INTRODUCTION AUX PROBLEMES INVERSES EN TRAITEMENT D'IMAGE

DEDRUITAGE

Guénon Marie et Favreau Jean-Dominique

VIM / MASTER SSTIM

Table des matières

Méthode

Pour chaque pixel de l'image $i_{(x,y)}$, on prend un patch p_0 autour de lui (en excluant la bande de bord pour éviter d'avoir à gérer les problèmes).
Pour une grande zone autour du pixel (fenêtre de recherche), on extrait tous les patches.

On calcule la distance entre p_0 et tous les autres patches et on garde les *k* patches les plus proches puis on fait la moyenne pondérée et on normalise et on obtient le patch :

$$p_{new} = \frac{\sum \omega_i * p_i}{\sum \omega_i}$$

Où $p_i$est le patch i et $\omega_i = e^{-\frac{|p_i - p_0|^2}{\alpha n^2}}$

Et finalement, on assigne au pixel $i_{(x,y)}$ la valeur au centre du p_{new} correspondant.

```
%return the new value of the pixel (x,y).
function [ val] = similar_patches(img, n, w,x,y,k,alpha)
    n_moins_un_sur_deux = (n-1)/2;%to compute this term only once.

    %this is the reference patch
    pref = img((y-n_moins_un_sur_deux):(y+n_moins_un_sur_deux),( x-n_moins_un_sur_deux):(x+n_moins_un_sur_deux));

   %we check the border condition for the searching windows
    row_begin = max(y-(w-1)/2,1+n_moins_un_sur_deux);
    row_end = min(y+(w-1)/2,size(img,1)-n_moins_un_sur_deux);

    n_rows = row_end - row_begin+1;

    col_begin = max(x-(w-1)/2,1+n_moins_un_sur_deux);
    col_end =  min(x+(w-1)/2,size(img,2)-n_moins_un_sur_deux);


    %for each pixel of the searching window we extract a patch n by n and we compute the distance with the reference patch
    %and we save it with the location (col,row) of this pixel.
    all_norm = zeros((col_end-col_begin+1)*n_rows,3);
    for col = col_begin : col_end
        ccols = col - col_begin;
        ccols_x_n_row = ccols*n_rows;
        for row = row_begin : row_end
            rrows = row - row_begin;
            all_norm(ccols_x_n_row+rrows+1,:)=[sum(sum((pref-img((row-n_moins_un_sur_deux):(row+n_moins_un_sur_deux),( col-n_moins_un_sur_deux):(col+n_moins_un_sur_deux))).^2)),row,col];
        end
    end

    %we sort the distances and for the k lower distances we compute the weight we will give to the corresponding patch
    [sorted_value, indice] = sort(all_norm(:,1));
    row = all_norm(indice,2);
    col = all_norm(indice,3);
    tmp2 = zeros(n,n);
    un_sur_alpha_nn = 1/(alpha * n * n);
    weight = exp(-sorted_value(1:k)*un_sur_alpha_nn);
    %we have to normalize the weight
    total_weight_inv = 1/sum(weight);
    weight=weight*total_weight_inv;
    %it would be better if we compute the weighted-sum only for the central pixel
    for i = 1 : k
        tmp2 = tmp2 + patch_extract(img,n,col(i),row(i))*weight(i);
    end
    val = tmp2(n_moins_un_sur_deux+1,n_moins_un_sur_deux+1);
end
```

Paramètres

Pour pouvoir appliquer notre algorithme, nous avons dû fixer un certain nombre des paramètres du modèle :

- Taille du patch : n

A partir de cette taille, nous avons créé un patch de taille nxn.

Nous avons choisi de fixer la taille du patch à $n = 11$.

- Taille de la fenêtre de recherche : w

Nous avons fixé la taille de la fenêtre de recherche à $w = 5 * n = 55$ et on obtient alors une fenêtre de recherche de taille wxw.

- Paramètre de poids : α

Nous avons testé plusieurs valeurs de α et obtenus les résultats suivants : (avec $k = 20$)

Image initial	Image bruitée	Image pour $\alpha = 10$

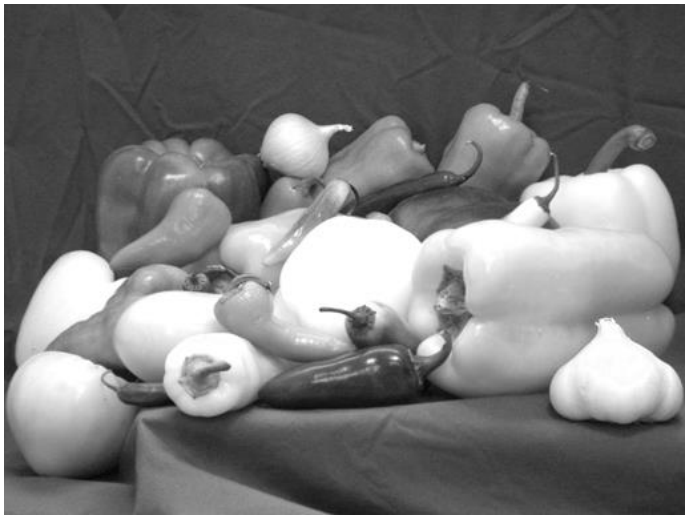
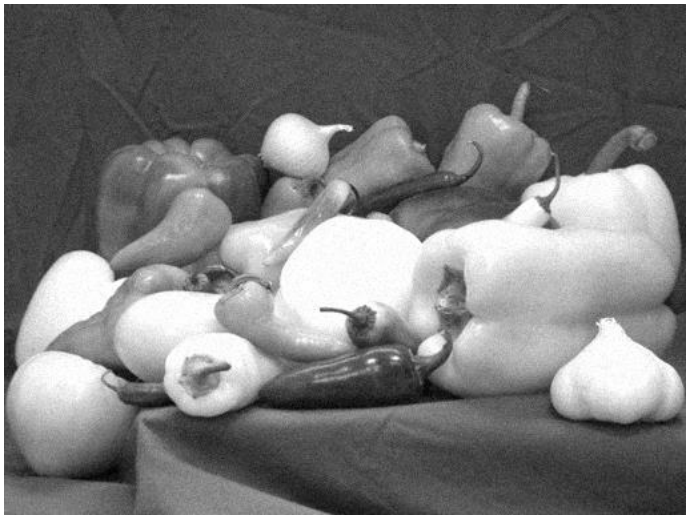
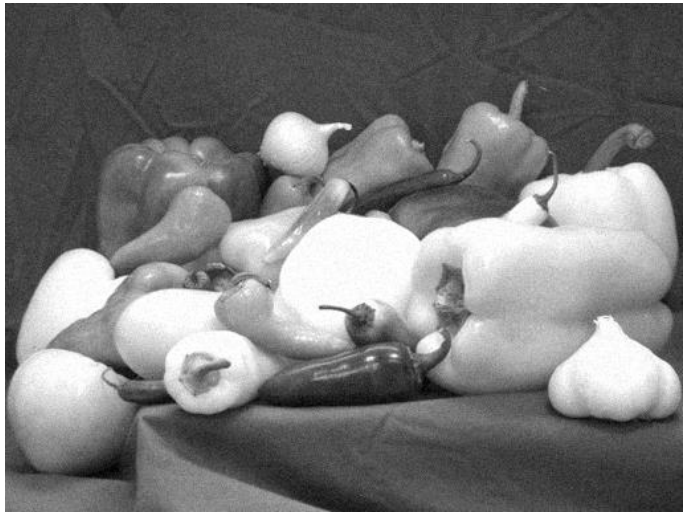
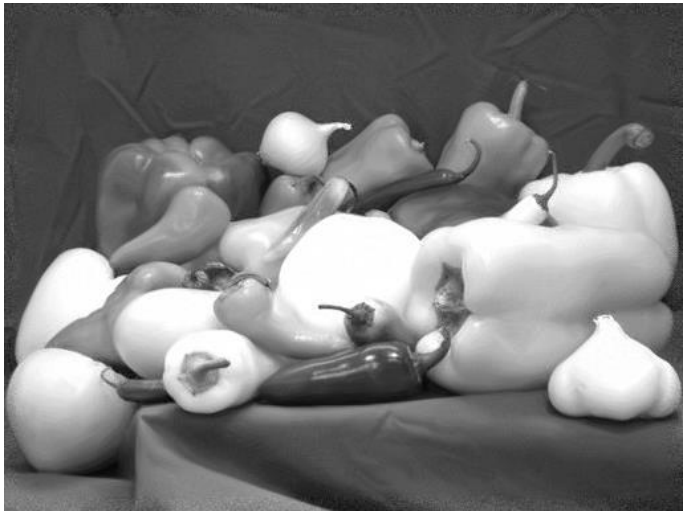

		

Image pour $\alpha = 50$	Image pour $\alpha = 100$	
		

Nous pouvons voir que lorsqu'on augmente α, nous améliorons l'image. En effet, lorsque α est petit nous donnons plus d'importance à la valeur initiale du pixel et donc nous le faisons moins varier ce qui fait que nous gardons une certaine part du bruit dans le résultat. Alors que lorsque nous augmentons α, on accorde moins d'importance aux patches qui sont éloigné de la valeur du pixel et donc on uniformise le résultat ce qui nous donne une image moins bruitée.

- Nombre de patches les plus similaires : k

Nous avons testé plusieurs valeurs de k et obtenus les résultats suivants : (avec $\alpha = 100$)

Image initial	Image bruitée	Image pour $k = 5$

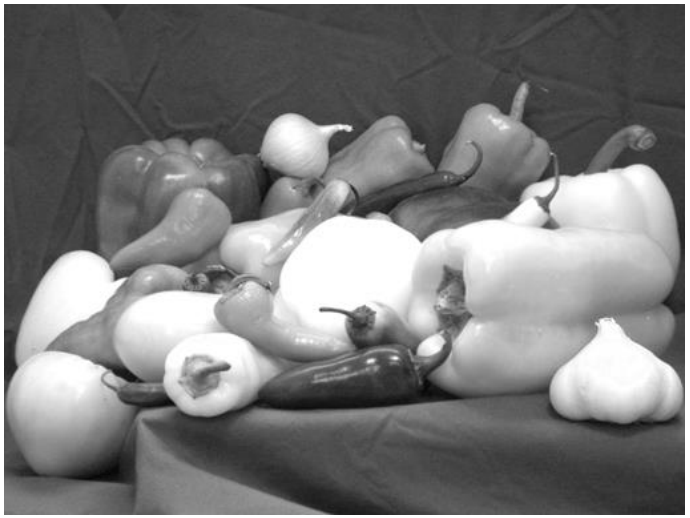
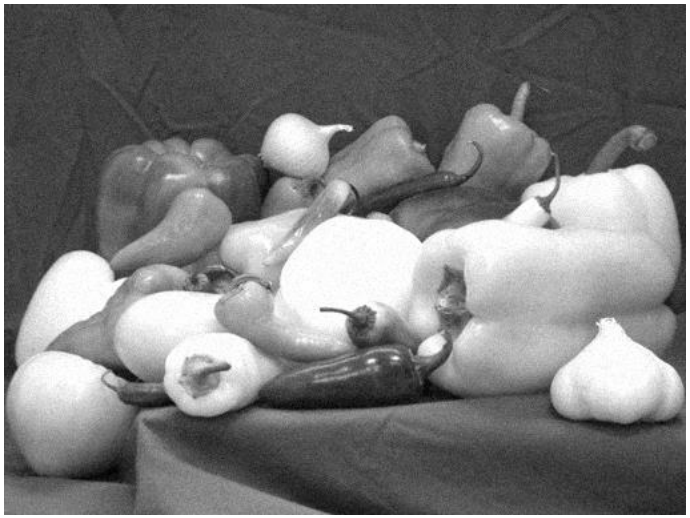
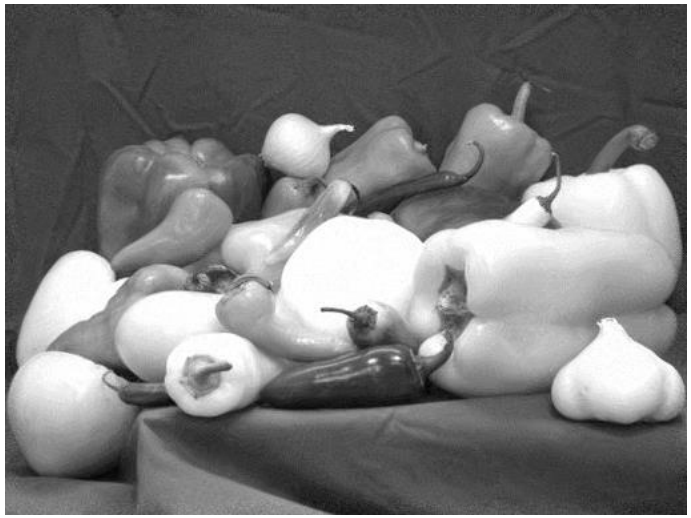
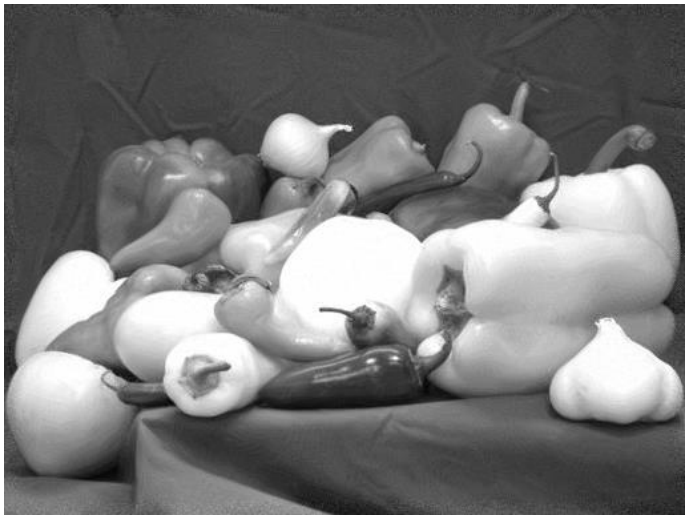
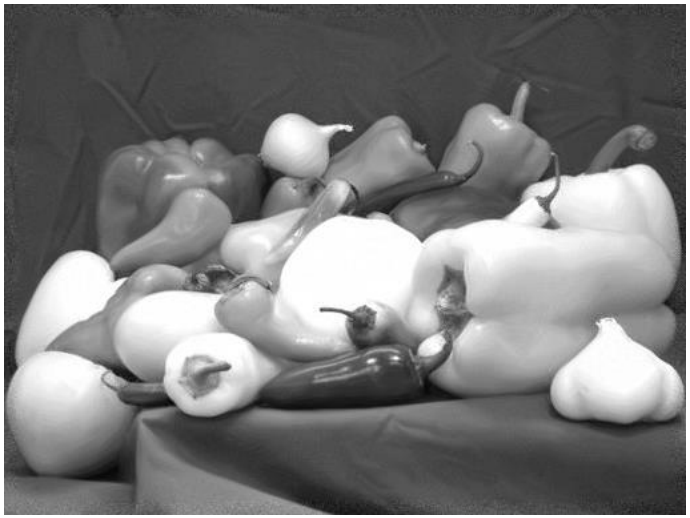
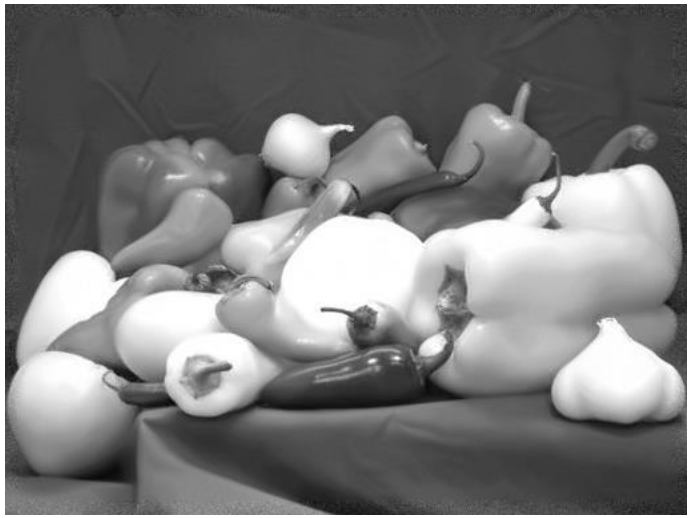

		

Image pour $k = 10$	Image pour $k = 20$	Image pour $k = 80$
		

Avec un *k* petit, nous pouvons voir que même si l'image est améliorée, il reste une partie du bruit sur l'image, comme c'est le cas ici pour $k = 5$.
Plus le *k* grandit, plus nous réduisons le bruit dans l'image obtenue en sortie et ce même jusqu'à obtenir en résultat une image moins bruitée que l'image initiale avant application du bruit gaussien pour $k = 80$.
Cependant, ce débruitage ne va pas sans désavantages. En effet, pour ce dernier *k*, nous pouvons constater une certaine perte d'information qui se traduit par une introduction de flou dans certaines parties de l'image en particulier aux endroits où les contrastes sont peu élevés.

Commentaires

Tout au long de ce projet, nous avons pu constater que la durée des calculs à effectuer pour obtenir les résultats et les images débruitées étaient très longs. Ceci est dû au nombre de calculs à faire mais aussi au fait que Matlab ne parallélise pas du tout les opérations alors qu'ici de nombreux calculs sont indépendants et pourraient être fait en OpenCL pour être lancés sur le CPU et le GPU pour gagner énormément de temps de calcul.

Par ailleurs, sur les images obtenues ci-avant nous avons pu constater l'existante du bande en bordure de l'image qui n'était pas débruitée. Ceci est dû au fait que nous n'appliquons pas notre algorithme sur une bande de taille n au bord de notre image initiale. Cette lacune pourrait être l'objet d'un prochain projet.