CS 663: Assignment 2

Group Members. Rishabh Shah 150050006
Anmol Mishra 150010041
Shriram SB 150050099

Notes and Observations

NOTE: Since none of our team members were able to register in Turnitin, we are attaching our Tcode and (fast to generate images). The published part contains output for parts 1 and 2 including RMSD values of part 2.
The tuned parameters can be found in the published code

Question 1 (Unsharp Masking)

Clearly as we can see from output of the two images, unsharp masking (using difference of gaussians to sharpen the image) produces images of better contrast along almost all edges in Lion image . It highlights bright spots in moon image and darkens the black spots creating a much better sharpened image.

Question 2 (Bilateral Filtering)

5 % is just too less a error that can be seen when added in barbara image so we have used 10% for this image. Also, since we were provided with the noisy versions of grass and honey comb images we did not over corrupt them. The plots and calculations are done using this consideration. Also, we have performed linear contrast stretching on the images (original and corrupted one) to [0,1], so the root mean square difference values can be also compared across the images.

Bilateral images preserved most of the edges in barbara and honeycomb. But smooth textures link internal parts of honeycomb, and many portions of grass images were blurred out creating a cartoonish appearance. Although the noise in the images was significantly reduced and images obtained were good enough with low values of RMSD.

The RMSDs are reported in published PDF from MATLAB.

Question 3 (Patch Based Filtering)

Even here we have used noisy images provided for grass and honeycomb. Also, due to long run times we downsampled barbara image by 2 after addition of gaussian noise with sigma = 0.3 * range. (30%). 15. After hours of tuning the parameters the performance of patch based filtering though not up to the expectations (raised high in classes), was a bit better than bilateral filtering. We can see its effects in the grass image where bilateral performs way worse. Also, barbara image is much more sharper and honeycomb image retains more of its details. Barbara noise addition is stochastic and different RSMD was observed for different runs. We've reported sigma which gave best visual results.

RMSD:

Barbara : sigma: 0.058 0.9 * sigma
HoneyComb: 0.045
Grass: 0.038

- Barbara
    - Sigma - 0.058
    - 0.9 * sigma - 0.057
    - 1.1 * sigma - 0.059
- HoneyComb
    - Sigma - 0.045
    - 0.9 * sigma - 0.048
    - 1.1 * sigma - 0.043
- Grass
    - Sigma - 0.038
    - 0.9 * sigma - 0.040
    - 1.1 * sigma - 0.038

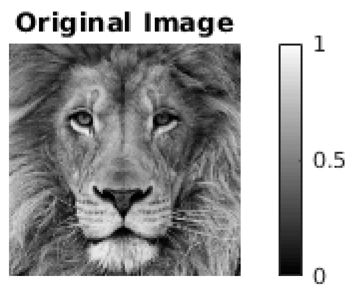
MainScript

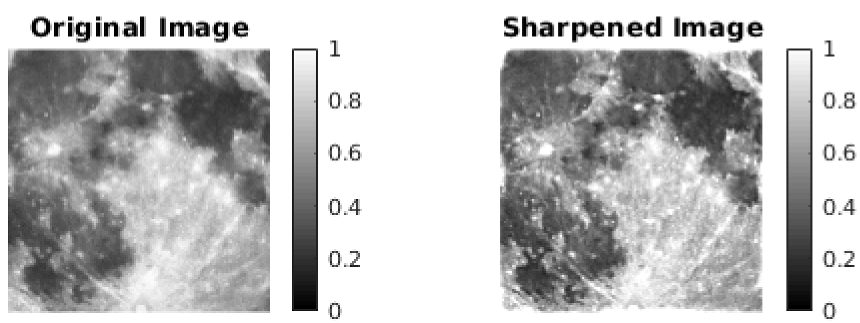
```
myNumOfColors = 256;
myColorScale = [[0:1/(myNumOfColors-1):1]' ,[0:1/
(myNumOfColors-1):1]', [0:1/(myNumOfColors-1):1]'];
```

Your code here

```
tic;
image_paths = ["../data/lionCrop.mat", "../data/superMoonCrop.mat"];
scale = [1,1];      %scale values for images
sigma = [5,10];     %sigma values for images
for i=1:2
    load(image_paths(i));

    img = imageOrig;
    [stretched_img, ~] = myLinearConstrastStretching(img);  %Linear
 Contrast Stretching

    h = figure;

    subplot(1,2,1);imshow(stretched_img);title('Original Image');
    daspect([1 1 1]);
    colormap(myColorScale);
    axis tight;
    colorbar;

    % Sharpening the image through myUnsharpMasking function
    sharp_img = myUnsharpMasking(stretched_img, sigma(i), scale(i));
    [stretched_sharp_img, ~] = myLinearConstrastStretching(sharp_img);
    subplot(1,2,2);imshow(stretched_sharp_img);title('Sharpened
 Image');
    daspect([1 1 1]);
    colormap(myColorScale);
    axis tight;
    colorbar;
end
toc;
```

Elapsed time is 0.501588 seconds.


Original Image
Sharpened Image
1
0.5
0
1
0.5
0


Original Image
Sharpened Image
1
0.8
0.6
0.4
0.2
0
1
0.8
0.6
0.4
0.2
0

Published with MATLAB® R2018a

```
function [sharpened_img] = myUnsharpMasking(img, std_dev, scale)
%UNTITLED2 Summary of this function goes here
%   Detailed explanation goes here
    filter = fspecial('gaussian', 6 * ceil(std_dev) + 1, std_dev);
    smoothed_img = imfilter(img, filter);
    sharpened_img = img + scale * (img - smoothed_img);
end
```

Published with MATLAB® R2018a

```
function [img, stretched_img] = myLinearConstrastStretching(img)
%MYLINEARCONSTRASTSTRETCHING Summary of this function goes here
%   Detailed explan.ation goes here
    img = double(img);
    stretched_img = (img - min(min(img))) / (max(max(img)) -
 min(min(img)));

end
```

Published with MATLAB® R2018a

MainScript

```
myNumOfColors = 256;
myColorScale = [[0:1/(myNumOfColors-1):1]' ,[0:1/
(myNumOfColors-1):1]', [0:1/(myNumOfColors-1):1]'];
```

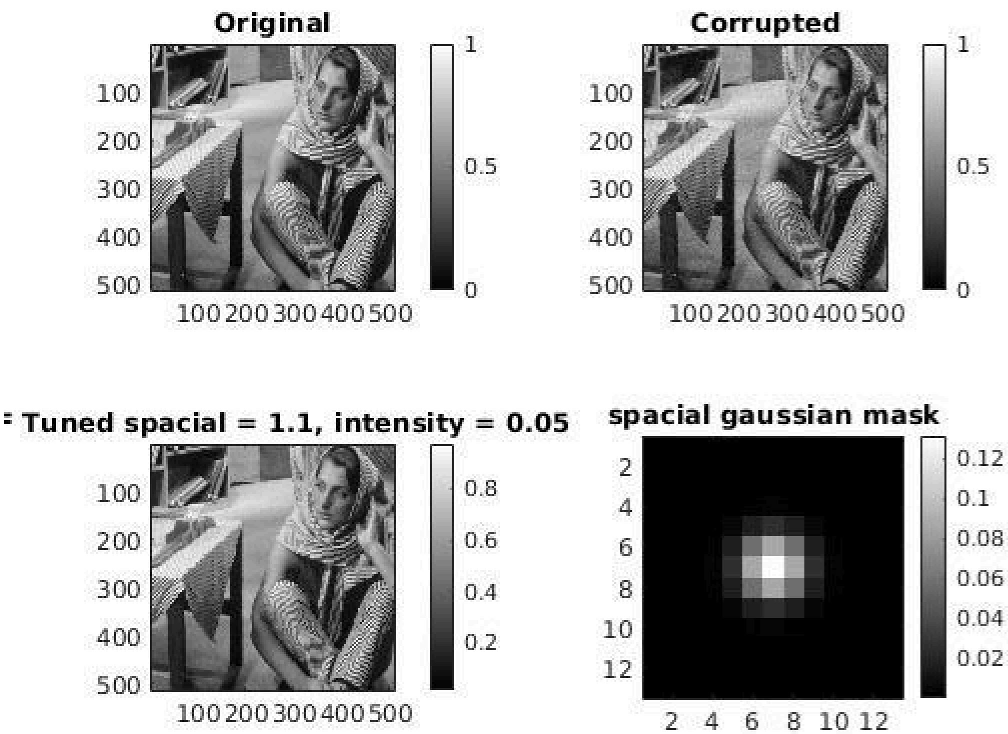
Running on 3 images

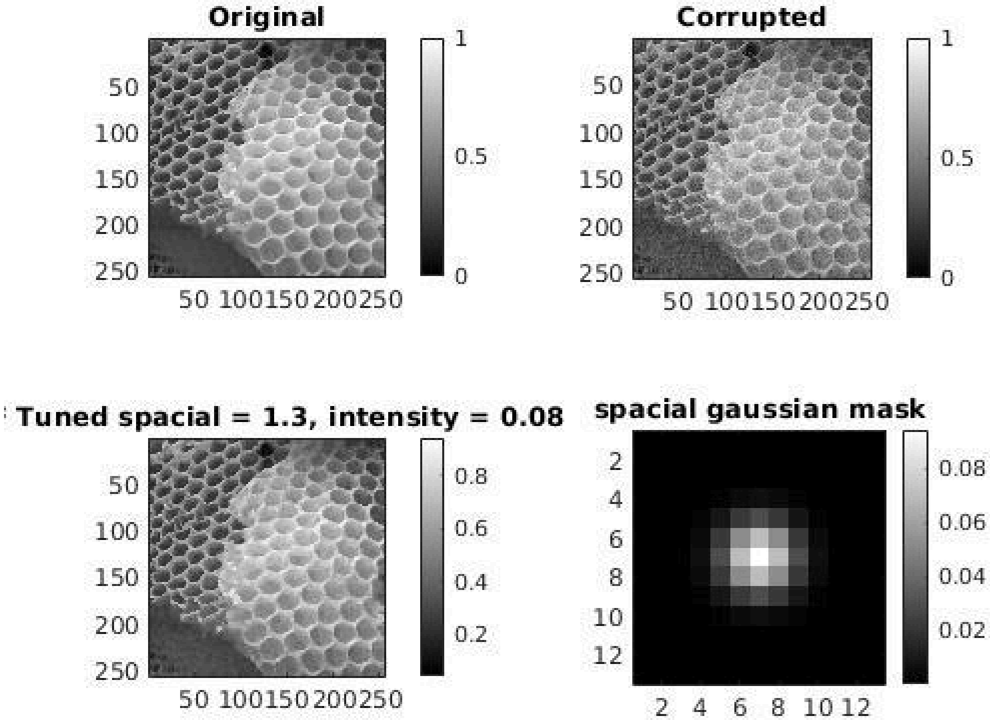
```
tic;
image_paths = ["barbara", "honeyCombReal", "grass"];
% RMSD
rmsd = @(a, b) (sum(sum((a-b).*(a-b)))/(length(a(1:end)))) .^ 0.5;
stretch = @(img) (img - min(min(img))) / (max(max(img)) -
 min(min(img)));
% Tuned Parameters for each image
sigma_spacial = [1.1, 1.3, 1.1];
sigma_intensity = [0.05, 0.08, 0.1];
for i=1:3
    % Different loading mechanisms for different images
    if(i == 1)
        I = load("../data/barbara.mat");
        image = I.imageOrig;
        range = max(max(image)) - min(min(image));
        I_C = image + (range * 0.10) * rand(512, 512);

    elseif( i == 2)
        image = imread('../data/honeyCombReal.png');
        I_C = load('../data/honeyCombReal_Noisy.mat');
        I_C = I_C.imgCorrupt;

    else
        image = imread('../data/grass.png');
        I_C = load('../data/grassNoisy.mat');
        I_C = I_C.imgCorrupt;
    end

    % Linear constrast stretching images so as to make them comparable
 for
    % RMSD  calulations
    image = double(image);
    image = stretch(image);

    I_C = double(I_C);
    I_C = stretch(I_C);

    % BF function for optimal sigma1, sigma2
    [I_BF, mask] = myBilateralFiltering(I_C, sigma_spacial(i),
 sigma_intensity(i));
    fprintf('RMSD for %s for optimal case is %f\n',
 char(image_paths(i)), rmsd(I_BF, image));

    % BF function for 0.9 sigma1, sigma2
```

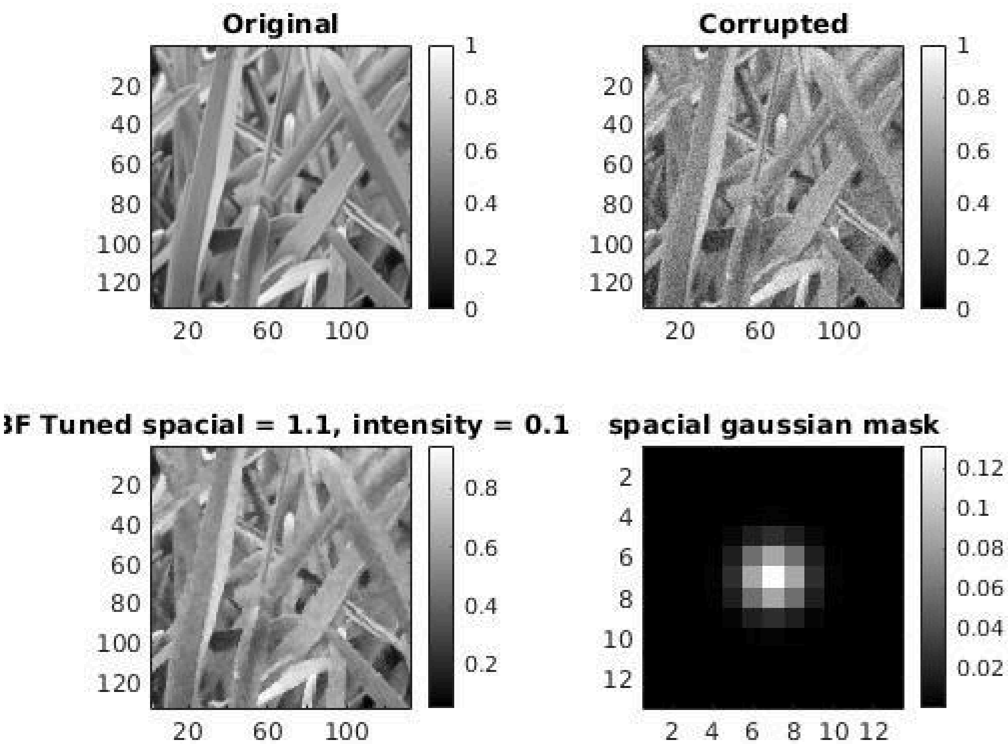
```
    [I_BF1, ~]= myBilateralFiltering(I_C, 0.9 * sigma_spacial(i),
 sigma_intensity(i));
    fprintf('RMSD for %s: 0.9 * spacial, 1.0 * intensity is %f\n',
 char(image_paths(i)), rmsd(I_BF1, image));

    % BF function for 1.1 sigma1, sigma2
    [I_BF2, ~]= myBilateralFiltering(I_C, 1.1 * sigma_spacial(i),
 sigma_intensity(i));
    fprintf('RMSD for %s: 1.1 * spacial, 1.0 * intensity is %f\n',
 char(image_paths(i)), rmsd(I_BF2, image));

    % BF function for sigma1, 0.9 sigma2
    [I_BF3, ~]= myBilateralFiltering(I_C, sigma_spacial(i), 0.9 *
 sigma_intensity(i));
    fprintf('RMSD for %s: 1.0 * spacial, 0.9 * intensity is %f\n',
 char(image_paths(i)), rmsd(I_BF3, image));

    % BF function for sigma1, 1.1 sigma2
    [I_BF4, ~]= myBilateralFiltering(I_C, sigma_spacial(i), 1.1 *
 sigma_intensity(i));
    fprintf('RMSD for %s: 1.0 * spacial, 1.1 * intensity is %f\n',
 char(image_paths(i)), rmsd(I_BF4, image));

    % Plotting all images
    h = figure;
    subplot(2, 2, 1), imagesc(image), title('Original');
    daspect([1 1 1]);
    colormap(myColorScale);
    axis tight;
    colorbar;

    subplot(2, 2, 2), imagesc(I_C), title('Corrupted');
    daspect([1 1 1]);
    colormap(myColorScale);
    axis tight;
    colorbar;

    subplot(2, 2, 3), imagesc(I_BF), title("BF Tuned spacial = " +
 sigma_spacial(i) + ", intensity = " + sigma_intensity(i));
    daspect([1 1 1]);
    colormap(myColorScale);
    axis tight;
    colorbar;

    subplot(2, 2, 4), imagesc(mask), title("spacial gaussian mask");
    daspect([1 1 1]);
    colormap(myColorScale);
    axis tight;
    colorbar;
end
toc;
```

RMSD for barbara for optimal case is 0.025322
*RMSD for barbara: 0.9 * spacial, 1.0 * intensity is 0.025347*

```
RMSD for barbara: 1.1 * spacial, 1.0 * intensity is 0.025333
RMSD for barbara: 1.0 * spacial, 0.9 * intensity is 0.025429
RMSD for barbara: 1.0 * spacial, 1.1 * intensity is 0.025349
RMSD for honeyCombReal for optimal case is 0.040800
RMSD for honeyCombReal: 0.9 * spacial, 1.0 * intensity is 0.040614
RMSD for honeyCombReal: 1.1 * spacial, 1.0 * intensity is 0.041076
RMSD for honeyCombReal: 1.0 * spacial, 0.9 * intensity is 0.041231
RMSD for honeyCombReal: 1.0 * spacial, 1.1 * intensity is 0.040615
RMSD for grass for optimal case is 0.038663
RMSD for grass: 0.9 * spacial, 1.0 * intensity is 0.038456
RMSD for grass: 1.1 * spacial, 1.0 * intensity is 0.039015
RMSD for grass: 1.0 * spacial, 0.9 * intensity is 0.038979
RMSD for grass: 1.0 * spacial, 1.1 * intensity is 0.038602
Elapsed time is 11.093121 seconds.
```




Original
Corrupted
Tuned spacial = 1.3, intensity = 0.08
spacial gaussian mask


Original
Corrupted
Tuned spacial = 1.1, intensity = 0.1
spacial gaussian mask

Published with MATLAB® R2018a

```
function [final_image, spacial_guassian_mask] =
 myBilateralFiltering(image, spacial_sigma, intensity_sigma)
    final_image = image;
    [rows, cols] = size(image);

    % Step 1: Create function for 2 guassians according to given input
 parameters
    f =@(x, sigma) exp(-double(x) .^ 2 / (2 * sigma ^ 2));

    % Step 2: Create Spacial guassian window
    window_size = 6 * ceil(spacial_sigma);
    if mod(window_size,2) == 0
        window_size = window_size + 1;
    end

    W = fspecial('gaussian', window_size, spacial_sigma);

    % Step 3: Convolution
    for r =1:rows
        for c =1:cols

            % Window limits in imput image
            left_lim = r - (window_size-1)/2;
            right_lim = r + (window_size-1)/2;
            down_lim = c - (window_size-1)/2;
            up_lim = c + (window_size-1)/2;

            % Window limits in mask
            left_lim_x = 1;
            right_lim_x = window_size;
            down_lim_x = 1;
            up_lim_x = window_size;

            % Crop window
            if r - (window_size-1)/2 < 1
               left_lim = 1;
               left_lim_x = (window_size+1)/2 - r + 1;
            end

            if r + (window_size-1)/2 > rows
               right_lim = rows;
               right_lim_x = (window_size+1)/2 + rows - r;
            end

            if c - (window_size-1)/2 < 1
               down_lim = 1;
               down_lim_x = (window_size+1)/2 - c + 1;
            end

            if c + (window_size-1)/2 > cols
               up_lim = cols;
               up_lim_x = (window_size+1)/2 + cols - c;
```

```
            end

            % Window and mask creation from image
            window = image(left_lim:right_lim, down_lim:up_lim);
            W_here = W(left_lim_x:right_lim_x, down_lim_x:up_lim_x);
            mask = W_here .* f(window - image(r,c), intensity_sigma);

            % Calculation of convoluted intensities
            numerator = sum(sum(mask .* window));
            final_image(r,c) = numerator/sum(sum(mask));
        end
    end

    spacial_guassian_mask = W;
end
```

Published with MATLAB® R2018a

MyMainScript

```
myNumOfColors = 256;
myColorScale = [[0:1/(myNumOfColors-1):1]' ,[0:1/
(myNumOfColors-1):1]', [0:1/(myNumOfColors-1):1]'];
```

Patch Based Filtering for 3 given images

```
tic;
image_paths = ["barbara", "honeyCombReal", "grass"];
o_image_paths = ["../Report/barbara.mat", "../Report/
honeyComb.mat", "../Report/grass.mat"];
% RMSD
rmsd = @(a, b) (sum(sum((a-b).*(a-b)))/(length(a(1:end)))) ^ 0.5;
stretch = @(img) (img - min(min(img))) / (max(max(img)) -
 min(min(img)));
% standard deviation for weighing patch (making it isotropic)
patch_weights_std_dev = [3, 3, 1.5];
% standard deviation for gaussian function of patch distance (h as
% represented in class)
gaussian_weights = [0.04, 0.035, 0.047];
for i=1:3
    if(i == 1)
        % downsampling only barbara image by 2
        I = load("../data/barbara.mat");
        image = I.imageOrig;
        range = max(max(image)) - min(min(image));
        I_C = image + (range * 0.3) * rand(512, 512);
        gaussian_blur_filter = fspecial('gaussian', 5, 0.66);
        I_C = imfilter(I_C, gaussian_blur_filter);
        image = image(1:2:end, 1:2:end);
        I_C = I_C(1:2:end, 1:2:end);

    elseif( i == 2)
        image = imread('../data/honeyCombReal.png');
        I_C = load('../data/honeyCombReal_Noisy.mat');
        I_C = I_C.imgCorrupt;

    else
        image = imread('../data/grass.png');
        image = double(image);
        range = max(max(image)) - min(min(image));
        I_C = load('../data/grassNoisy.mat');
        I_C = I_C.imgCorrupt;
    end
    % image - original image, I_C - corrupted image
    image = double(image);
    image = stretch(image);
    I_C = double(I_C);
    I_C = stretch(I_C);

    % PF function for optimal sigma
```

```
    [I_PF, mask] = myPatchBasedFiltering(I_C,
 patch_weights_std_dev(i), gaussian_weights(i));
    I_PF = stretch(I_PF);
    fprintf('RMSD for %s for case 0 is %f\n', char(image_paths(i)),
 rmsd(I_PF, image));

    % PF function for 0.9 sigma
    [I_PF1, ~] = myPatchBasedFiltering(I_C, patch_weights_std_dev(i),
 0.9 * gaussian_weights(i));
    I_PF1 = stretch(I_PF1);
    fprintf('RMSD for %s for case 1 is %f\n', char(image_paths(i)),
 rmsd(I_PF1, image));

    % PF function for 1.1 sigma
    [I_PF2, ~] = myPatchBasedFiltering(I_C, patch_weights_std_dev(i),
 1.1 * gaussian_weights(i));
    I_PF2 = stretch(I_PF2);
    fprintf('RMSD for %s for case 2 is %f\n', char(image_paths(i)),
 rmsd(I_PF2, image));

    % Plotting all images
    h = figure;
    subplot(2, 2, 1), imagesc(image), title('Original');
    daspect([1 1 1]);
    colormap gray;
    axis tight;
    colorbar;

    subplot(2, 2, 2), imagesc(I_C), title('Corrupted');
    daspect([1 1 1]);
    colormap gray;
    axis tight;
    colorbar;

    subplot(2, 2, 3), imagesc(I_PF), title("Patch weights stddev =
 " + patch_weights_std_dev(i) + ", gaussian intensity weights = " +
 gaussian_weights(i));
    daspect([1 1 1]);
    colormap gray;
    axis tight;
    colorbar;

    subplot(2, 2, 4), imagesc(mask), title("spacial gaussian mask");
    daspect([1 1 1]);
    colormap gray;
    axis tight;
    colorbar;

    % saving images to file - barbara takes more than 5 minutes
    save(char(o_image_paths(i)), 'image', 'I_C', 'I_PF', 'mask');
end
toc;
```

Published with MATLAB® R2018a

```
function [filtered_img, patch_weights] = myPatchBasedFiltering(img,
 patch_weights_std_dev, h)

    patch_size = 9; half_patch_size = (patch_size - 1) / 2;
    window_size = 25; half_window_size = (window_size - 1) / 2;
    [rows, cols] = size(img);

    % gaussian weighting of patch intensity vector for making
 filtering isotropic
    patch_weights = fspecial('gaussian', patch_size,
 patch_weights_std_dev);

    filtered_img = img;
    for r = 1 : rows
        for c = 1 : cols
            % weights for each pixel inside the window
            pixel_weights = zeros(window_size, window_size);
            % patch width on (top, bottom, left, right) of center of
 window
            % center of window - p
            p_patch_lim = zeros(1, 4);
            p_patch_lim(1) = r - max(r - half_patch_size, 1);
            p_patch_lim(2) = min(r + half_patch_size, rows) - r;
            p_patch_lim(3) = c - max(c - half_patch_size, 1);
            p_patch_lim(4) = min(c + half_patch_size, cols) - c;

            % window width (top, bottom, left, right) of center of
 window
            window_lim = zeros(1, 4);
            window_lim(1) = r - max(r - half_window_size, 1);
            window_lim(2) = min(r + half_window_size, rows) - r;
            window_lim(3) = c - max(c - half_window_size, 1);
            window_lim(4) = min(c + half_window_size, cols) - c;

            % iterating over each point inside the window
            for i = r - window_lim(1) : r + window_lim(2)
                for j = c - window_lim(3) : c + window_lim(4)
                    % q - a point inside the window
                    % patch width on (top, bottom, left, right) of q
                    q_patch_lim = zeros(1, 4);
                    q_patch_lim(1) = i - max(i - half_patch_size, 1);
                    q_patch_lim(2) = min(i + half_patch_size, rows) -
 i;
                    q_patch_lim(3) = j - max(j - half_patch_size, 1);
                    q_patch_lim(4) = min(j + half_patch_size, cols) -
 j;

                    % matching the patch dimensions around p and q to
                    % handle boundary of images
                    patch_lim = min(p_patch_lim, q_patch_lim);

                    % extracting patch around p and q
```

```
                   p_patch = img(r - patch_lim(1) : r + patch_lim(2),
c - patch_lim(3) : c + patch_lim(4));
                   q_patch = img(i - patch_lim(1) : i + patch_lim(2),
j - patch_lim(3) : j + patch_lim(4));

                   % extracting gaussian weights for patch from
                   % patch_weights
                   clipped_patch_weights =
patch_weights(half_patch_size + 1 - patch_lim(1) : half_patch_size
+ 1 + patch_lim(2), half_patch_size + 1 - patch_lim(3) :
half_patch_size + 1 + patch_lim(4));
                   % multiplying gaussian weight to patch and taking
                   % difference
                   cur_weight = sum(sum((clipped_patch_weights .*
p_patch - clipped_patch_weights .* q_patch).^2)) /
sum(sum(clipped_patch_weights.^2));
                   % weight for current point
                   cur_weight = exp(-cur_weight / h.^2);
                   pixel_weights(i - r + half_window_size + 1, j - c
+ half_window_size + 1) = cur_weight;
               end
           end

           % normalizing weights
           pixel_weights = pixel_weights ./ sum(sum(pixel_weights));
           % weighted sum of points in the window to get filtered
image
           filtered_img(r, c) =
sum(sum(pixel_weights(half_window_size + 1 - window_lim(1) :
half_window_size + 1 + window_lim(2), half_window_size + 1 -
window_lim(3) : half_window_size + 1 + window_lim(4)) .* img(r
- window_lim(1) : r + window_lim(2), c - window_lim(3) : c +
window_lim(4))));
       end
   end

end
```

Published with MATLAB® R2018a