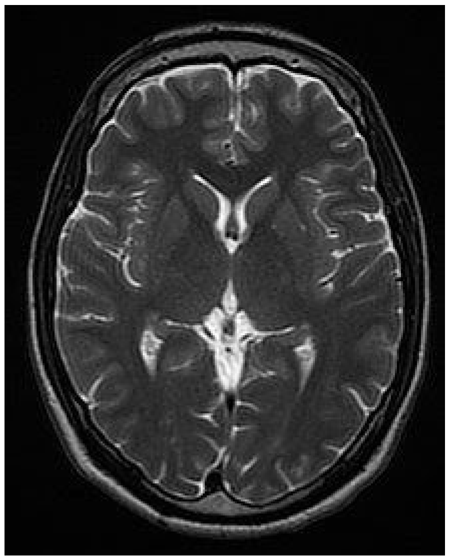
Name: Shrinidhi Anil Varna
Roll no: 171CO145
Course: Digital Image Processing
--
Q: Implement at least three different versions of the NLM method. Compare the methods of simulated noisy images. Create noisy images by adding Gaussian noise of standard deviations 5, 10, 15, 20 and 25. Use PSNR as a similarity metric. Assignments should contain a brief description of each method, Experimental results and analysis and Conclusion.

Introduction:

Three different versions of NLM are implemented, and the image used for it is that of a brain.
The image is grayscale.
The image shown below is taken as ground truth to perform experiments in our study.



This image will be used in studying how different versions of NLM denoise a noisy image of the above image. The code used as part of performing these experiments is provided in the end of this study.

Gaussian noise of standard deviation 5, 10, 15, 20, and 25 is added to the original image, and each one of them will be denoised using all the three versions of NLM. Peak signal-to-noise ratio (PSNR) will be used as a similarity metric for the denoised images. The next section will give a brief description of the three versions implemented.

NLM and the versions implemented:

Non-local means is an algorithm in image processing for image denoising. Unlike "local mean" filters, which take the mean value of a group of pixels surrounding a target pixel to smooth the image, non-local means filtering takes a mean of all pixels in the image, weighted by how similar these pixels are to the target pixel. This results in much greater post-filtering clarity, and less loss of detail in the image compared with local mean algorithms.

Version 1: Implementation having a quadratic time complexity
In this approach, the implementation makes use of Euclidean function to measure the patch similarity. This compares entire patches (not individual pixel intensity values) to compute weights for denoising pixel intensities. Comparison of whole patches is more robust, i.e. if two patches are similar in a noisy image, they will be similar in the underlying clean image with very high probability.

$$NL[v](i) = \sum_{j \in I} w(i,j)v(j)$$

where $v = \{v(i) \mid i \in I\}$ and $0 \leq w(i, j) \leq 1$.

The similarity between two pixels i and j depend on the similarity of the intensity grey level vectors v(Ni) and v(Nj), where Nk denotes a square neighbourhood of fixed size and centres at a pixel k. This similarity is measured as a decreasing function of the weighted Euclidean distance,

$\|v(N_i) - v(N_j)\|^2_{2,a}$, where a > 0 is the standard deviation of the Gaussian kernel.

The application of the Euclidean distance to the noisy neighbourhoods raises the following equality.

$$E\,\|v(N_i) - v(N_j)\|^2_{2,a} = \|u(N_i) - u(N_j)\|^2_{2,a} + 2\sigma^2$$

This equality shows the robustness of the algorithm since, in expectation, the Euclidean distance conserves the order of similarity between pixels. The pixels with a similar grey level neighbourhood to $v(N_i)$ have larger weights in the average. These weights are defined as,

$$w(i,j) = \frac{1}{Z(i)} e^{-\frac{\|v(N_i) - v(N_j)\|^2_{2,a}}{h^2}}$$

where Z(i) is the normalising constant

$$Z(i) = \sum_j e^{-\frac{\|v(N_i) - v(N_j)\|^2_{2,a}}{h^2}}$$

and the parameter h acts as a degree of filtering. It controls the decay of the exponential function and therefore the decay of the weights as a function of the Euclidean distances.

Version 2: A small variant of the previous version

This variant consists of restricting the computation of the mean for each pixel to a search window centred on the pixel itself, instead of the whole image. Here the inputs that an algorithm takes are *a noisy image, a patch size, a window search size, the number of neighbours, the standard deviation and h (degree of filtering)*.

Few Nearest Neighbors (NN) can be used to limit the computational burden of the algorithm. This approach is inspired by the idea of the statistically nearest neighbour. A sampling neighbour with the NN approach introduces a bias in the denoised patch.

This approach is a lot faster than the traditional NLM algorithm but compromises a bit on the quality of the image. But hyper-parameter tuning of this approach could outperform the conventional approach, but in my experiment, I haven’t done much of hyper-parameter tuning.

Version 3: A re-implementation of the traditional version with a slight variation to make it computationally faster

It reimplements the NLM filter submitted by *J.V. Manjon-Herrera*, using only MATLAB functions and no loops. A summed-area table is a data structure and algorithm for quickly and efficiently generating the sum of values in a rectangular subset of a grid. There are some approximations made on the way to reduce some computational time. As I studied this version, the main priority in this approach was to bring down the computational time complexity but not compromise on the quality of the image by a great extent. This method can also be tuned to produce better results, but that hasn’t been done in our study because that would give an unfair advantage to this approximation algorithm over the traditional one. The hyper-parameters that are given as input to this algorithm are the same as that used in the previous two approaches. This is because we want to test how two algorithms fare for the same set of hyper-parameters.

Some critical operations that are done in this approach are:

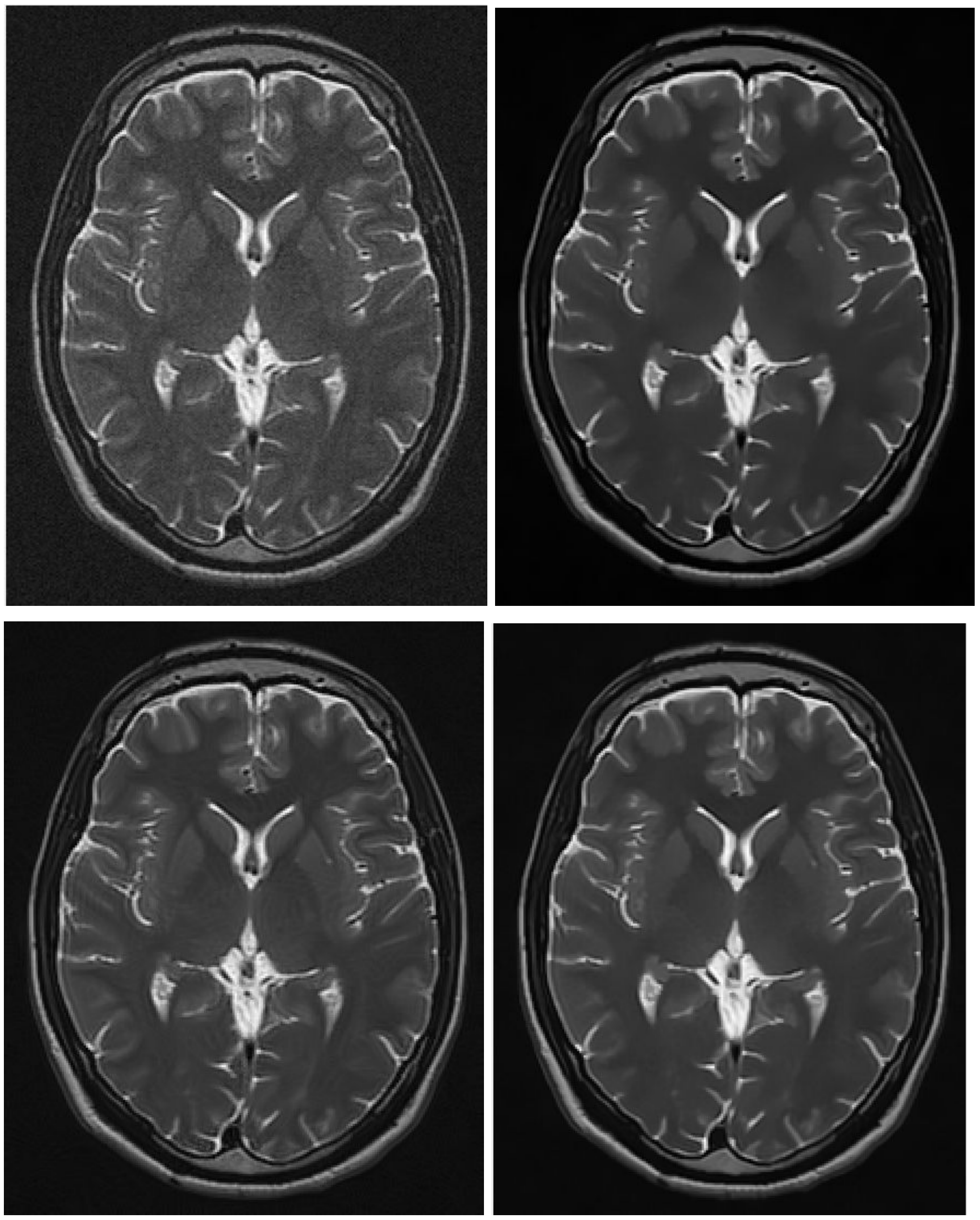
```
1. Compute patches
2. Compute a list of edges (pixel pairs within the same search window)
3. Compute weight matrix (using weighted Euclidean distance)
4. Make matrix symmetric and set diagonal elements.
```

```
5. Normalise weights
6. Compute denoised image
```

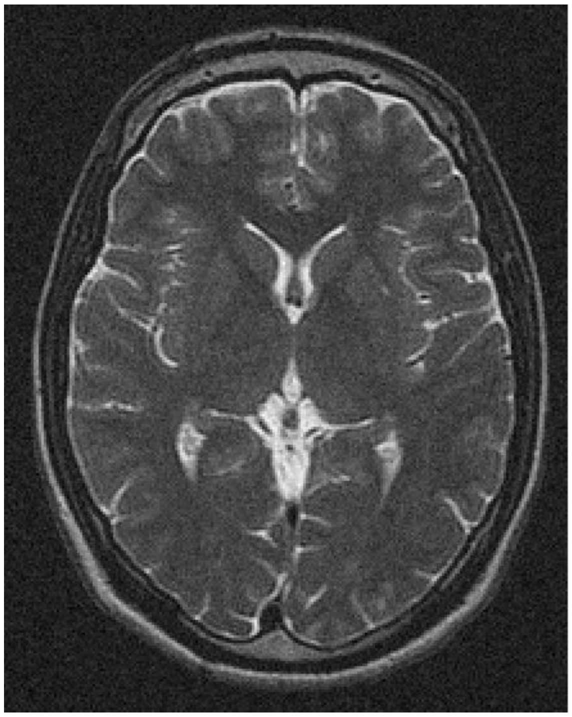
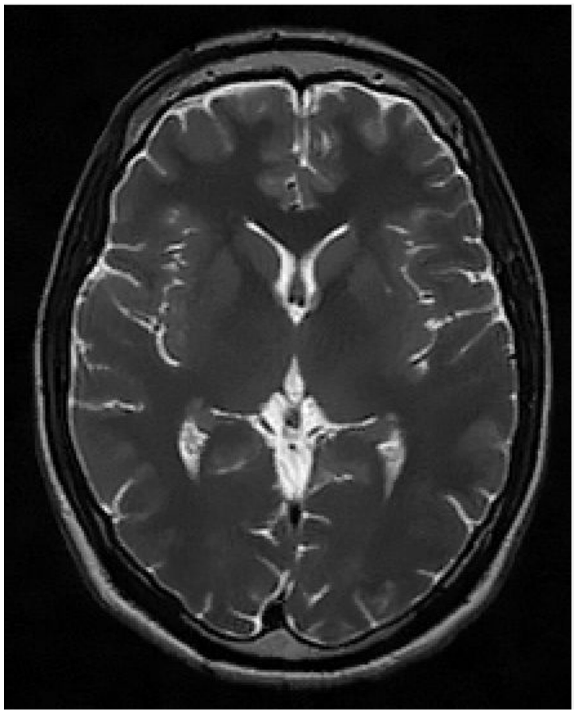
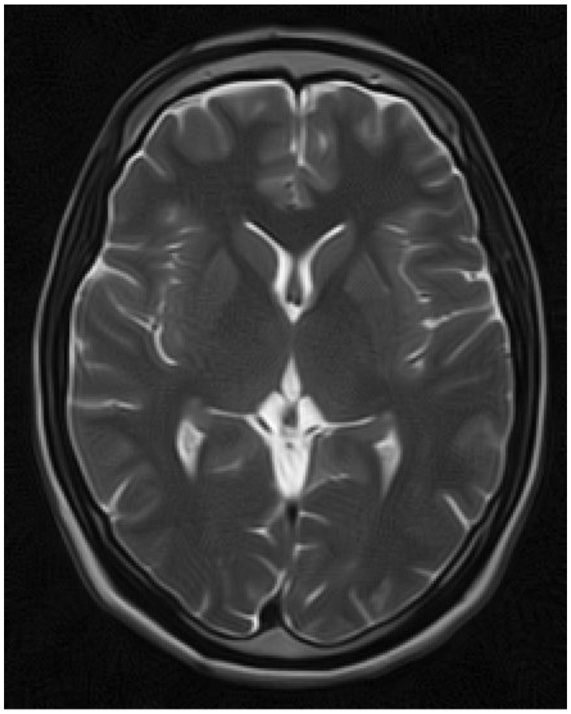
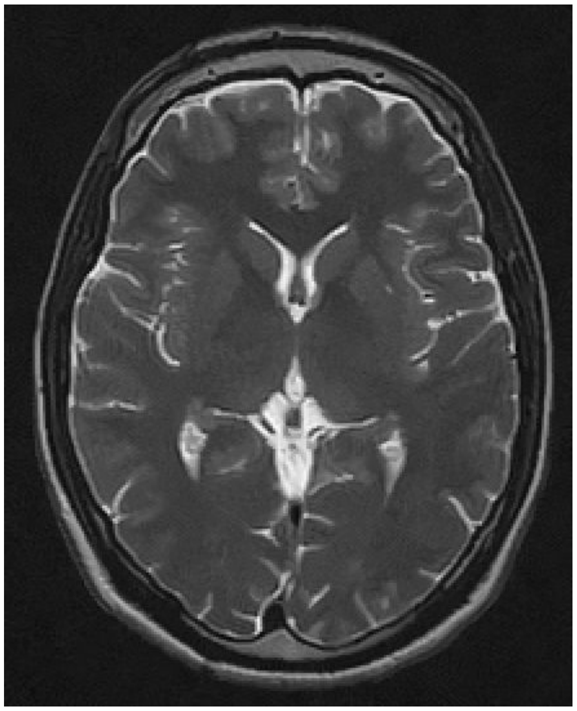
Experimental results:

In this section, the result of the experiments is displayed. PSNR is used as a similarity metric for the three versions. Based on the denoised image obtained from NLM approaches, the mean square error and PSNR are computed. The pattern followed will be as follows: **Top-left**: noisy image; **Top-right:** Output from version-1; **Bottom left:** Output from version-2; **Bottom right:** Output from version-3;

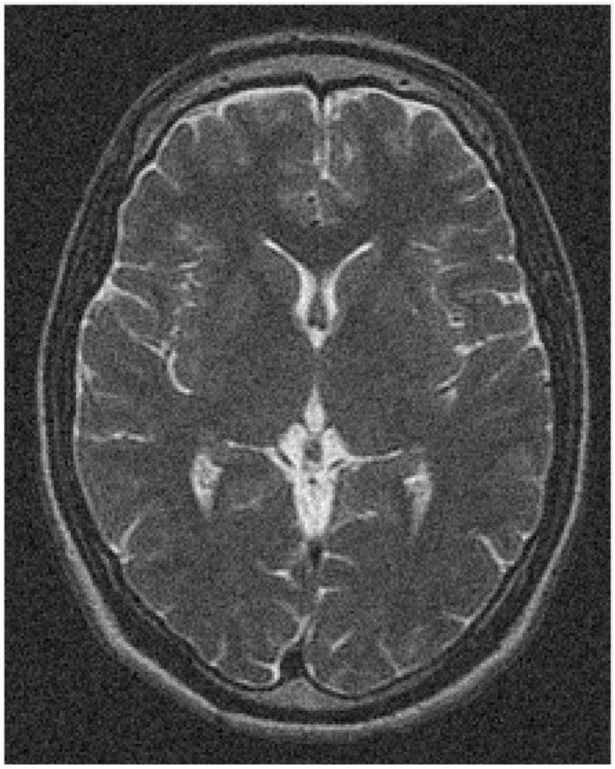
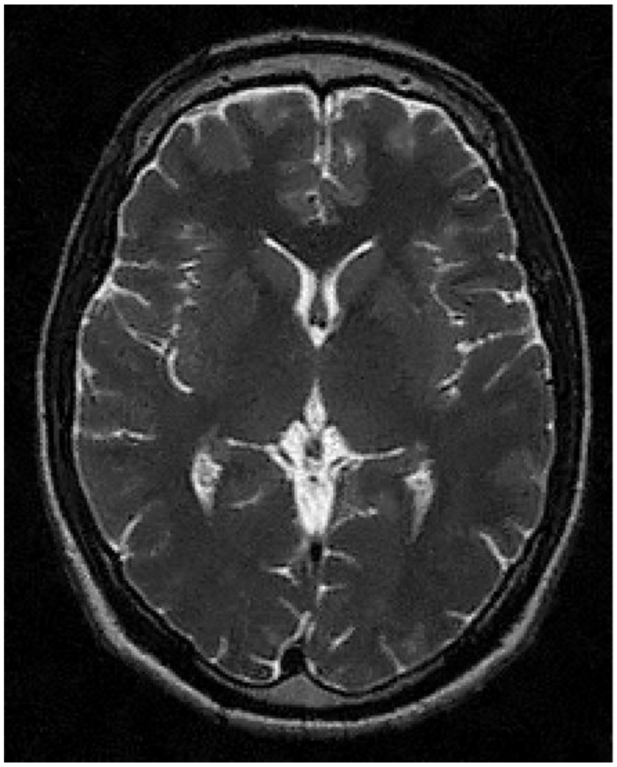
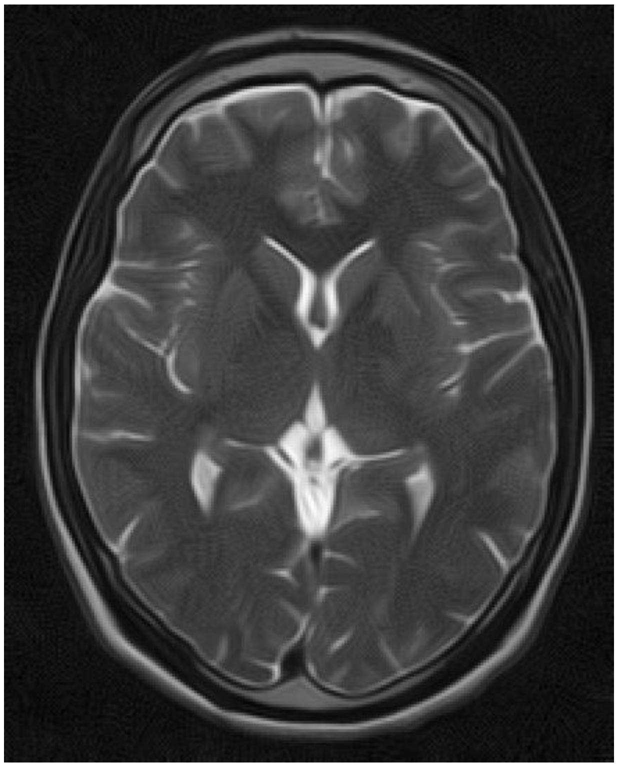
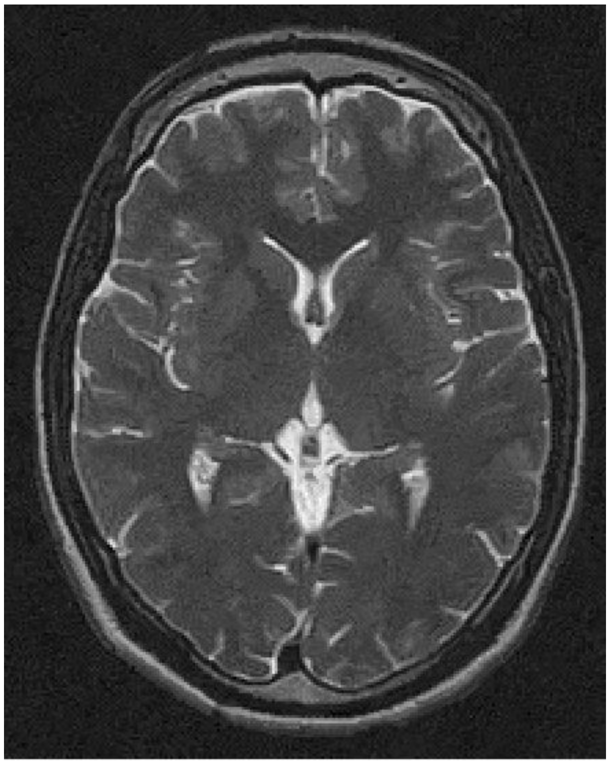
1. Standard deviation = 5



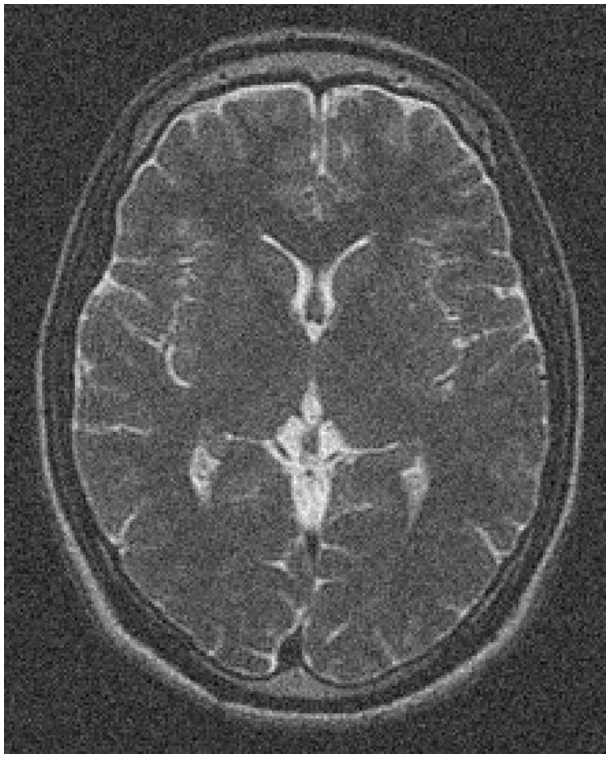
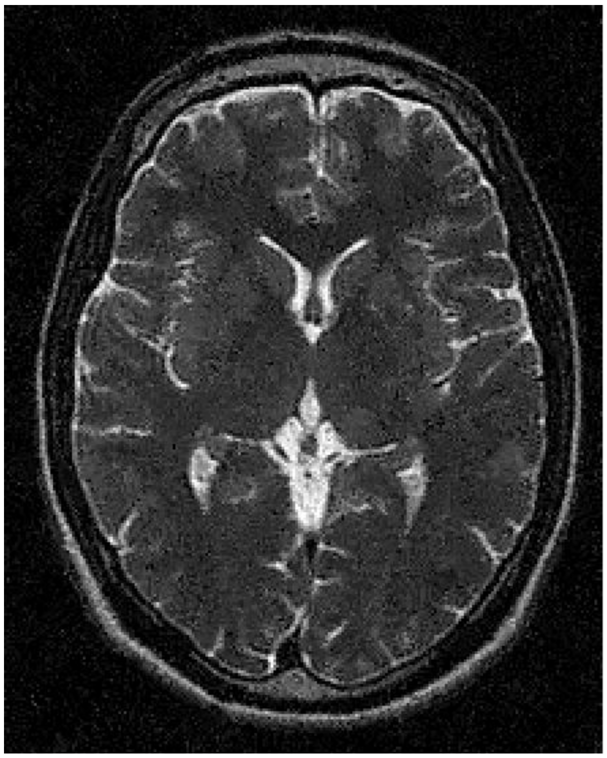
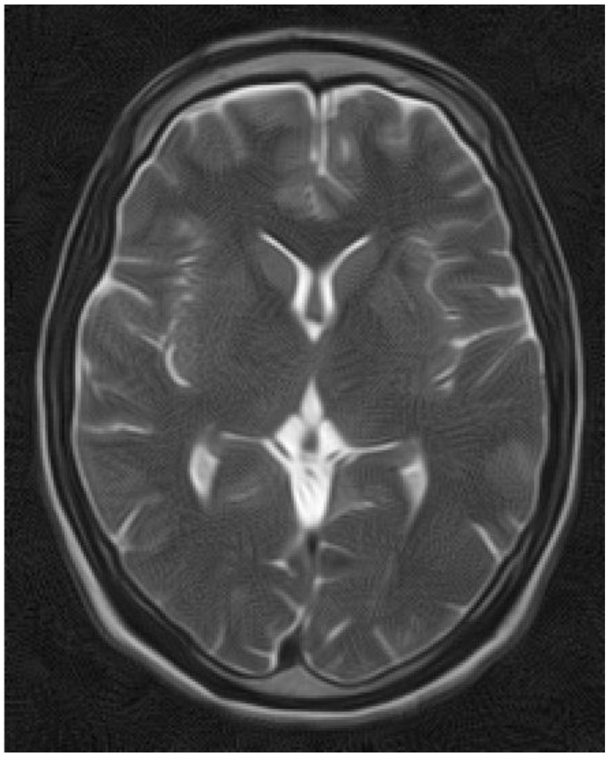
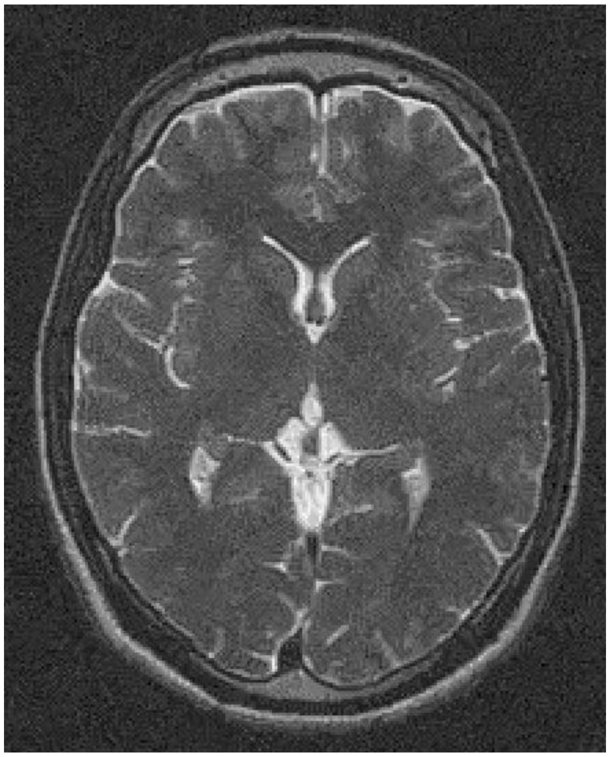
2. Standard deviation = 10

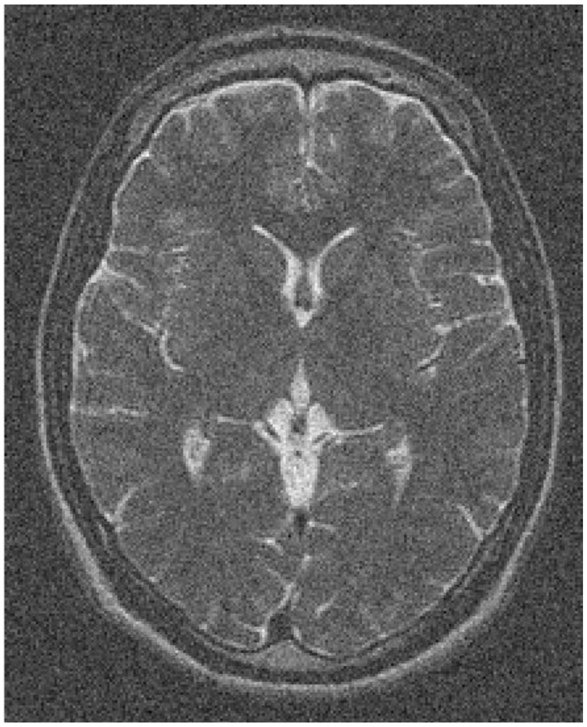
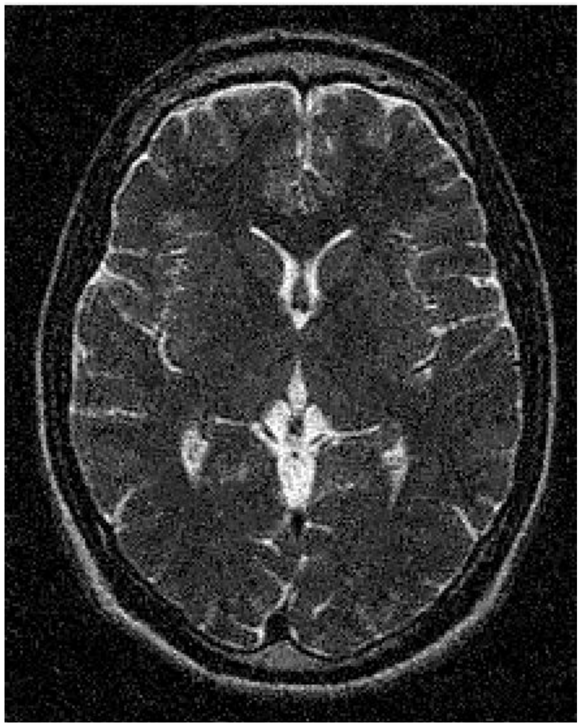
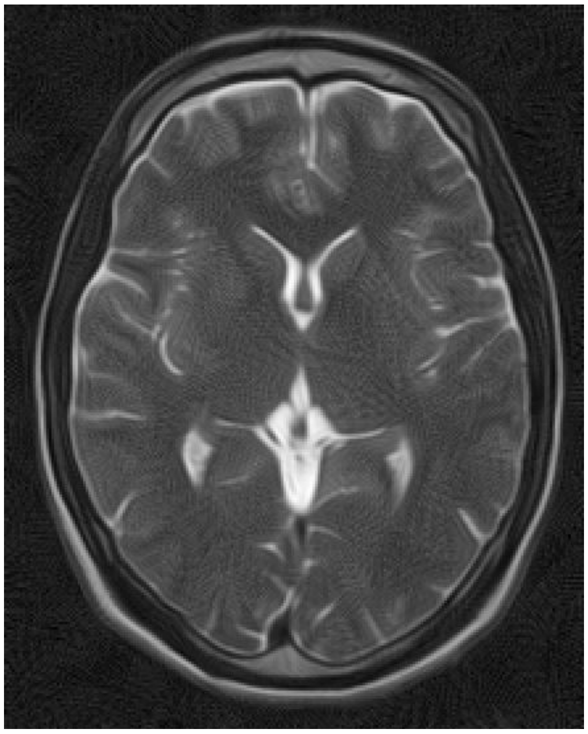
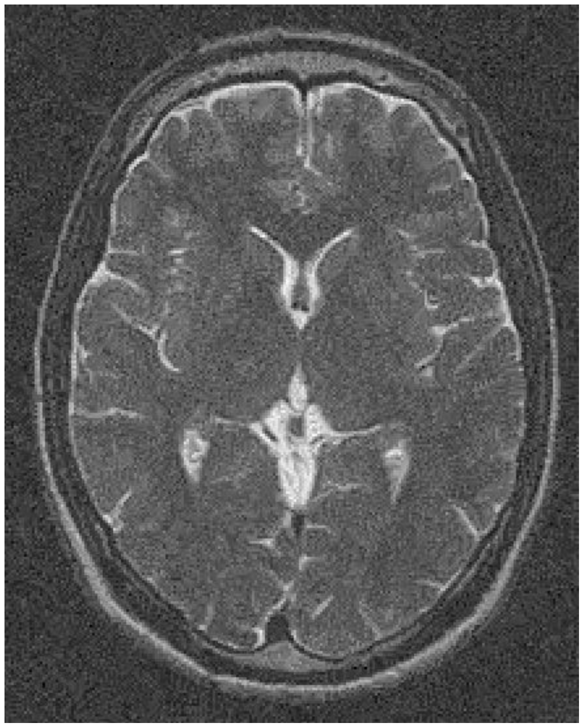
3. Standard deviation = 15

4. Standard deviation = 20

5. Standard deviation = 25

So these were the images obtained from the experiments. The tabulated data of the PSNR values across versions for this experiment are as follows:
PSNR-1,2,3 are the PSNR values of version 1,2,3 of NLM, respectively.

$$PSNR = 20\, x\, log(\frac{MAX_i}{RMS})$$

where RMS is the root mean squared value of the output image and MAX_i is the maximum possible pixel value of the image.

Sr	Standard deviation	PSNR-1	PSNR-2	PSNR-3
1	5	37.3241	37.1344	33.9869
2	10	36.1187	30.2954	32.1808
3	15	33.8603	28.2751	28.5642
4	20	31.9639	27.4801	25.0949
5	25	30.9094	26.6723	22.5099

Analysis:

The traditional method of NLM performs better than two other efficient procedures. As per the given performance metric, the quality of the denoised image is produced best by the first method which is the conventional method and worst by the third method which is more like an approximation for the traditional method.
To improve the quality of the denoised images in the second and the third approaches, I had to adjust the value of h, t and f which stand for the degree of filtering, size of search window and radius of similarity window, respectively. The hyperparameter tuning is not included as a part of this study to avoid any bias for a particular method.
The time of execution taken by respective approaches is tabulated below, though they are not a similarity metric or a way of comparing the performance in this study.

Sr	Standard deviation	Time 1 (in sec)	Time 2 (in sec)	Time 3 (in sec)
1	5	10.162714	6.641598	2.000310
2	10	10.379607	6.628633	1.682357
3	15	10.264120	6.476949	1.678272
4	20	10.333295	6.497329	1.671890
5	25	10.365459	6.550332	1.687772

From the above table, the third method is much faster than the other two techniques. While there are approximations made and the quality of the denoised image is compromised to an extent which may or may not be bearable depending on the constraints of the real-world problems. In critical safety problems where accuracy is all that matters, the first approach is recommended. Still, if a nearly good quality image is sufficient, then the study can recommend the third approach. The second approach is in between the first and the third in both time complexity and the similarity metric (PSNR).

Conclusion:

Non-local means filtering is a better way of denoising an image when compared to local filtering. The edges of the image are not lost since this method makes use of similar regions. The essential details of the image are not lost while denoising it. Among the three methods of NLM implemented above, it is preferable to use the first method because, in critical scenarios, quality is of the highest priority. The approximate versions can be helpful when the time given for denoising is less and high-quality denoising isn’t required, say in some machine learning problem. In such problems, the machine learning model might be able to make predictions on denoised images without losing edges and contrast of the image. And the condition might require a denoised image as quickly as possible with the model taking care of the rest efficiently. The traditional model performs better for a specific range of ‘h’ because the denoising process must not make an image too smooth or too rough. There can be improvements suggested for each of the three methods discussed in this study. The first one, which is quite efficient, can be made to perform a bit faster. The third approach can make better approximations which might cost some time but might yield better quality and higher PSNR.

Reference:

A. Buades, B. Coll and J. -. Morel, "A non-local algorithm for image denoising," 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), San Diego, CA, USA, 2005, pp. 60-65 vol. 2, doi: 10.1109/CVPR.2005.38.

==

- The assignment ends here. The implementation discussed in this study are given below:

% version 1

```
k = imread('brain.jpg');


k = double(k);
k1 = k + 5 * randn(size(k));
```

```
k2 = k + 10 * randn(size(k));
k3 = k + 15 * randn(size(k));
k4 = k + 20 * randn(size(k));
k5 = k + 25 * randn(size(k));

tic;
y1 = NLM(k, k1);
toc;
d1 = (uint8(k) - y1).^2;
mse1 = mean(d1(:));
rms1 = sqrt(mse1);
psnr1 = 20 * log10(255/rms1);

tic;
y2 = NLM(k, k2);
toc;
d2 = (uint8(k) - y2).^2;
mse2 = mean(d2(:));
rms2 = sqrt(mse2);
psnr2 = 20 * log10(255/rms2);

tic;
y3 = NLM(k, k3);
toc;
d3 = (uint8(k) - y3).^2;
mse3 = mean(d3(:));
rms3 = sqrt(mse3);
psnr3 = 20 * log10(255/rms3);

tic;
y4 = NLM(k, k4);
toc;
d4 = (uint8(k) - y4).^2;
mse4 = mean(d4(:));
rms4 = sqrt(mse4);
psnr4 = 20 * log10(255/rms4);

tic;
y5 = NLM(k, k5);
toc;
```

```
d5 = (uint8(k) - y5).^2;
mse5 = mean(d5(:));
rms5 = sqrt(mse5);
psnr5 = 20 * log10(255/rms5);

function cleared = NLM(I, k1)

   [m, n] = size(I);

   f = 2;
   t = 5;

   su = 1;
   sm = 0;
   ks = 2 * f + 1;

   ker = zeros(ks, ks);

   for x = 1:ks
       for y = 1:ks
           ab = x - f - 1;
           cd = y - f - 1;
           ker(x, y) = 100 * exp(((ab * ab) + (cd * cd))/(-2 * (su *
su)));
           sm = sm + ker(x, y);
       end
   end

   kernel = ker ./ f;
   kernel = kernel / sm;

   noisy = k1;

   cleared = zeros(m, n);

   h = 10;

   noisy2 = padarray(noisy, [f, f], 'symmetric');

   for i = 1:m
```

```
        for j = 1:n
            im = i + f;
            jn = j + f;
            W1 = noisy2(im - f : im + f, jn - f : jn + f);
            rmin = max(im - t, f + 1);
            rmax = min(im + t, m + f);
            smin = max(jn - t, f + 1);
            smax = min(jn + t, n + f);

            NL = 0;
            Z = 0;

            for r = rmin : rmax
                for s = smin : smax
                    W2 = noisy2(r - f : r + f, s - f : s + f);
                    d2 = sum(sum(kernel.*(W1 - W2).*(W1 - W2)));
                    sij = exp(-d2/(h * h));
                    Z = Z + sij;
                    NL = NL + (sij * noisy2(r, s));
                end
            end
            cleared(i, j) = NL/Z;
        end
    end

    cleared = uint8(cleared);

    imtool(k1, []);
    imtool(cleared, []);
end
```

% version-2

```
% function: img_f = nlm(img_n, halfPatchSize, windowHalfSearchSize, N_n,
sigma, h)
% input:    img_n, double image (grayscale) corrupted by Gaussian noise
with std sigma;
%           halfPatchSize, half size of the patch used by NLM;
%           windowHalfSearchSize, half size of the windows used to search
for neighbors;
```

```
%           N_n, number of neighbors used by NLM;
%           sigma, std of the noise;
%           h, filtering parameter for NLM;
% output:   img_f, filtered image.

% parameters
halfPatchSize = 3;           % half size of the patch
windowHalfSearchSize = 6;    % half size for searching the neighbors
N_n = 16;                    % number of neighbors

% create an image to denoise
img = imread('brain.jpg');
img = double(img);

% standard deviation = 5
sigma = 5;                  % noise std
h = 0.3 * sigma^2;           % nlm filtering parameter

img_n_1 = img + randn(size(img)) * sigma;
tic;
denoised_1 = nlm(img_n_1, halfPatchSize, windowHalfSearchSize, N_n, sigma,
h);
toc;

% errors
mse_n_1 = mean((img(:)-img_n_1(:)).^2);
mse_nn_1 = mean((img(:)-denoised_1(:)).^2);
rms1 = sqrt(mse_nn_1);
psnr1 = 20 * log10(255/rms1);

imtool(img_n_1, []);
imtool(denoised_1, []);

% standard deviation = 10
sigma = 10;                  % noise std
h = 0.3 * sigma^2;           % nlm filtering parameter
img_n_2 = img + randn(size(img)) * sigma;
tic;
denoised_2 = nlm(img_n_2, halfPatchSize, windowHalfSearchSize, N_n, sigma,
h);
```

```
toc;

% errors
mse_n_2 = mean((img(:)-img_n_2(:)).^2);
mse_nn_2 = mean((img(:)-denoised_2(:)).^2);
rms2 = sqrt(mse_nn_2);
psnr2 = 20 * log10(255/rms2);

imtool(img_n_2, []);
imtool(denoised_2, []);

% standard deviation = 15
sigma = 15;                   % noise std
h = 0.3 * sigma^2;             % nlm filtering parameter

img_n_3 = img + randn(size(img)) * sigma;
tic;
denoised_3 = nlm(img_n_3, halfPatchSize, windowHalfSearchSize, N_n, sigma,
h);
toc;

% errors
mse_n_3 = mean((img(:)-img_n_3(:)).^2);
mse_nn_3 = mean((img(:)-denoised_3(:)).^2);
rms3 = sqrt(mse_nn_3);
psnr3 = 20 * log10(255/rms3);

imtool(img_n_3, []);
imtool(denoised_3, []);

% standard deviation = 20
sigma = 20;                   % noise std
h = 0.3 * sigma^2;             % nlm filtering parameter
img_n_4 = img + randn(size(img)) * sigma;
tic;
denoised_4 = nlm(img_n_4, halfPatchSize, windowHalfSearchSize, N_n, sigma,
h);
toc;

% errors
```

```
mse_n_4 = mean((img(:)-img_n_4(:)).^2);
mse_nn_4 = mean((img(:)-denoised_4(:)).^2);
rms4 = sqrt(mse_nn_4);
psnr4 = 20 * log10(255/rms4);

imtool(img_n_4, []);
imtool(denoised_4, []);

% standard deviation = 25
sigma = 25;                  % noise std
h = 0.3 * sigma^2;            % nlm filtering parameter
img_n_5 = img + randn(size(img)) * sigma;
tic;
denoised_5 = nlm(img_n_5, halfPatchSize, windowHalfSearchSize, N_n, sigma,
h);
toc;

% errors
mse_n_5 = mean((img(:)-img_n_5(:)).^2);
mse_nn_5 = mean((img(:)-denoised_5(:)).^2);
rms5 = sqrt(mse_nn_5);
psnr5 = 20 * log10(255/rms5);

imtool(img_n_5, []);
imtool(denoised_5, []);


function img_f = nlm(img_n, halfPatchSize, windowHalfSearchSize, N_n,
sigma, h)

   % init
   [ys, xs] = size(img_n);
   cs = 1;
   patchSize = 2 * halfPatchSize + 1;
   P = cs * patchSize^2;
   expected_squared_distance = 2 * sigma^2;

   % Init buffers
   neighbors_indexes = zeros(ys, xs, N_n);
   neighbors_d2 = ones(ys, xs, N_n) * inf;
   padded_img_n = padarray(img_n, [windowHalfSearchSize
windowHalfSearchSize 0], 'symmetric');

   % For each shift
   index = 0;
   for dy = -windowHalfSearchSize:windowHalfSearchSize
       for dx = -windowHalfSearchSize:windowHalfSearchSize

           % shift index
           index = index + 1;

           % shifted image (with mirroring)
           shifted_img_n = padded_img_n((windowHalfSearchSize + 1 +
dy):(windowHalfSearchSize + ys + dy), ...
               (windowHalfSearchSize + 1 + dx):(windowHalfSearchSize + xs
+ dx), ...
               :);

           % squared distance between the shifted and the reference image
           current_d2 = imfilter(sum((shifted_img_n - img_n).^2, 3),
ones(patchSize)/P, 'symmetric');
           current_indexes = ones(ys, xs) * index;

           % update neighbors
           for n = 1 : N_n

               % is the current neighbor closer than the stored one?
               swap = abs(current_d2 - 0 * expected_squared_distance) <
abs(neighbors_d2(:, :, n) - 0 * expected_squared_distance);

               % swap (indexes, distances)
               neighbors_indexes_n = neighbors_indexes(:, :, n);
               buffer_indexes_n = neighbors_indexes_n;
               neighbors_indexes_n(swap) = current_indexes(swap);
               current_indexes(swap) = buffer_indexes_n(swap);
               neighbors_indexes(:, :, n) = neighbors_indexes_n;

               neighbors_d2_n = neighbors_d2(:, :, n);
               buffer_d2_n = neighbors_d2_n;
               neighbors_d2_n(swap) = current_d2(swap);
```

```
                current_d2(swap) = buffer_d2_n(swap);
                neighbors_d2(:, :, n) = neighbors_d2_n;

            end
        end
    end

    % init num / den
    num = zeros(ys, xs, cs);
    den = zeros(ys, xs, cs);

    % do another loop on the possible neighbors to filter
    index = 0;
    for dy = -windowHalfSearchSize:windowHalfSearchSize
        for dx = -windowHalfSearchSize:windowHalfSearchSize

            % shift index
            index = index + 1;

            % shifted image (with mirroring)
            shifted_img_n = padded_img_n((windowHalfSearchSize + 1 +
dy):(windowHalfSearchSize + ys + dy), ...
                (windowHalfSearchSize + 1 + dx):(windowHalfSearchSize + xs
+ dx), ...
                :);

            % For every neighbors
            for n = 1 : N_n

                % weights
                buffer_weights = (neighbors_indexes(:, :, n) == index) .*
exp(-(max(0, neighbors_d2(:, :, n) - 2 * sigma^2))/(h^2));
                weights = repmat(imfilter(buffer_weights, ones(patchSize),
'symmetric'), [1 1 cs]);
                num = num + weights .* shifted_img_n;
                den = den + weights;

            end
        end
    end
```

```
   % filtered image
   img_f = num./den;

end
```

% version-3

```
k = imread('brain.jpg');
k = double(k);

% Denoising parameters
t = 5;
f = 2;
h1 = 1;
h2 = 10;
selfsim = 0;

k1 = k + 5 * randn(size(k));
k2 = k + 10 * randn(size(k));
k3 = k + 15 * randn(size(k));
k4 = k + 20 * randn(size(k));
k5 = k + 25 * randn(size(k));

imtool(k1, []);
tic;
denoised_1 = simple_nlm(k1,t,f,h1,h2,selfsim);
toc;
imtool(denoised_1, []);
d1 = (k - denoised_1).^2;
mse1 = mean(d1(:));
rms1 = sqrt(mse1);
psnr1 = 20 * log10(255/rms1);

imtool(k2, []);
tic;
denoised_2 = simple_nlm(k2,t,f,h1,h2,selfsim);
toc;
imtool(denoised_2, []);
```

```
d2 = (k - denoised_2).^2;
mse2 = mean(d2(:));
rms2 = sqrt(mse2);
psnr2 = 20 * log10(255/rms2);

imtool(k3, []);
tic;
denoised_3 = simple_nlm(k3,t,f,h1,h2,selfsim);
toc;
imtool(denoised_3, []);
d3 = (k - denoised_3).^2;
mse3 = mean(d3(:));
rms3 = sqrt(mse3);
psnr3 = 20 * log10(255/rms3);

imtool(k4, []);
tic;
denoised_4 = simple_nlm(k4,t,f,h1,h2,selfsim);
toc;
imtool(denoised_4, []);
d4 = (k - denoised_4).^2;
mse4 = mean(d4(:));
rms4 = sqrt(mse4);
psnr4 = 20 * log10(255/rms4);

imtool(k5, []);
tic;
denoised_5 = simple_nlm(k5,t,f,h1,h2,selfsim);
toc;
imtool(denoised_5, []);
d5 = (k - denoised_5).^2;
mse5 = mean(d5(:));
rms5 = sqrt(mse5);
psnr5 = 20 * log10(255/rms5);

function [output]=simple_nlm(input,t,f,h1,h2,selfsim)
   %
   %  input   : image to be filtered
   %  t       : radius of search window
   %  f       : radius of similarity window
   %  h1,h2    : w(i,j) = exp(-||GaussFilter(h1) .* (p(i) -
p(j))||_2^2/h2^2)
   %  selfsim : w(i,i) = selfsim, for all i
   %
   %  Note:
   %    if selfsim = 0, then w(i,i) = max_{j neq i} w(i,j), for all i

   [m n]=size(input);
   pixels = input(:);

   s = m*n;

   psize = 2*f+1;
   nsize = 2*t+1;
   % Compute patches
   padInput = padarray(input,[f f],'symmetric');
   filter = fspecial('gaussian',psize,h1);
   patches = repmat(sqrt(filter(:))',[s 1]) .* im2col(padInput, [psize
psize], 'sliding')';

   % Compute list of edges (pixel pairs within the same search window)
   indexes = reshape(1:s, m, n);
   padIndexes = padarray(indexes, [t t]);
   neighbors = im2col(padIndexes, [nsize, nsize], 'sliding');
   TT = repmat(1:s, [nsize^2 1]);
   edges = [TT(:) neighbors(:)];
   RR = find(TT(:) >= neighbors(:));
   edges(RR, :) = [];

   % Compute weight matrix (using weighted Euclidean distance)
   diff = patches(edges(:,1), :) - patches(edges(:,2), :);
   V = exp(-sum(diff.*diff,2)/h2^2);
   W = sparse(edges(:,1), edges(:,2), V, s, s);

   % Make matrix symmetric and set diagonal elements
   if selfsim > 0
   W = W + W' + selfsim*speye(s);
   else
   maxv = max(W,[],2);
   W = W + W' + spdiags(maxv, 0, s, s);
   end

   % Normalize weights
   W = spdiags(1./sum(W,2), 0, s, s)*W;

   % Compute denoised image
   output = W*pixels;
   output = reshape(output, m , n);
end
```
