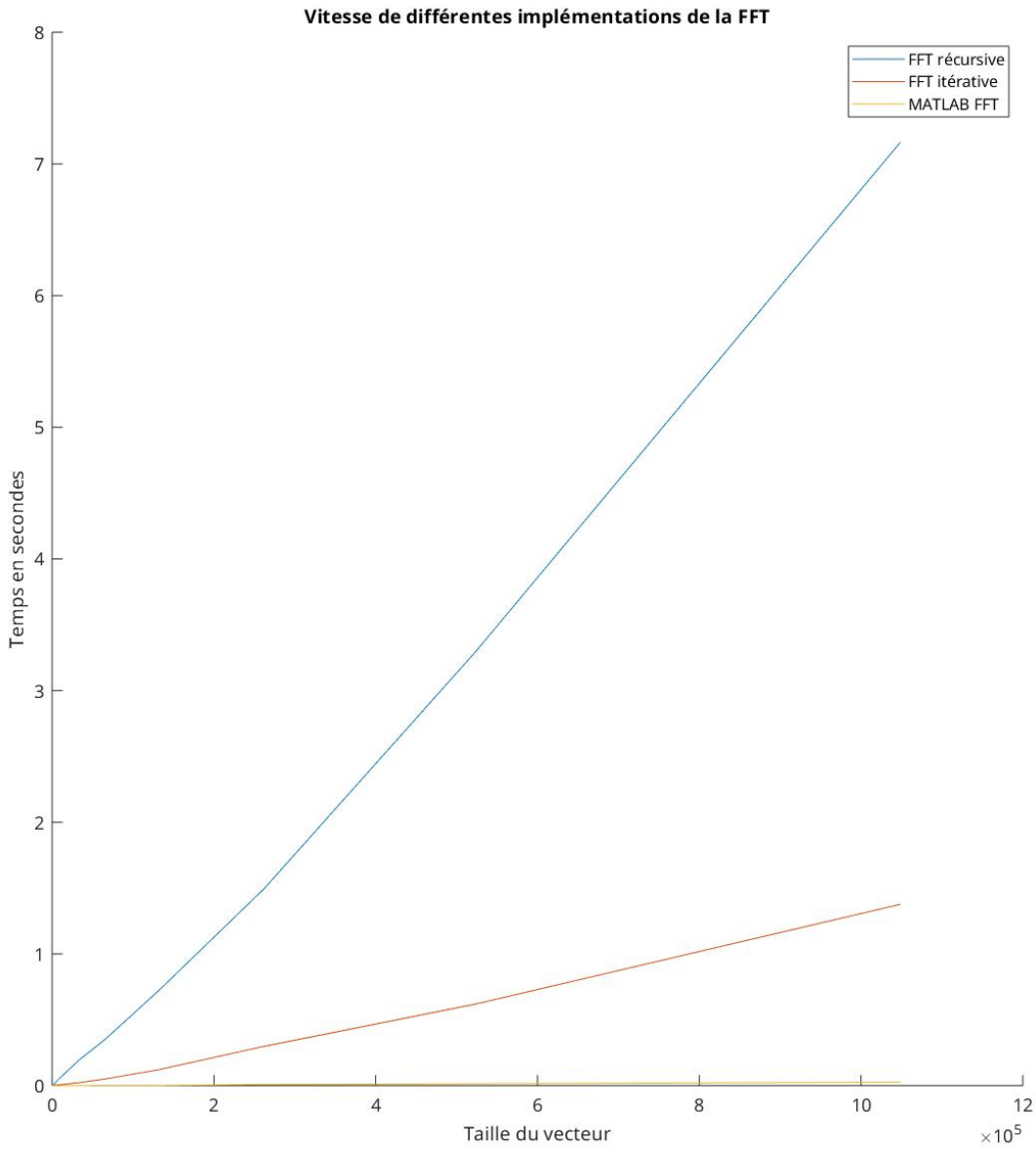
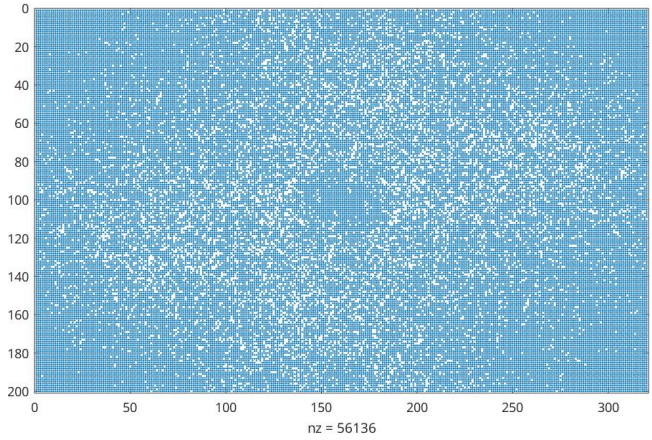


FIGURE 1 – Efficacité des implémentations de la FFT



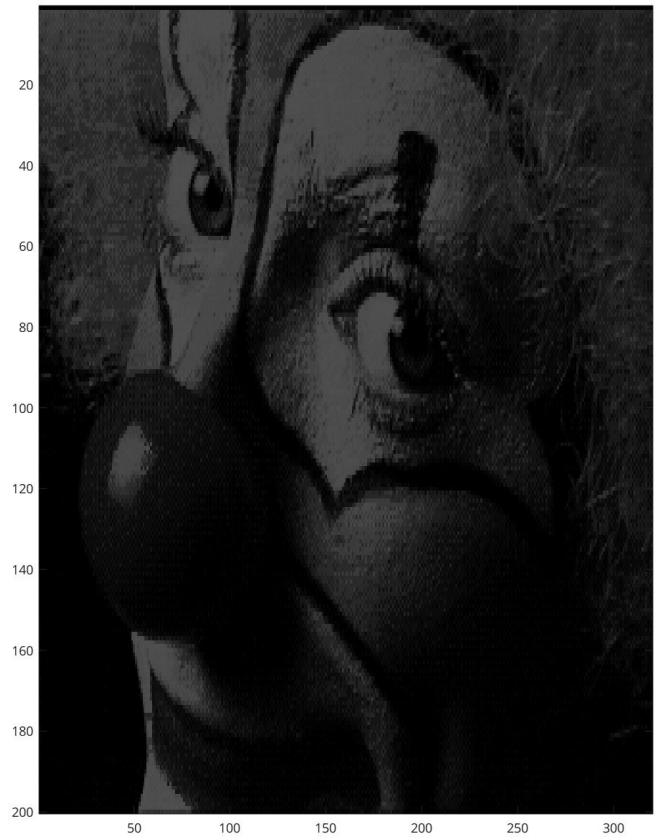
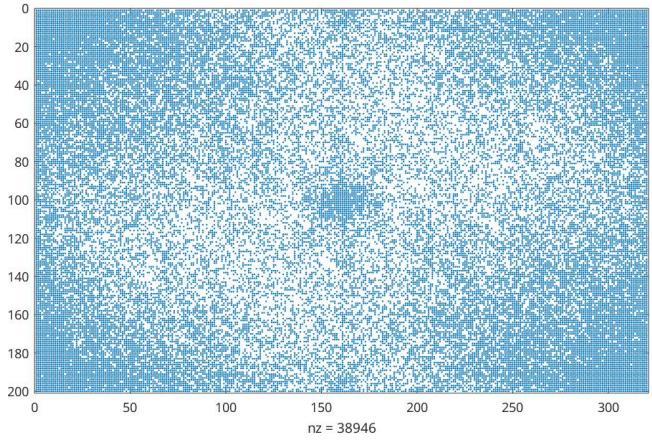


FIGURE 2 – Compression avec ε_1, taux de compression: 12 % (56136 éléments non nuls)



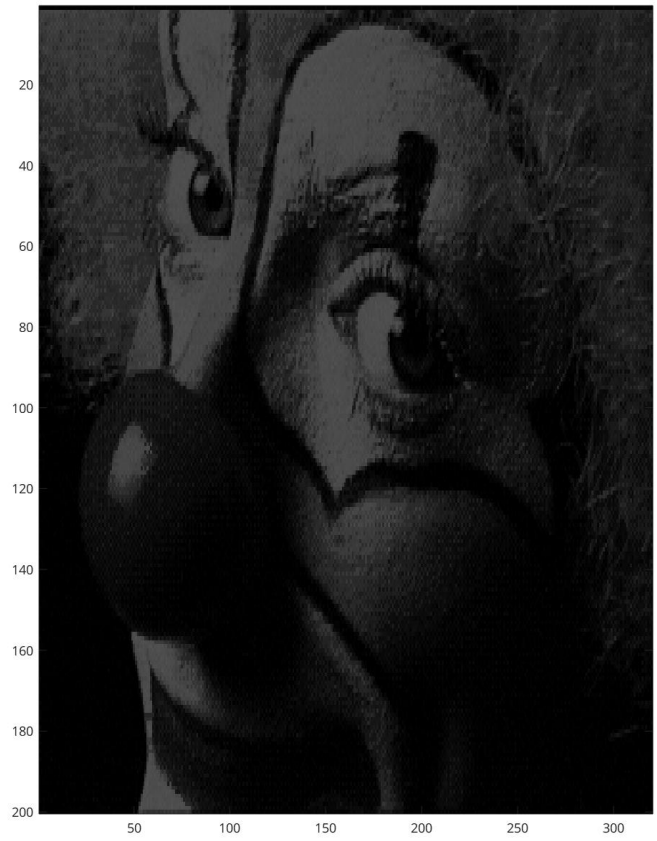
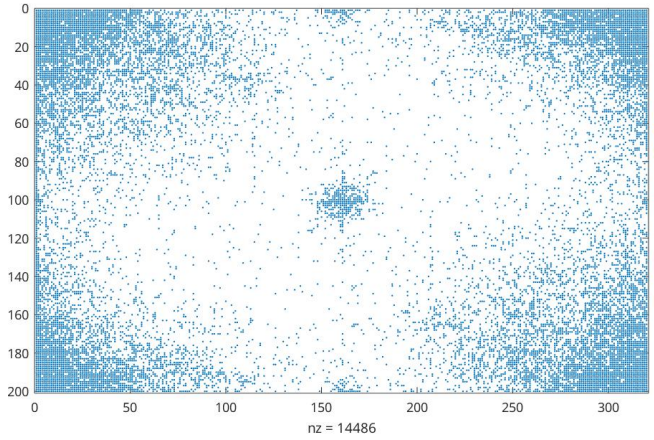


FIGURE 3 – Compression avec $\varepsilon = 10^2$, taux de compression: 39 % (38946 éléments non nuls)



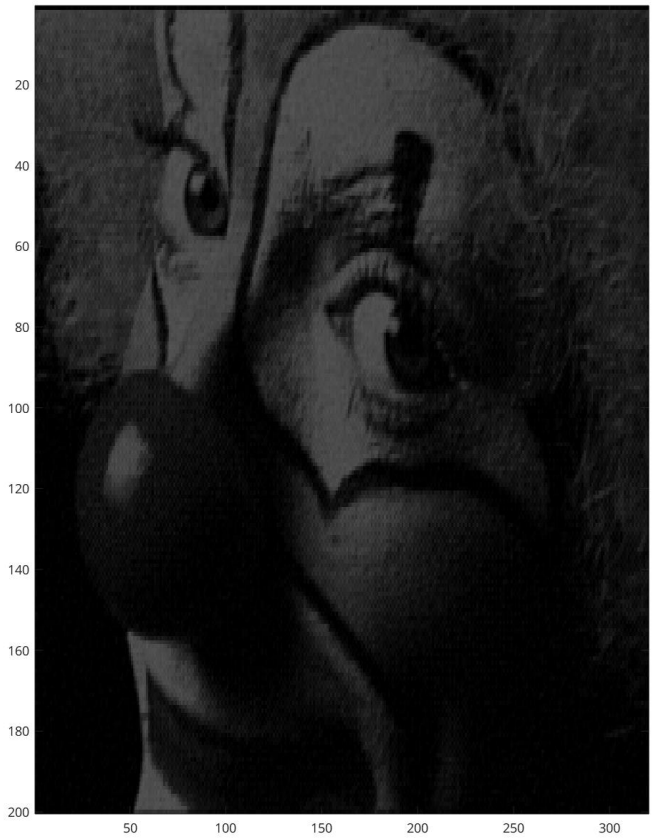


FIGURE 4 – Compression avec $\varepsilon = 2 \times 10^2$, taux de compression: 77 % (14486 éléments non nuls)

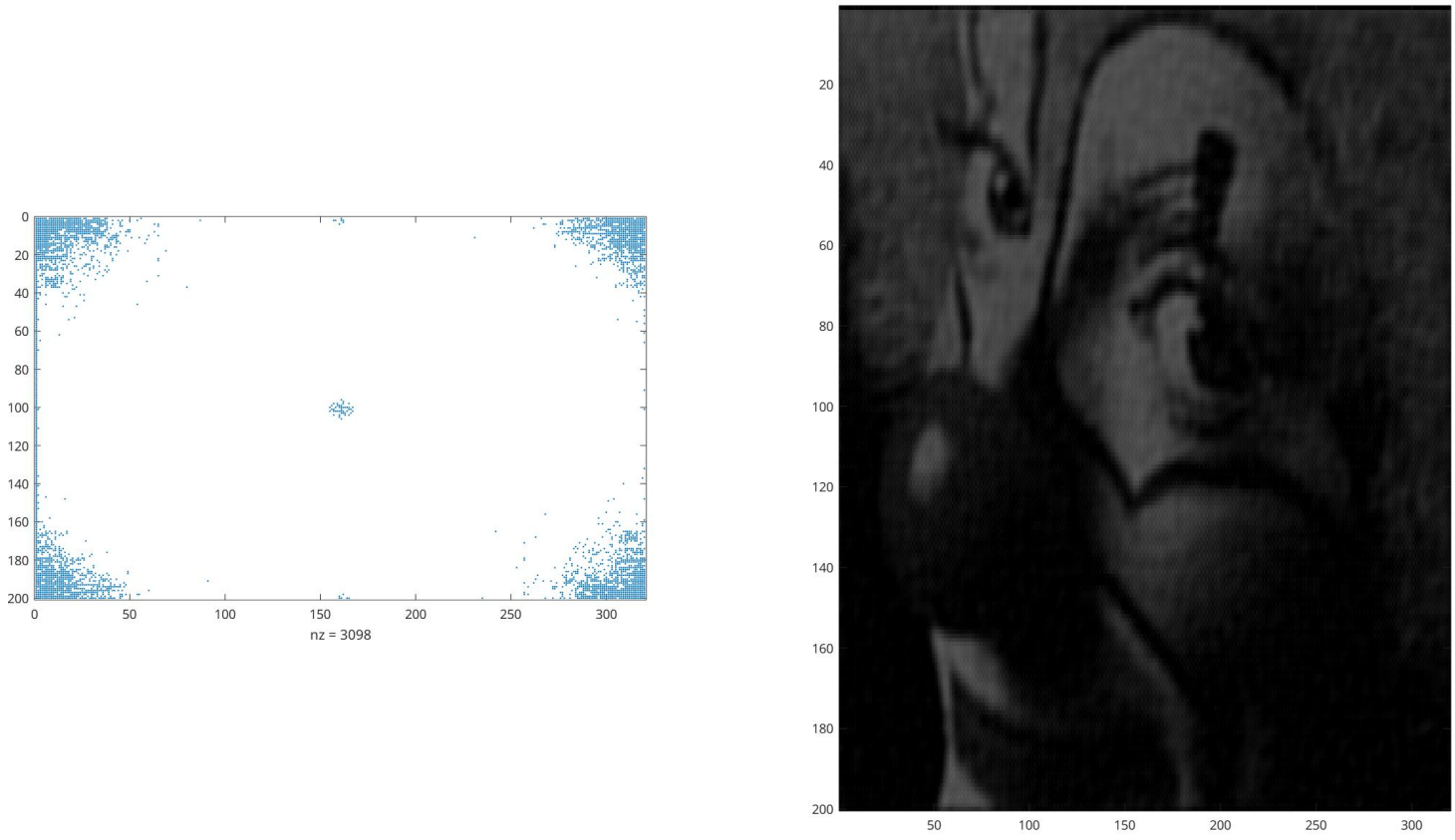


FIGURE 5 – Compression avec ε_2, taux de compression: 95 % (3098 éléments non nuls)

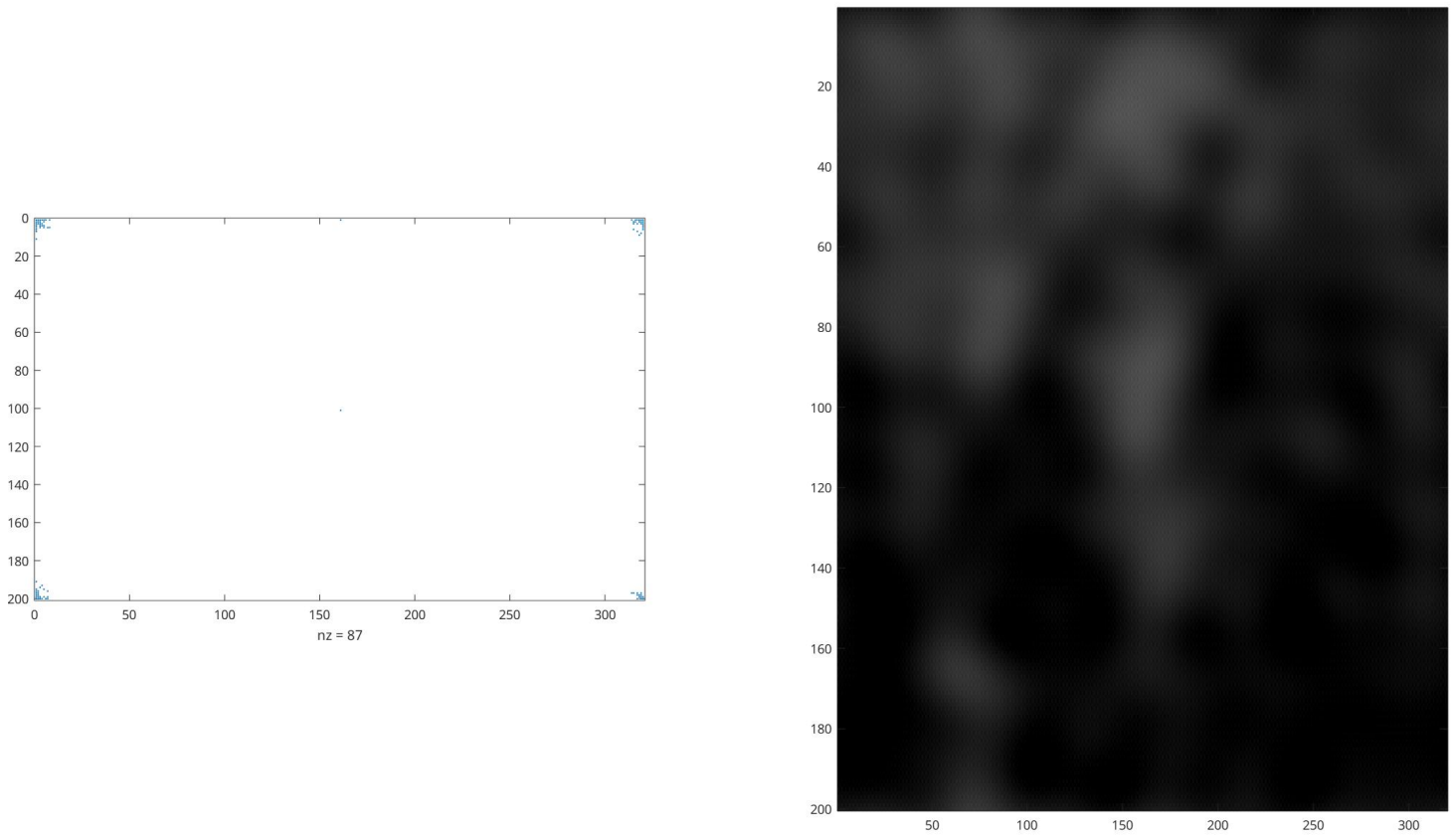


FIGURE 6 – Compression avec ε_3, taux de compression: 99 % (87 éléments non nuls)

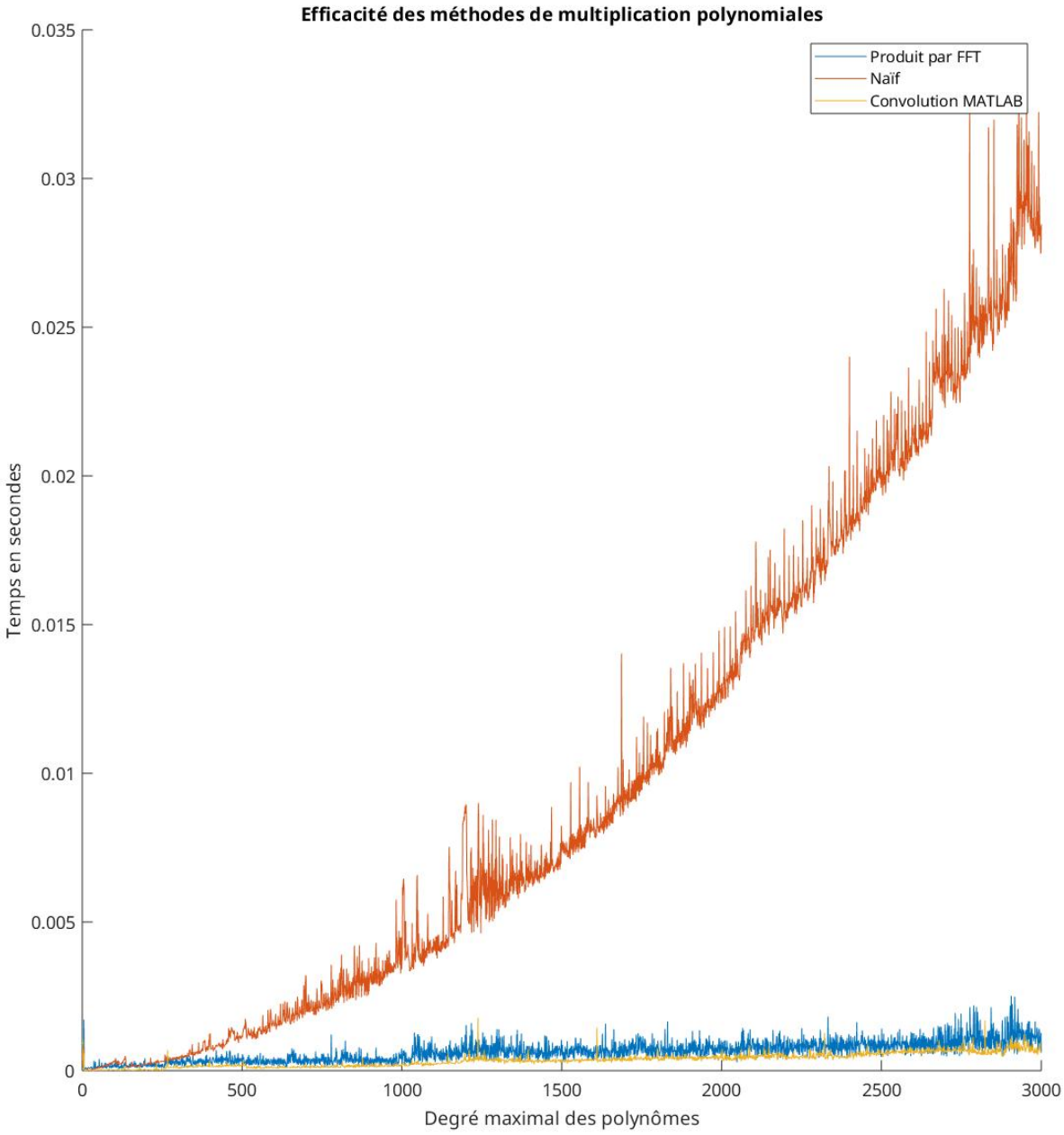


FIGURE 7 – Efficacité des méthodes de multiplication polynomiales