ECE 558 Project 02

Problem 1

a) 2D Convolution –

p1_conv2d.m - Main script file to implement 2d convolution on the input image with provided kernels

```
% Project 2 Question 1a and 1b
%% Convolution. Spatial filtering in images clc; clear all; close all;
img = im2double(imread('lena.png'));
% img = rgb2gray(img);
img1 = im2double(imread('wolves.png'));
img1 = rgb2gray(img1); %considering a grayscale conversion for wolves.png
%defining all the filters
box_deriv = {1/9*[1,1,1;1,1,1;1,1,1],[1,-1],[1;-1]};
prewitt = {[-1,0,1;-1,0,1;-1,0,1],[-1,-1,-1;0,0,0;1,1,1]};
sobel = {[-1,0,1;-2,0,2;-1,0,1],[1,2,1;0,0,0;-1,-2,-1]};
roberts = {[0,1 ; -1,0],[1,0;0,-1]};
%creating a cell of all kernels kernels = horzcat(box_deriv,prewitt, sobel, roberts);
%setting prompts for selecting kernel and padding type
prompt = 'Set padding type\n1. Zero Padding\n2. Copy Edge\n3. Wrap around\n4. Reflect
across edge\n';
prompt2 = ['Set Kernel type\n1. Box Filter 2. Derivative horizontal'...
    ' 3. Derivative vertical\n4. Prewitt horizontal 5. Prewitt vertical\n'...
    '6. Sobel horizontal 7. Sobel vertical\n8. Roberts Horizontal'...
    ' 9 Roberts Vertical\n'];
%setting up cells of kernel and padding type for saving output with
%suitable filename padType = {'Zero', 'Copy Edge', 'Wrap Around', 'Reflect'};
kernType = {'Box', 'DerivHorizontal', 'DerivVertical', 'PrewittHorizontal',...
    'PrewittVertical','SobelHorizontal','SobelVertical', 'RobertsHorizontal',...
    'RobertsVertical'};
kern = input(prompt2); %take input of kernel from user pad = input(prompt); %take padding
type from user if size(img,3)==3 %if the image is a color image, convolve channels
separately     op(:,:,1) = conv2d(img(:,:,1), kernels{kern},pad);
    op(:,:,2) = conv2d(img(:,:,2), kernels{kern},pad);
    op(:,:,3) = conv2d(img(:,:,3), kernels{kern},pad);
else     op = conv2d(img, kernels{kern},pad); %else convolve the image directly
end ops = (op-min(op(:)))./(max(op(:))-min(op(:))); %scale image to visualize negative
values better
%the process is repeated to check wolves.png as a grayscale image.  if size(img1,3)==3
op1(:,:,1) = conv2d(img1(:,:,1), kernels{kern},pad);
    op1(:,:,2) = conv2d(img1(:,:,2), kernels{kern},pad);
    op1(:,:,3) = conv2d(img1(:,:,3), kernels{kern},pad);
else     op1 = conv2d(img1, kernels{kern},pad);
end ops1 = (op1-min(op1(:)))./(max(op1(:))-min(op1(:)));
%create a figure to display the results
figure('units', 'normalized', 'outerposition', [0 0 1 1]);
subplot(2,3,1);
imshow(img);
title('lena.png');
subplot(2,3,2);
imshow(op);
title('o/p (Negative values Clipped)');
subplot(2,3,3)
```

```
imshow(ops);
title('o/p Scaled');
subplot(2,3,4);
imshow(img1);
title('wolves.png');
subplot(2,3,5);
imshow(op1);
title('o/p (Negative values Clipped)');
subplot(2,3,6)
imshow(ops1);
title('o/p Scaled');
%save the results to a png file
fileName = horzcat('Part1a',kernType{kern},padType{pad},'.png');
print(gcf, fileName,'-dpng', '-r300');
%% part 2  % unit impulse function   %the unit impulse is defined with a matrix of zeros
with a one in the
%center as shown below
uimp = zeros(1024);
uimp(512, 512) = 1;
ouimp = conv2d(uimp, kernels{6}, pad);
%show the results in the figure figure;
subplot(1,2,1)
imshow(uimp)
subplot(1,2,2)
imshow(ouimp)
```

Padding is done using the *SetPadding.m* function which takes the image, kernel and padding type as inputs. In this function any oddxodd sized square kernels evenxeven sized square kernels and the special derivative kernels can be handled for all 4 types of padding.

```
function [img_pad] = SetPadding(img, kern, pad)
%SETPADDING Set padding based on kernel size
%   Set padding to set up an image for convolution based on the kernel size
%   of the chosen kernel. The size of the kernel is used as a reference for
%   the amount of padding required. The padding is performed by considering
%   padding type and then slicing appropriate indices from original image
%   to copy them to the actual image.
[r,c] = size(img); %size of image
[rk, ck] = size(kern); %size of kernel
if rk==1 && ck==2 %special case for horizontal derivative filter
    if pad == 1 %zero padding
        img_pad = zeros(r,c+1); %add a column
        img_pad(:, 1:end-1) = img;
    elseif pad == 2 %copy edge
        img_pad = zeros(r,c+1);
        img_pad(:, 1:end-1) = img;
        img_pad(:, end) = img(:,end);
    elseif pad == 3 %wrap around
        img_pad = zeros(r,c+1);
        img_pad(:, 1:end-1) = img;
        img_pad(:, end) = img(:,1);
    elseif pad == 4 %reflect across edge
        img_pad = zeros(r,c+1);
        img_pad(:, 1:end-1) = img;
        img_pad(:, end) = img(:,end);
    end
elseif rk==2 && ck==1 %special case for horizontal derivative filter
    if pad == 1 %zero padding
        img_pad = zeros(r+1,c);
```

```
        img_pad(1:end-1,:) = img;
    elseif pad == 2 %copy edge
        img_pad = zeros(r+1,c);
        img_pad(1:end-1,:) = img;
        img_pad(end,:) = img(end,:);
    elseif pad == 3 %wrap around
        img_pad = zeros(r+1,c);
        img_pad(1:end-1,:) = img;
        img_pad(end,:) = img(1,:);
    elseif pad == 4 %reflect across edge
        img_pad = zeros(r+1,c);
        img_pad(1:end-1,:) = img;
        img_pad(end,:) = img(end,:);
    end
elseif (mod(rk,2) && mod(ck,2)) %for odd sized kernels. (3x3, 5x5...)
    if pad == 1 %zero padding
        img_pad = zeros(r+ceil(rk/2),c+ceil(ck/2)); %add ceil(kernelsize/2) number of rows
and cols
        img_pad((rk+1)/2:end-1, (ck+1)/2:end-1) = img;
    elseif pad == 2 %copy edge
        %set up as zero padding
        img_pad = zeros(r+ceil(rk/2),c+ceil(ck/2));
        img_pad((rk+1)/2:end-1, (ck+1)/2:end-1) = img;
        %copy edges to the newly added edges
        img_pad(floor((rk+1)/2):end-1, 1:floor(ck/2)) = repmat(img(:,1),floor(ck/2));
        img_pad(floor((rk+1)/2):end-1, end-floor(ck/2)+1:end) =
repmat(img(:,end),floor(rk/2));
        img_pad(1:floor(rk/2),floor((ck+1)/2):end-1) = repmat(img(1,:),floor(rk/2));
        img_pad(end-floor(rk/2)+1:end,floor((ck+1)/2):end-1) =
repmat(img(end,:),floor(ck/2));
        %corner cases
        img_pad(1:floor(rk/2), 1:floor(ck/2)) = repmat(img(1,1), floor(rk/2), floor(ck/2));
        img_pad(end-floor(rk/2):end, 1:floor(ck/2)) = repmat(img(end,1), floor(rk/2),
floor(ck/2));
        img_pad(1:floor(rk/2), end-floor(ck/2):end) = repmat(img(1,end), floor(rk/2),
floor(ck/2));
        img_pad(end-floor(rk/2):end, end-floor(ck/2):end) = repmat(img(end,end),
floor(rk/2), floor(ck/2));
    elseif pad == 3 %wrap around
        %set up as zero padding
        img_pad = zeros(r+ceil(rk/2),c+ceil(ck/2));
        img_pad((rk+1)/2:end-1, (ck+1)/2:end-1) = img;
        %wrap edges from other end to the newly added edges
        img_pad(floor((rk+1)/2):end-1, 1:floor(ck/2)) = img(:,end-floor(ck/2)+1:end);
        img_pad(floor((rk+1)/2):end-1, end-floor(ck/2)+1:end) = img(:,1:floor(ck/2));
        img_pad(1:floor(rk/2),floor((ck+1)/2):end-1) = img(end-floor(rk/2)+1:end,:);
        img_pad(end-floor(rk/2)+1:end,floor((ck+1)/2):end-1) = img(1:floor(rk/2),:);
        %corner cases
        img_pad(1:floor(rk/2), 1:floor(ck/2)) = img(end-floor(rk/2)+1:end, end-
floor(ck/2)+1:end);
        img_pad(end-floor(rk/2)+1:end, 1:floor(ck/2)) = img(1:floor(rk/2), end-
floor(ck/2)+1:end);
        img_pad(1:floor(rk/2), end-floor(ck/2)+1:end) = img(end-floor(rk/2)+1:end,
1:floor(ck/2));
        img_pad(end-floor(rk/2)+1:end, end-floor(ck/2)+1:end) = img(1:floor(rk/2),
1:floor(ck/2));
    elseif pad == 4 %reflect across edge
        %set up as zero padding
```

```
        img_pad = zeros(r+ceil(rk/2),c+ceil(ck/2));
        img_pad((rk+1)/2:end-1, (ck+1)/2:end-1) = img;
        %reflect across edges
        img_pad(floor((rk+1)/2):end-1, 1:floor(ck/2)) = img(:,floor(ck/2):1);
        img_pad(floor((rk+1)/2):end-1, end-floor(ck/2)+1:end) = img(:,end:end-
floor(ck/2)+1);
        img_pad(1:floor(rk/2),floor((ck+1)/2):end-1) = img(floor(rk/2):1,:);
        img_pad(end-floor(rk/2)+1:end,floor((ck+1)/2):end-1) = img(end:end-
floor(rk/2)+1,:);
        %corner cases
        img_pad(1:floor(rk/2), 1:floor(ck/2)) = img(floor(rk/2):1, floor(ck/2):1);
        img_pad(end-floor(rk/2)+1:end, 1:floor(ck/2)) = img(end:end-floor(rk/2)+1,
floor(ck/2):1);
        img_pad(1:floor(rk/2), end-floor(ck/2)+1:end) = img(floor(rk/2):1, end:end-
floor(ck/2)+1);
        img_pad(end-floor(rk/2)+1:end, end-floor(ck/2)+1:end) = img(end:end-floor(rk/2)+1,
end:end-floor(ck/2)+1);
    end
else %for even sized kernels (eg. 2x2, 4x4...)
    if pad == 1 %zero padding
        img_pad = zeros(r+rk/2,c+ck/2);
        img_pad(1:end-(rk/2), 1:end-(ck/2)) = img;
    elseif pad == 2 %copy edge
        %set up as zero padding
        img_pad = zeros(r+rk/2,c+ck/2);
        img_pad(1:end-(rk/2), 1:end-(ck/2)) = img;
        %copy edges to the image.
        img_pad(1:end-(rk/2), end-ck/2+1:end) = repmat(img(:,end), floor(ck/2));
        img_pad(end-rk/2+1:end,1:end-(ck/2)) = repmat(img(end,:), floor(rk/2));
        %corner cases
        img_pad(end-(rk/2)+1:end, end-(ck/2)+1:end) = repmat(img(end,end),
floor(rk/2),floor*ck/2));
    elseif pad == 3 %wrap around
        %set up as zero padding
        img_pad = zeros(r+rk/2,c+ck/2);
        img_pad(1:end-(rk/2), 1:end-(ck/2)) = img;
        %wrap around edges
        img_pad(1:end-1, end-ck/2+1:end) = img(:,1:ck/2);
        img_pad(end-rk/2+1:end,1:end-1) = img(1:rk/2,:);
        %corner case
        img_pad(end-(rk/2)+1:end, end-(ck/2)+1:end) = img(1:(rk/2),1:(ck/2));
    elseif pad == 4 %reflect across edge
        img_pad = zeros(r+rk/2,c+ck/2);
        img_pad(1:end-1, 1:end-1) = img;
        %reflect across edge
        img_pad(1:end-1, end-ck/2+1:end) = img(:,end:end-ck/2+1);
        img_pad(end-rk/2+1:end,1:end-1) = img(end:end-rk/2+1,:);
        %corner case
        img_pad(end-(rk/2)+1:end, end-(ck/2)+1:end) = img(end:end-(rk/2)+1,end:end-
(ck/2)+1);
end
end
```

The actual Convolution is done using the *conv2d.m* function which takes the image, kernel and padding type as inputs.

```
function [op_img] = conv2d(img,kern, pad)
%CONV2 perform 2d convolution of an image with given kernel
%   Perform 2d convolution of an image.
    [r,c] = size(img);
    [rk, ck] = size(kern);
    img_pad = SetPadding(img, kern, pad);
    op_img = img;
    if rk==1 && ck==2 %special case, horizontal derivative.
        for i = 2:r+1
            for j = 2:c+1
                op_img(i-1,j-1) = img_pad(i-1,j-1).*kern(1,1)+ img_pad(i-1,j).*kern(1,2) ;
            end
        end
    elseif rk==2 && ck==1 %special case vertical derivative.
        for i = 2:r+1
            for j = 2:c+1
                op_img(i-1,j-1) = img_pad(i-1,j-1).*kern(1,1)+ img_pad(i,j-1).*kern(2,1) ;
            end
        end
    elseif not(mod(rk,2) && mod(ck,2)) %even sized kernels (eg. 2x2, 4x4...)
        for i = 1:r
            for j = 1:c
                for k = 1:rk
                    for l = 1:ck
                        su(k,l) = img_pad(i+k-1,j+l-1).*kern(k,l);
                    end
                end
                op_img(i,j) = sum(sum(su));
            end
        end
    else %for odd sized kernels (3x3, 5x5...)
        for i = 2:r+1
            for j = 2:c+1
                for k = -floor(rk/2):floor(rk/2)
                    for l = -floor(ck/2):floor(ck/2)
                        su(k+2,l+2) = img_pad(i+k,j+l).*kern(k+2,l+2);
                    end
                end
                op_img(i-1,j-1) = sum(sum(su));
            end
        end
    end
end
```

Output Images.

The output images are saved in the project folder with appropriate names. All images for all the kernels with all the paddings are saved. Below attached are few examples. As we can see there is an improvement over zero padding in the other types of padding. Especially in case of box filter, we can't see the dark edges as we get in zero padding.

The values for output pixels can go to negative ranges, hence, a scaled output is also plotted to visualize the actual convolution operation in a better way.

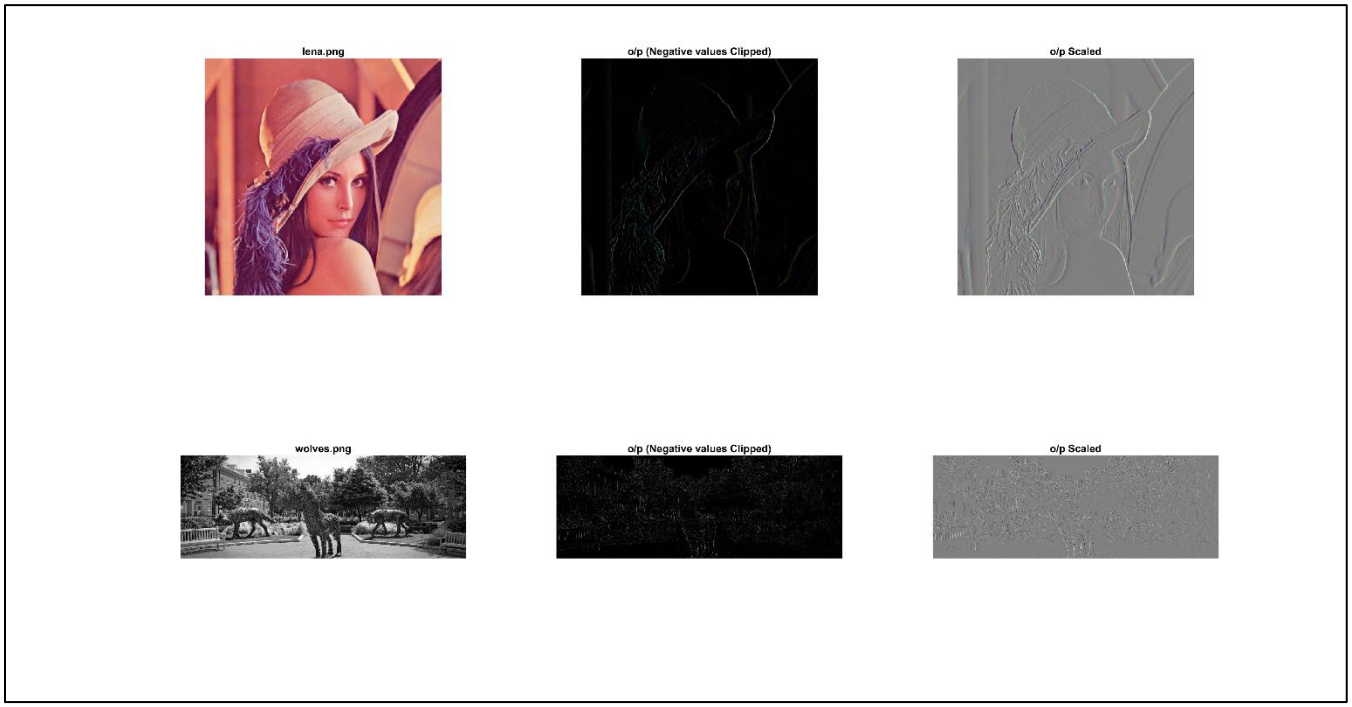
1. Box Filter– Zero Padding

2. Box Filter – Copy Edge

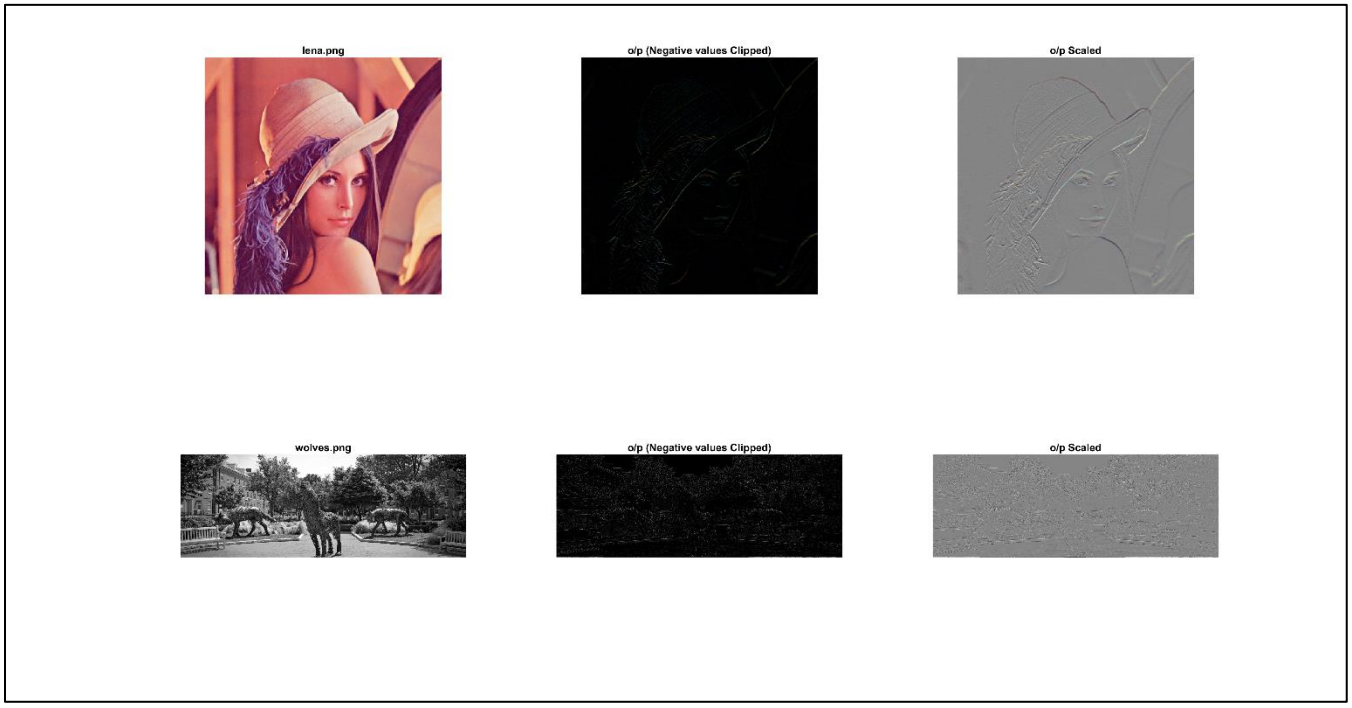
3. Box Filter – Wrap around

4. Box Filter – Reflect across edge

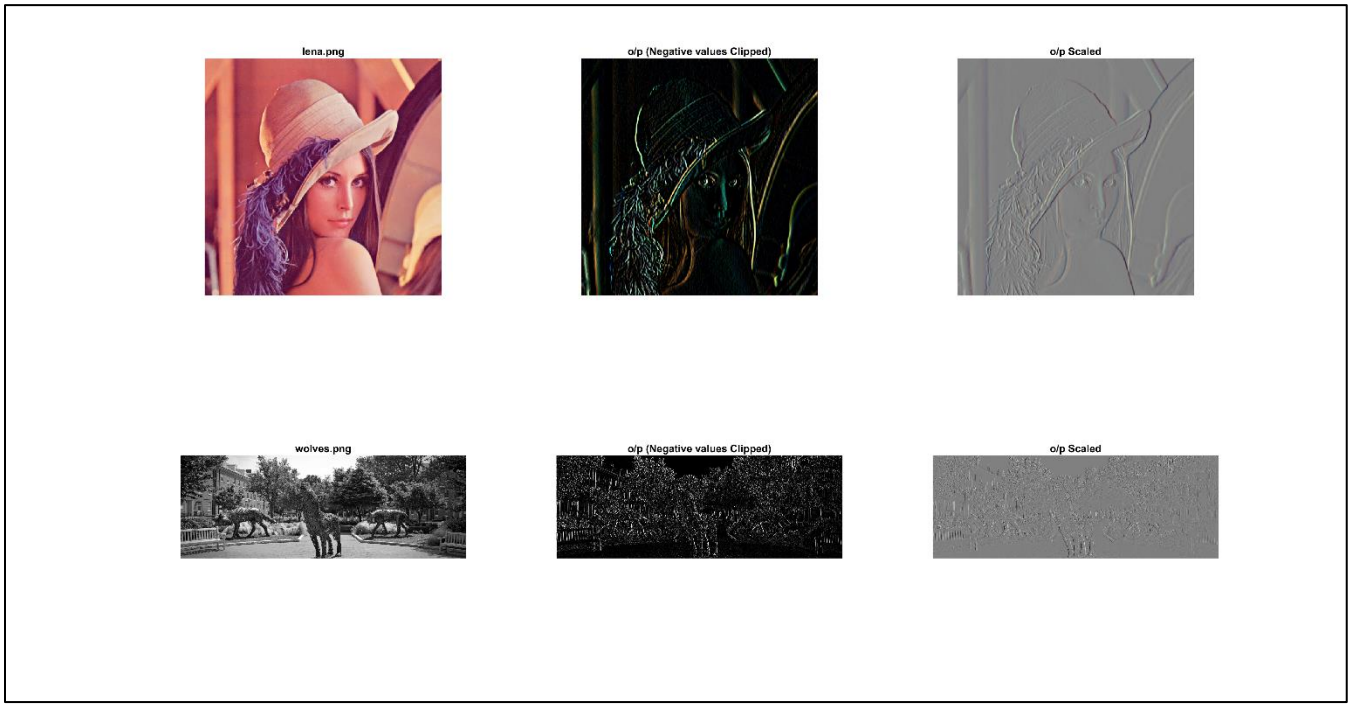
5. Derivative Horizontal – Copy Edge



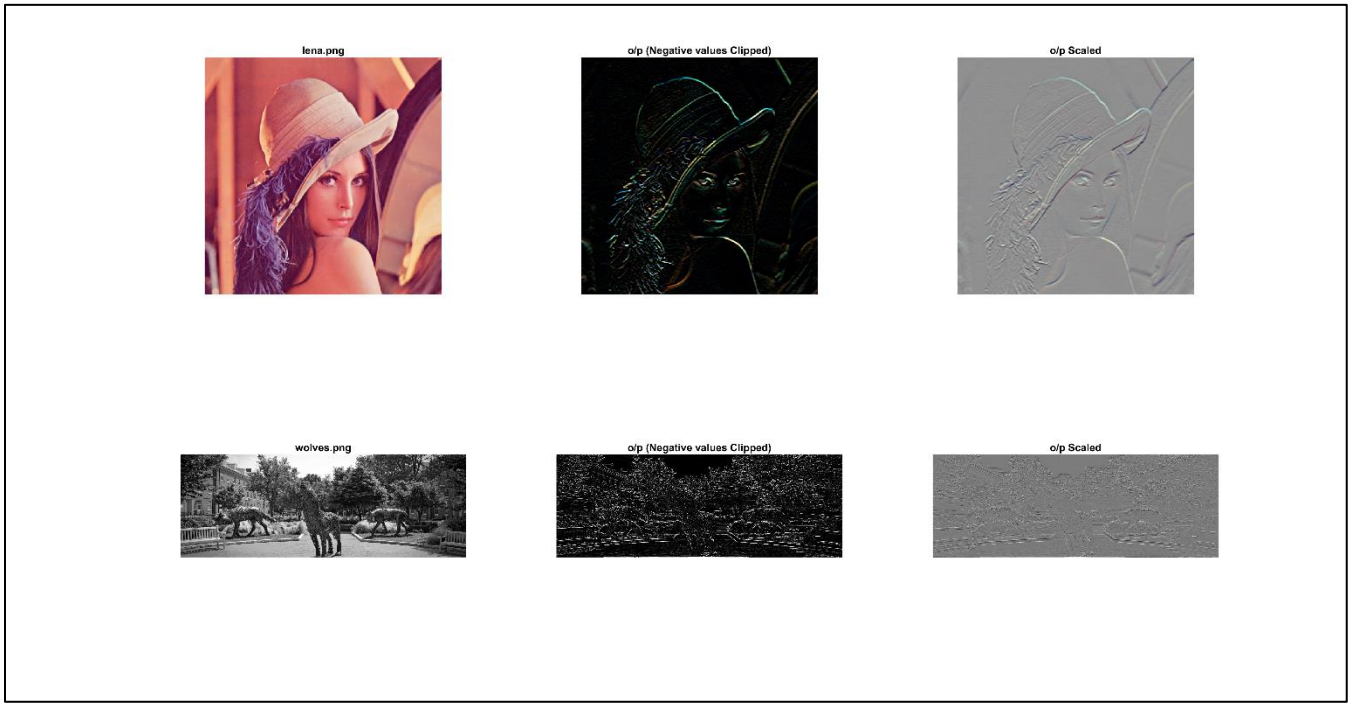
6. Derivative Vertical – Wrap around



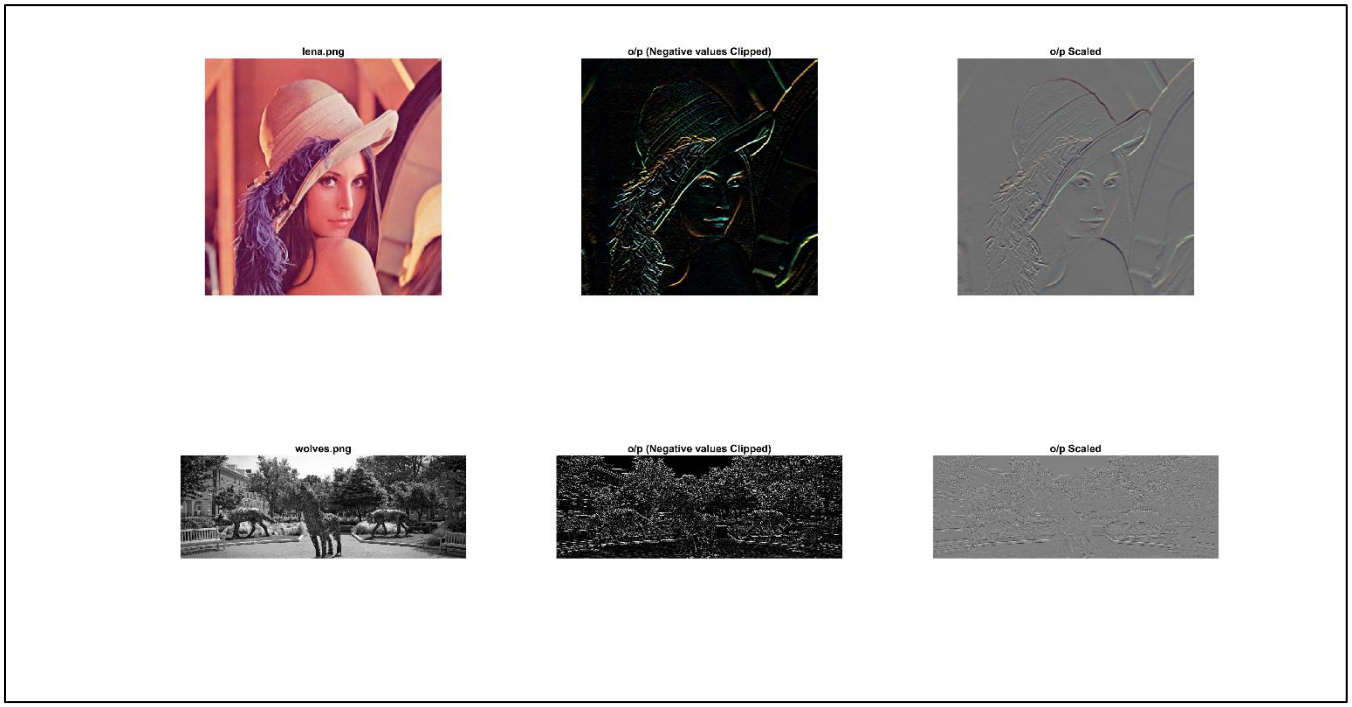
7. Prewitt Horizontal– Zero padding



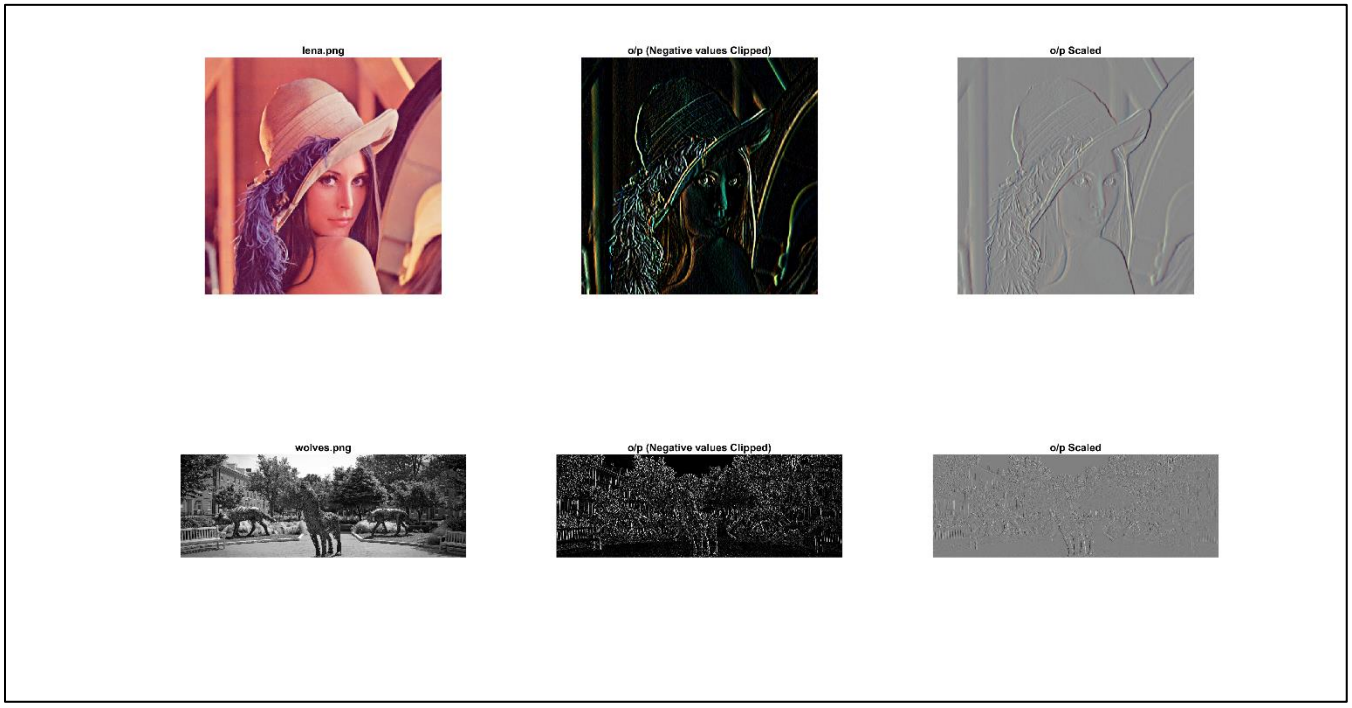
8. Prewitt Vertical – Wrap around



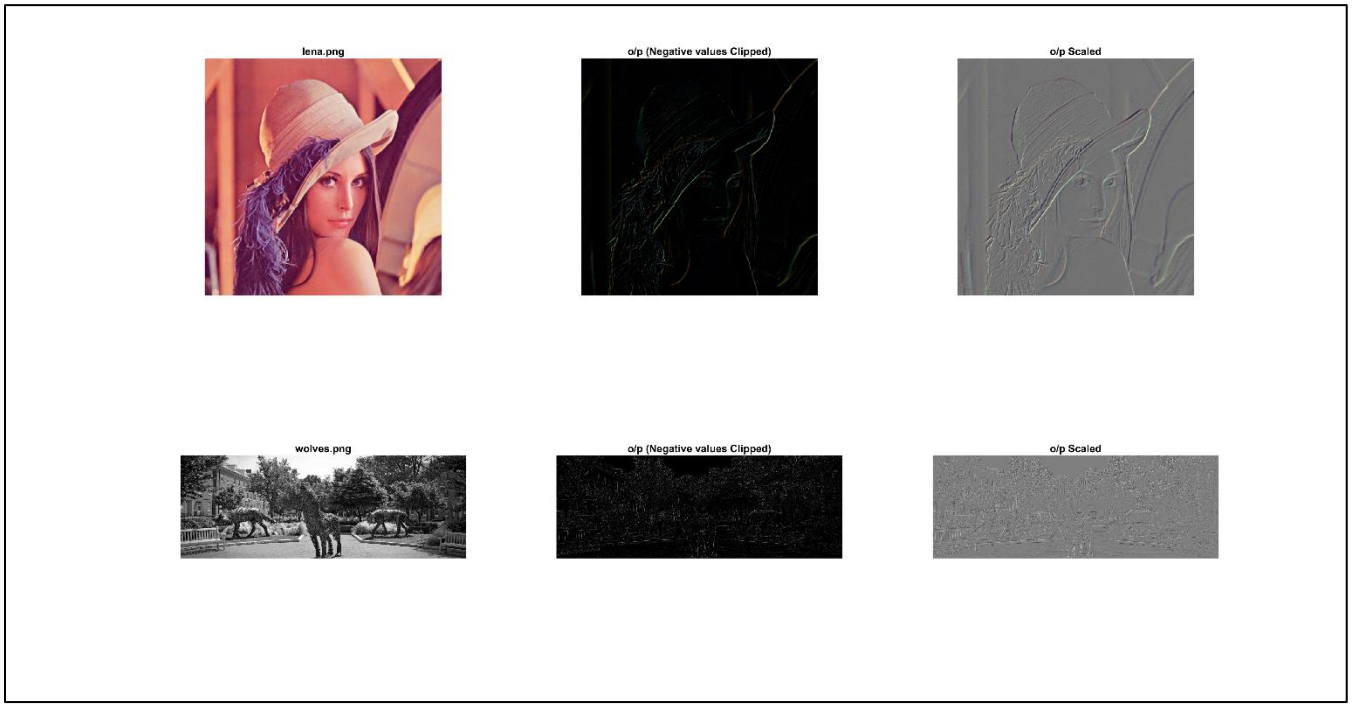
9. Sobel Vertical – Copy Edge



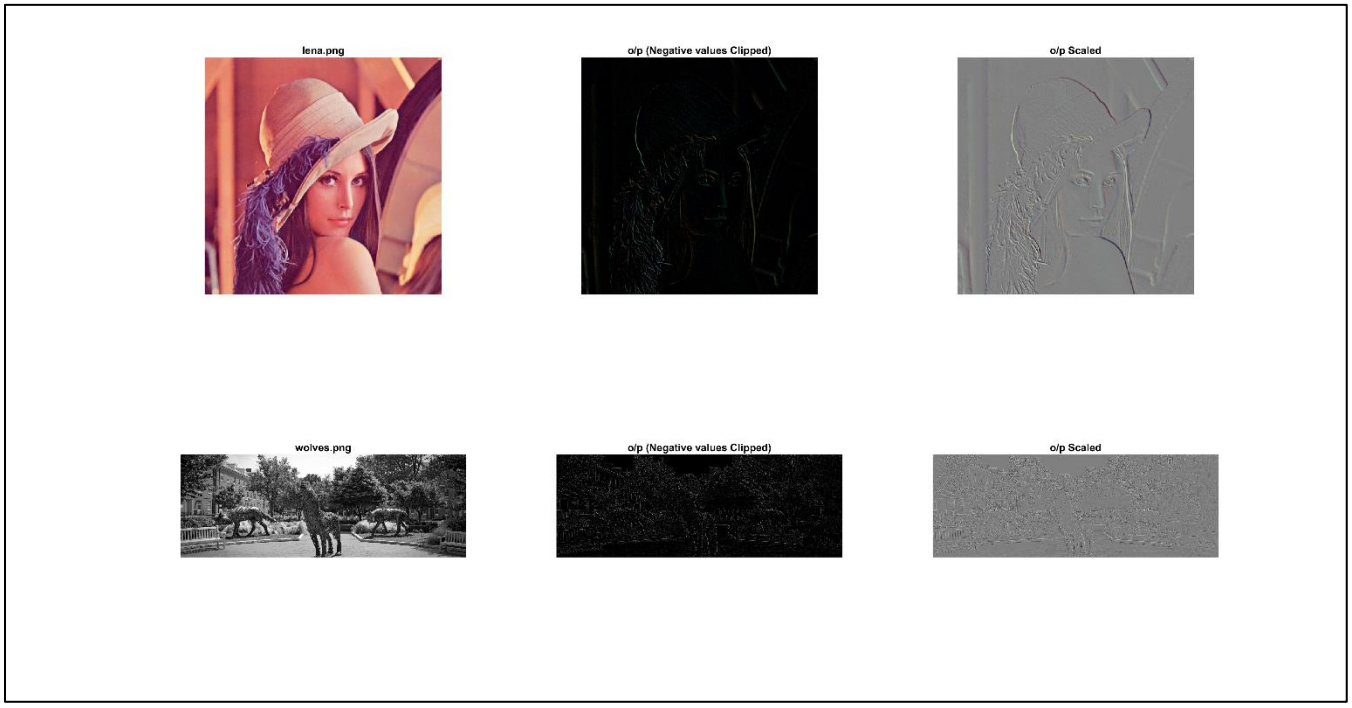
10. Sobel Horizontal – Reflect across edge



11. Roberts Vertical – Copy Edge



12. Roberts Horizontal – Reflect across edge



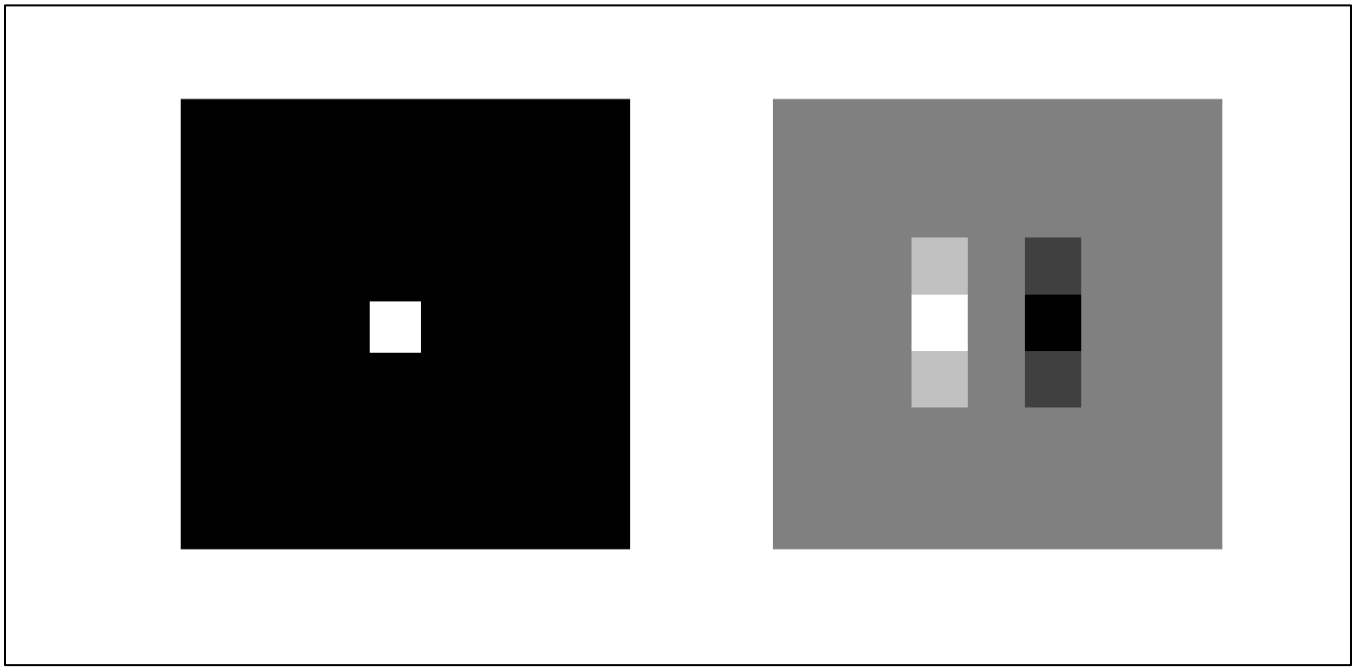
b) Confirmation of convolution results

Using an impulse function, we can confirm the results of our convolution function. Convolving a kernel with an impulse image will give a reflected version of the kernel itself. It is apparent if we zoom in to the center of the output image

Code to convolve the Unit impulse image with the Sobel filter.

```
%% part 2
% unit impulse function
%the unit impulse is defined with a matrix of zeros with a one in the
%center as shown below
uimp = zeros(1024);
uimp(512, 512) = 1;
ouimp = conv2d(uimp, kernels{6}, pad);
%show the results in the figure
figure;
subplot(1,2,1)
imshow(uimp)
subplot(1,2,2)
imshow(ouimp)
```

1. The output is as shown below



Bothe the images are zoomed in to get a better Idea of what's going on.
The left image is the input image with a Unit impulse centered at 512, 512. If we convolve this image with the Sobel filter ([-1, 0, 1;-2, 0, 2; -1, 0, 2]), we get the inverted kernel as we see on the right side. Hence, we can confirm that the convolution function is operating correctly on the images. In our code, we are not flipping the kernel before performing the operation, hence, this result is what we expect to get when we do convolution with the unit impulse image.

Problem 2

a) Implement 2D fft using 1D fft

```
%% Problem 2 2D fft
% Author - mmsardes Campus ID 200320514;
% Inputs - image.
% Outputs - Spectrum and Phase visualization, recovered image from fft of
% the original image
% The problem is to use 1d fft to get a 2d fft of an image. Visualize the
% spectrum
%%
clc; clear all; close all;
img = imread('wolves.png'); %read image - Results for wolves.png are saved in the folder
img = rgb2gray(img); %convert to grayscale
img = im2double(img); %convert to double precision
[r,c] = size(img); %take the size of the image

fftImg = dft2(img); %take fft using the dft2 function - (2d fft using 1d fft)
fftsImg = fftshift(fftImg); %shift fft to get centered fft for visualization purposes
%spectrum is root of sum of squares of real and imaginary parts of fft
spec = sqrt(real(fftsImg).^2+imag(fftsImg).^2);
% Apply log transformation for better visualization
spec = log(1+abs(spec));
% Normalize to 0-255 again to display using imshow
spec = uint8(floor((spec-min(spec(:))).*255./(max(spec(:))-min(spec(:)))));
% Phase is arctan(imaginary part of fft/real part of fft)
phase = atan(imag(fftsImg)/real(fftsImg));
% Apply log transformation for better visualization
phase = log(1+abs(phase));
% Normalize to 0-255 again to display using imshow
phase = uint8(floor((phase-min(phase(:))).*255./(max(phase(:))-min(phase(:)))));

%% Doing the same process using builtin fft function to verify results.
fft2Img = fft2(img);
fft2sImg = fftshift(fft2Img);
spec1 = sqrt(real(fft2sImg).^2+imag(fft2sImg).^2);
spec1 = log(1+abs(spec1));
spec1 = uint8(round((spec1-min(spec1(:))).*255./(max(spec1(:))-min(spec1(:)))));
phase1 = atan(imag(fft2sImg)/real(fft2sImg));
phase1 = log(1+abs(phase1));
phase1 = uint8(round((phase1-min(phase1(:))).*255./(max(phase1(:))-min(phase1(:)))));
```

Function - *dft2.m - used to calculate 2d fft of image using 1dfft*

```
function [fftImg] = dft2(img)
%DFT2 Summary of this function goes here
%   Detailed explanation goes here
    img = (img-min(img(:))./(max(img(:))-min(img(:)));
    %approach 1 - operate fft on first dimension first and then the second
    %using the syntax fft(x, [], dim)
    fftImg = fft(fft(img, [],1), [], 2);
%     Approach 2 - use 2 for loops, one to take fft of the rows. Then use
%     another one to take the fft of the obtained result from the first
%     loop
%     [r,c] = size(img);
%     for i = 1:r
%         fftImg(i,:) = fft(img(i,:));
%     end
```

```
%     for j = 1:c
%         fftImg(:,j) = fft(fftImg(:,j));
%     end
end
```

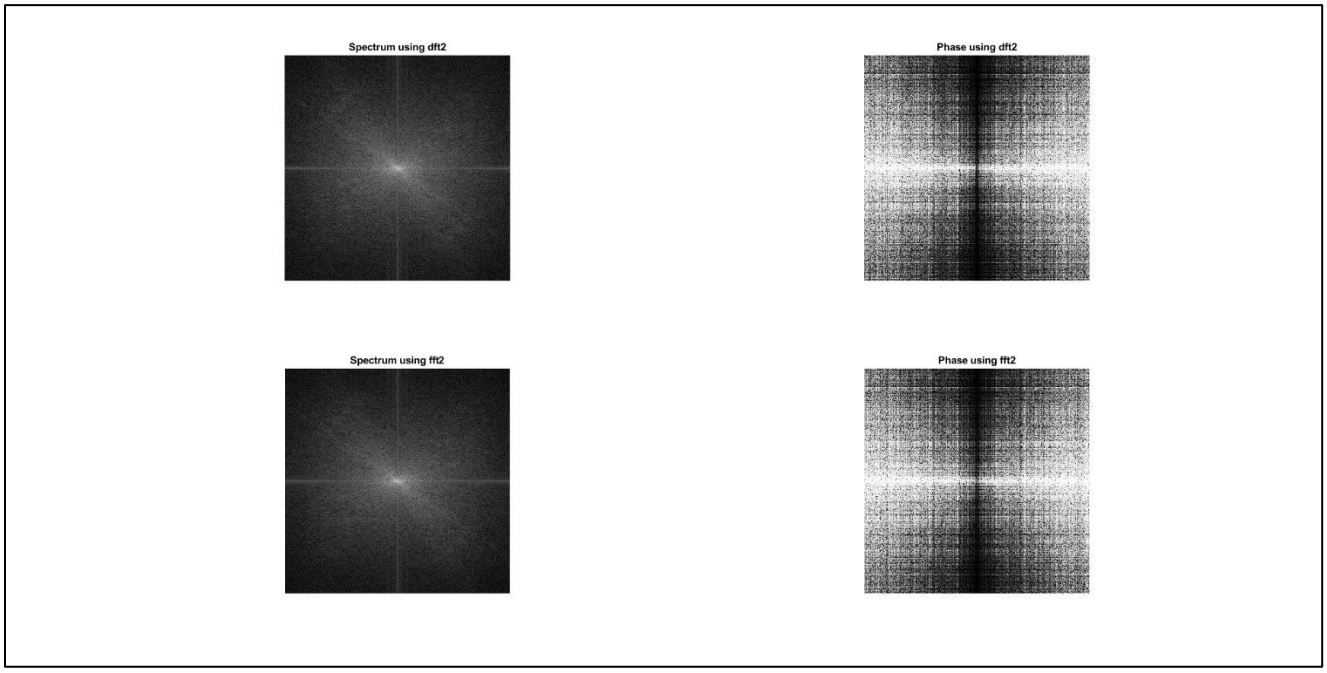
b) Use DFT2 to implement inverse FFT of input transform F

```
%% part 2b Recover original image from the fft obtained using dft2 function

%recover original image by operating dft on conjugate of the fft and then
%taking the conjugate of result and dividing by MN
recov = (1/numel(fftImg)).*real(conj(dft2(conj(fftImg))));
%scale recovered image from 0 -255 to get image compared to original image
recov = uint8(floor((recov-min(recov(:))).*255./(max(recov(:))-min(recov(:)))));
%scale original image from 0 -255 to get it in the original form (It was
%scaled from 0-1 for fft calculation purposes
img = uint8(floor((img-min(img(:))).*255./(max(img(:))-min(img(:)))));
% Get difference image
d = img - recov;

%% plotting the results

%plot spectra and phase using dft2 and fft2(built-in) functions
subplot(2,2,1)
imshow(spec)
title('Spectrum using dft2');
subplot(2,2,2)
imshow(phase)
title('Phase using dft2');
subplot(2,2,3)
imshow(spec1)
title('Spectrum using fft2');
subplot(2,2,4)
imshow(phase1)
title('Phase using fft2');
%plot original image, recovered image and difference image
figure;
subplot(3,1,1);
imshow(img);
title('Original Image');
subplot(3,1,2);
imshow(recov);
title('Recovered Image');
subplot(3,1,3);
imshow(d);
title('Difference Image');
```

Outputs:

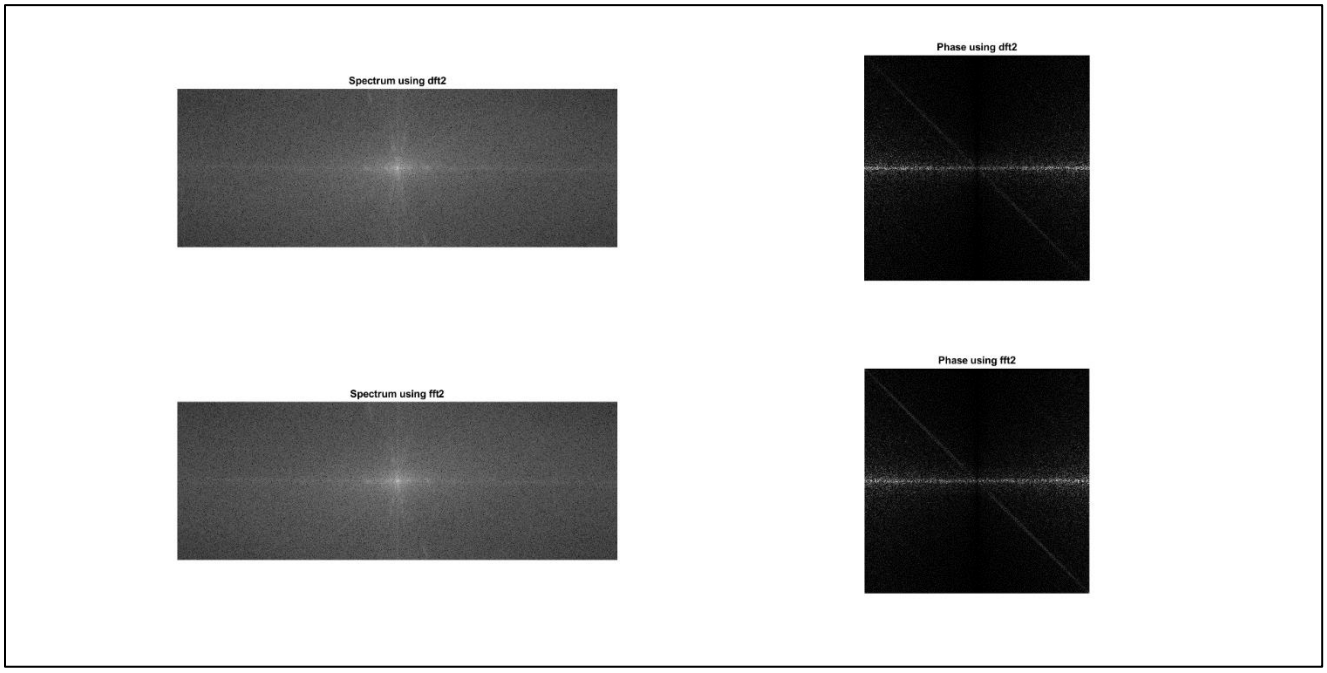
The phase and the spectra are plotted for both dft2 i.e. my implementation of the 2d fft using 1d fft function and also the fft2 function that is built-in MATLAB. fftshift is used to center the spectra and phase for a better visualization of them. The obtained fft is used to calculate idft of the image which is then compared with the original image and plotted along with the difference of the image which, as expected is a black image with all zeros. i.e. faithful recovery of Image to its original form.

1. Lena.png – Spectrum and Phase



2. Lena.png – Original and recovered images

3. Wolves.png – Spectrum and Phase



4. Wolves.png – Original and recovered images