**Problem 1(a):**

**my_conv2 function:**

For this problem I have written a function called ***my_conv2.m***

```
function [padded_f, res] = my_conv2(f, w, pad, shape)
```

It takes main three inputs -

- f: Grayscale or RGB image (H x W x 3 or H x W)
- w: 2D kernel
- pad: padding type can be of four types: "**clip**", "**wrap-around**", "**copy-edge**", "**reflect-edge**"

We can ignore the fourth input (shape) for this problem as we will always use shape = "same" to make sure that the output image is of same shape of input.

**Code:**

The complete code is as follows.

```
function [padded_f, res] = my_conv2(f, w, pad, shape)
  im_h = size(f,1); % image height
  im_w = size(f,2); % image width
  ker_h = size(w,1); % kernel height
  ker_w = size(w,2); % kernel width

  if shape=="same"
    % combination of ceil and floor ensures the below:
    % padding_left + padding_right = ker_w-1
    padding_left = ceil((ker_w-1)/2);
    padding_right = floor((ker_w-1)/2);
    padding_top = ceil((ker_h-1)/2);
    padding_bottom = floor((ker_h-1)/2);
  elseif shape=="full"
    padding_left = ker_w-1;
    padding_right = ker_w-1;
    padding_top = ker_h-1;
    padding_bottom = ker_h-1;
  elseif shape=="valid"
    padding_left = 0;
    padding_right = 0;
    padding_top = 0;
    padding_bottom = 0;
  else
    error("Invalid padding type provided")
```

```
    end

    % number of channels in input image
    layers = 1;
    if length(size(f))>2
        layers = size(f,3);
    end
    padded_f =  zeros(im_h + padding_top + padding_bottom,...
              im_w + padding_left + padding_right, layers);
    res = zeros(size(padded_f,1)-ker_h+1,size(padded_f,2)-ker_w+1,layers);

    for layer=1:layers
        fc = f(:,:,layer); % single channel image

        % placing the center image without padding
        padded_f(padding_top+1:padding_top+im_h, ...
              padding_left+1:padding_left+im_w, layer) = fc;
        if pad == "clip" % clip/zero padding
            % we are good
        elseif pad == "wrap-around"
            % left right top and bottom padding
            padded_f(padding_top+1:padding_top+im_h,1:padding_left,layer) = fc(1:im_h,end-
padding_left+1:end);
            padded_f(padding_top+1:padding_top+im_h,end-padding_right+1:end,layer) =
fc(1:im_h,1:padding_right);
            padded_f(1:padding_top,padding_left+1:padding_left+im_w,layer) = fc(end-
padding_top+1:end,1:im_w);
            padded_f(end-padding_bottom+1:end,padding_left+1:padding_left+im_w,layer) =
fc(1:padding_bottom,1:im_w);
            % four corners (left top, left bottom, right top, right bottom)
            padded_f(1:padding_top,1:padding_left,layer) = fc(end-padding_top+1:end,end-
padding_left+1:end);
            padded_f(end-padding_bottom+1:end,1:padding_left,layer) = fc(1:padding_bottom,end-
padding_left+1:end);
            padded_f(1:padding_top,end-padding_right+1:end,layer) = fc(end-
padding_top+1:end,1:padding_right);
            padded_f(end-padding_bottom+1:end,end-padding_right+1:end,layer) =
fc(1:padding_bottom,1:padding_right);
        elseif pad == "copy-edge"
            % left right top and bottom padding
            padded_f(padding_top+1:padding_top+im_h,1:padding_left,layer) =
repmat(fc(1:im_h,1),1,padding_left);
            padded_f(padding_top+1:padding_top+im_h,end-padding_right+1:end,layer) =
repmat(fc(1:im_h,end),1,padding_right);
```

```
        padded_f(1:padding_top,padding_left+1:padding_left+im_w,layer) =
repmat(fc(1,1:im_w),padding_top,1);
        padded_f(end-padding_bottom+1:end,padding_left+1:padding_left+im_w,layer) =
repmat(fc(end,1:im_w),padding_bottom,1);
        % four corners (left top, left bottom, right top, right bottom)
        padded_f(1:padding_top,1:padding_left,layer) = fc(1,1);
        padded_f(end-padding_bottom+1:end,1:padding_left,layer) = fc(end,1);
        padded_f(1:padding_top,end-padding_right+1:end,layer) = fc(1,end);
        padded_f(end-padding_bottom+1:end,end-padding_right+1:end,layer) = fc(end, end);
      elseif pad == "reflect-edge"
        % left right top and bottom padding
        padded_f(padding_top+1:padding_top+im_h,1:padding_left,layer) =
fliplr(fc(1:im_h,1:padding_left));
        padded_f(padding_top+1:padding_top+im_h,end-padding_right+1:end,layer) =
fliplr(fc(1:im_h,end-padding_right+1:end));
        padded_f(1:padding_top,padding_left+1:padding_left+im_w,layer) =
flip(fc(1:padding_top,1:im_w));
        padded_f(end-padding_bottom+1:end,padding_left+1:padding_left+im_w,layer) = flip(fc(end-
padding_bottom+1:end,1:im_w));
        % four corners (left top, left bottom, right top, right bottom)
        padded_f(1:padding_top,1:padding_left,layer) = fliplr(flip(fc(1:padding_top,1:padding_left)));
        padded_f(end-padding_bottom+1:end,1:padding_left,layer) = fliplr(flip(fc(end-
padding_bottom+1:end,1:padding_left)));
        padded_f(1:padding_top,end-padding_right+1:end,layer) = fliplr(flip(fc(1:padding_top,end-
padding_right+1:end)));
        padded_f(end-padding_bottom+1:end,end-padding_right+1:end,layer) = fliplr(flip(fc(end-
padding_bottom+1:end,end-padding_right+1:end)));
      else
        error("unsupported padding")
      end

      % convolution
      for i = 1:size(res,1) % output height
        for j = 1: size(res,2) % output length
          res(i,j,layer) = sum(w.*padded_f(i:i+ker_h-1,j:j+ker_w-1,layer),'all');
        end
      end
    end
end
```

**Testing the code for padding effects:**

I have tested the function to see the results of different padding within ***prob_1.m*** script

```
%% testing the padding types (1a)
kernel = ones(40,40)/1600; % big kernel (40X40) so that the paddings are visible

% grayscale image
im_gray = rgb2gray(imread("lena.png"));
test_all_paddings(im_gray, kernel, "gray");
% rgb image
im_rgb = imread("wolves.png");
test_all_paddings(im_rgb, kernel, "rgb");
```

It makes use of the function below:

```
function test_all_paddings(img, kernel, img_type)
   pads = ["clip", "wrap-around", "copy-edge", "reflect-edge"];
   for i=1:length(pads)
      [padded_f, res] = my_conv2(img, kernel, pads(i), "same");
      fgr = figure();
      subplot(1,2,1)
      imshow(uint8(padded_f))
      title(pads(i) + " padded image")
      imwrite(uint8(padded_f),
"output/prob_1/"+"padding_"+pads(i)+"_"+img_type+"_padded_img.png")
      subplot(1,2,2)
      imshow(uint8(res))
      title("conv2 (40 X 40 box filter)")
      imwrite(uint8(res),
"output/prob_1/"+"padding_"+pads(i)+"_"+img_type+"_conv2_result.png")
      saveas(fgr, "output/prob_1/padding_"+pads(i)+"_"+img_type+".png")

      sgtitle(pads(i)+" box filter")
   end
end
```

**Result of padding effects:**

**Clip (zero-padding):**

All of the images shown below (with full size) are also available in the “output” folder.

clip padded image

conv2 (40 X 40 box filter)

**clip padded image**

**conv2 result (40 x 40) filter**

**Wrap-around:**

wrap-around padded image

conv2 (40 X 40 box filter)

**wrap-around padded image**

**conv2 result (40 x 40) filter**

**Copy-edge:**

**copy-edge padded image**

**conv2 (40 X 40 box filter)**

**copy-edge padded image**

**conv2 result (40 x 40) filter**

**Reflect-edge:**

reflect-edge padded image

conv2 (40 X 40 box filter)

**reflect-edge padded image**

**conv2 result (40 x 40) filter**

**Testing the Code for different kernels:**

I have tested the convolution result for different kernels within ***prob_1.m*** script

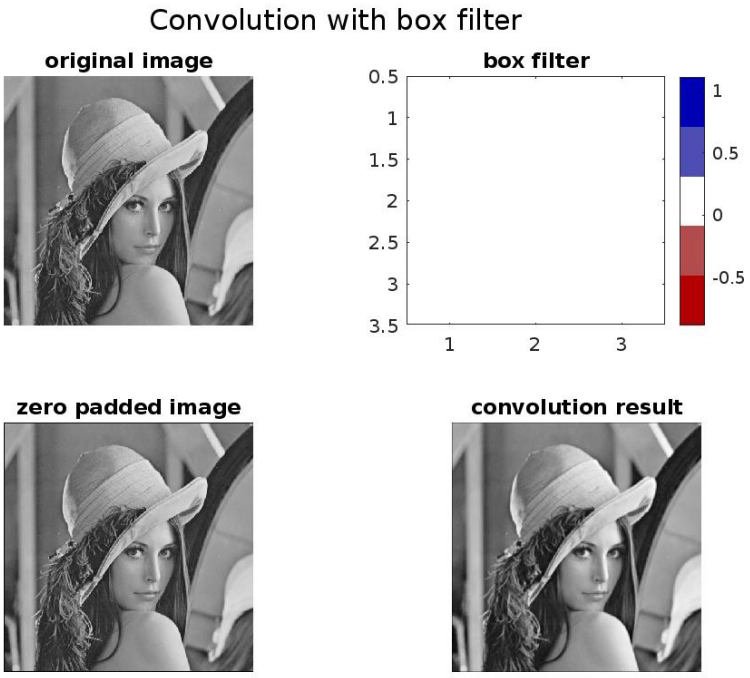
```
%% testing the kernels (1a)
% only clip (zero-padding) will be used for this section

w_box = [1,1,1; 1,1,1; 1,1,1]/9;
w_derx = [-1, 1];
w_dery = [-1; 1];
w_prewittx = [-1,0,1;-1,0,1;-1,0,1];
w_prewitty = flip(w_prewittx');
w_sobelx = [-1,0,1; -2,0,2; -1,0,1];
w_sobely = flip(w_sobelx');
w_robertsx = [0,1; -1,0];
w_robertsy = flip(w_robertsx');

test_kernel(im_gray, w_box, "box filter", "gray")
test_kernel(im_gray, w_derx, "der x", "gray")
test_kernel(im_gray, w_dery, "der y", "gray")
test_kernel(im_gray, w_prewittx, "prewitt x", "gray")
test_kernel(im_gray, w_prewitty, "prewitt y", "gray")
test_kernel(im_gray, w_sobelx, "sobel x", "gray")
test_kernel(im_gray, w_sobely, "sobel y", "gray")
test_kernel(im_gray, w_robertsx, "sobel x", "gray")
test_kernel(im_gray, w_sobely, "sobel y", "gray")

test_kernel(im_rgb, w_box, "box filter", "rgb")
test_kernel(im_rgb, w_derx, "der x", "rgb")
test_kernel(im_rgb, w_dery, "der y", "rgb")
test_kernel(im_rgb, w_prewittx, "prewitt x", "rgb")
test_kernel(im_rgb, w_prewitty, "prewitt y", "rgb")
test_kernel(im_rgb, w_sobelx, "sobel x", "rgb")
test_kernel(im_rgb, w_sobely, "sobel y", "rgb")
test_kernel(im_rgb, w_robertsx, "sobel x", "rgb")
test_kernel(im_rgb, w_sobely, "sobel y", "rgb")
```

I have used the function below for plotting the results (original image, kernel, padded image and convolution result) in a same figure.
For **plotting kernel** blue and red color representation has been used (see the 'cmap' variable below). **Blue** represents positive value (**+1/+2**) and **red** represents negative value (**-1/-2**). **Zero and near zero** values are represented by **white** color.

```
function test_kernel(img, kernel,kernel_name, img_type)
   [padded_f, res] = my_conv2(img, kernel, "clip", "same");
   fgr = figure();
   subplot(2,2,1)
   if img_type=="gray"
      colormap("gray");
   end
   imshow(img)
   title("original image")
   ax = subplot(2,2,2);
   imagesc(kernel)
   cmap = [0.7 0 0
           0.7 0.3 0.3
           1 1 1
           0.3 0.3 0.7
           0 0 0.7];
   title(kernel_name); colormap(ax, cmap); colorbar();
   subplot(2,2,3)
   imshow(uint8(padded_f))
   title("zero padded image")
   subplot(2,2,4);

   if img_type=="gray"
      %imagesc(res)
      disp_img = scale_image(res);
      colorbar();
   else
      disp_img = scale_rgb(res);
   end
   imshow(disp_img)
   title("convolution result");
   imwrite(uint8(res),
"output/prob_1/"+"kernel_conv_result_"+kernel_name+"_"+img_type+".png")

   sgtitle("Convolution with " + kernel_name)

   saveas(fgr, "output/prob_1/kernel_"+kernel_name+"_"+img_type+".png")
end
```

**Results for different kernels:**

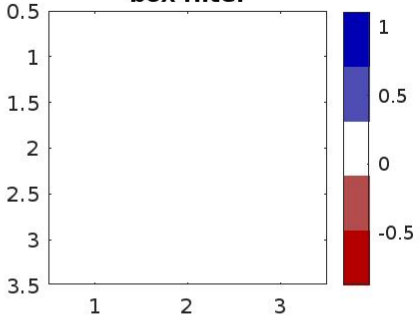
**Box filter (3X3):**




**Convolution result (zoomed)**

Convolution with box filter

**box filter**




**original image**

**zero padded image**

**convolution result**

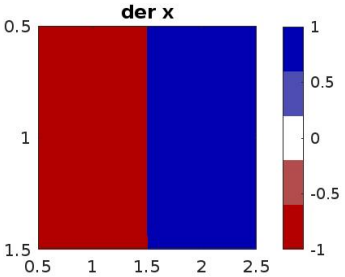
**Convolution result (zoomed)**

**First order derivative filter (X):**

Convolution with der x

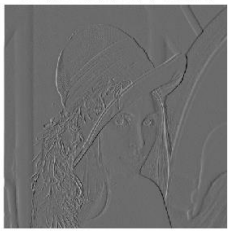
**original image**

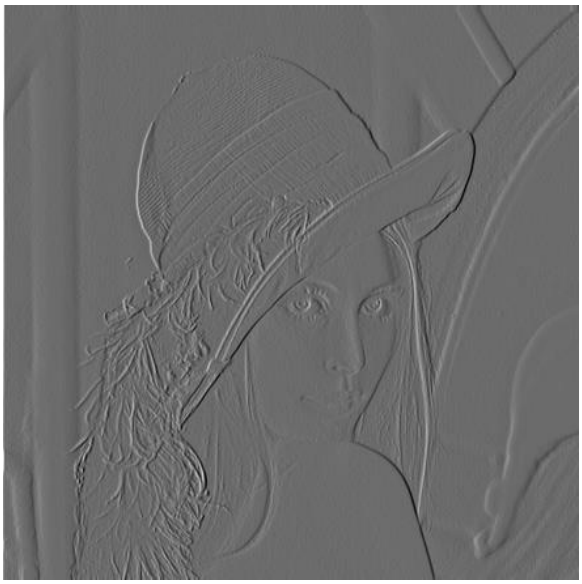
**der x**




**zero padded image**

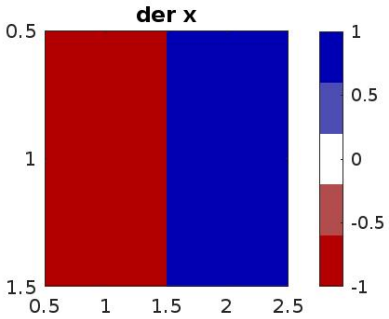
**convolution result**





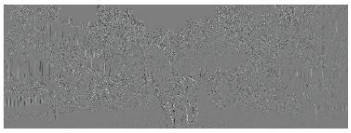
Convolution with der x

**original image**



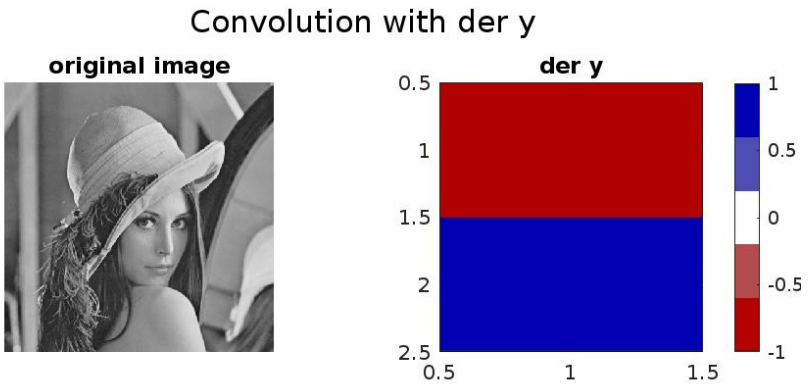

**zero padded image**

**convolution result**

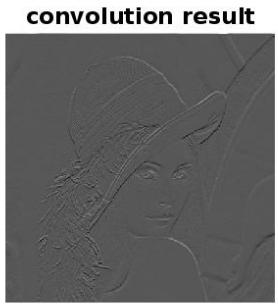


**Convolution result (zoomed)**

**First order derivative filter (Y):**



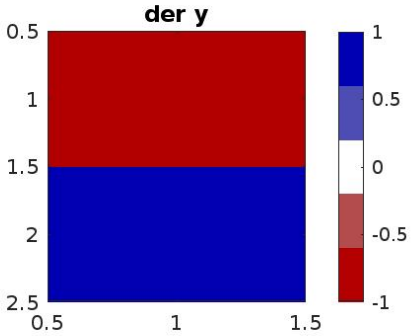





**Convolution result (zoomed)**

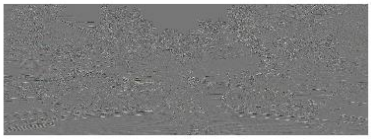
Convolution with der y

original image



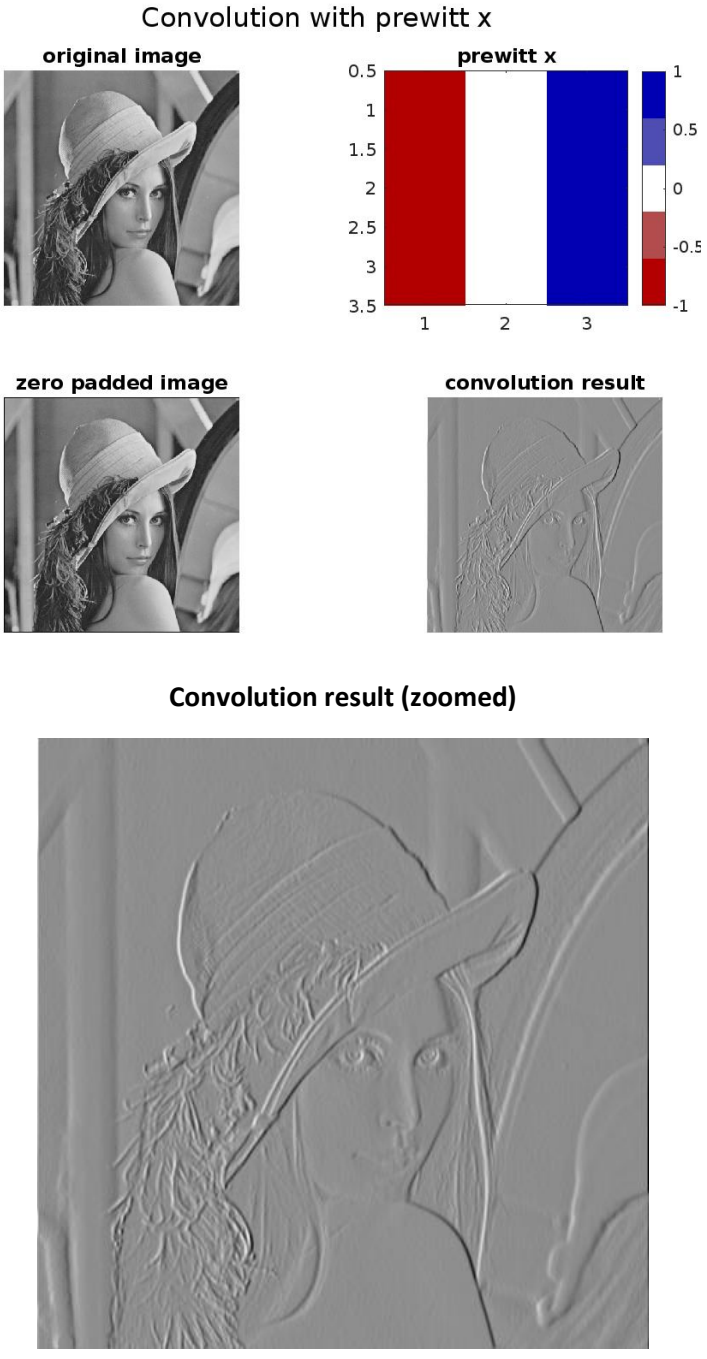

zero padded image

convolution result



**Convolution result (zoomed)**

**Prewitt filter (X):**

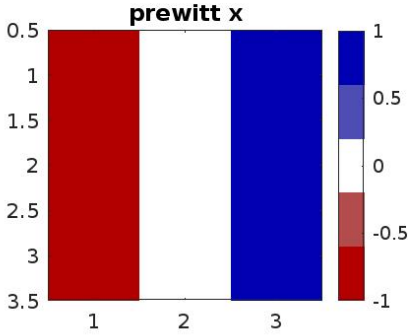



**Convolution result (zoomed)**

Convolution with prewitt x

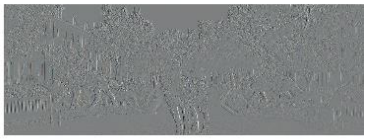
**original image**

**prewitt x**



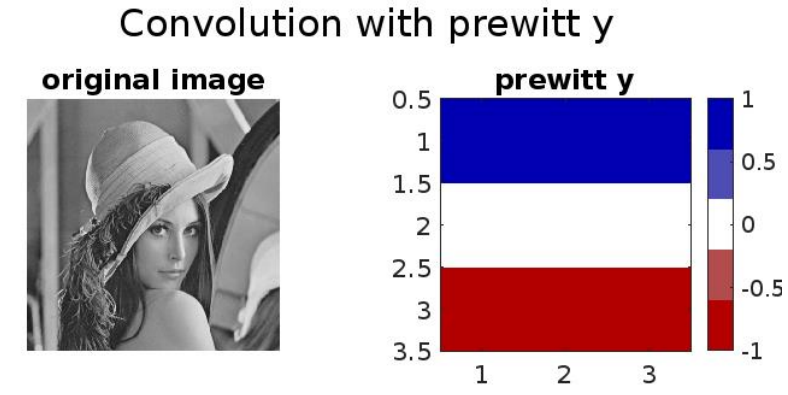

**zero padded image**

**convolution result**

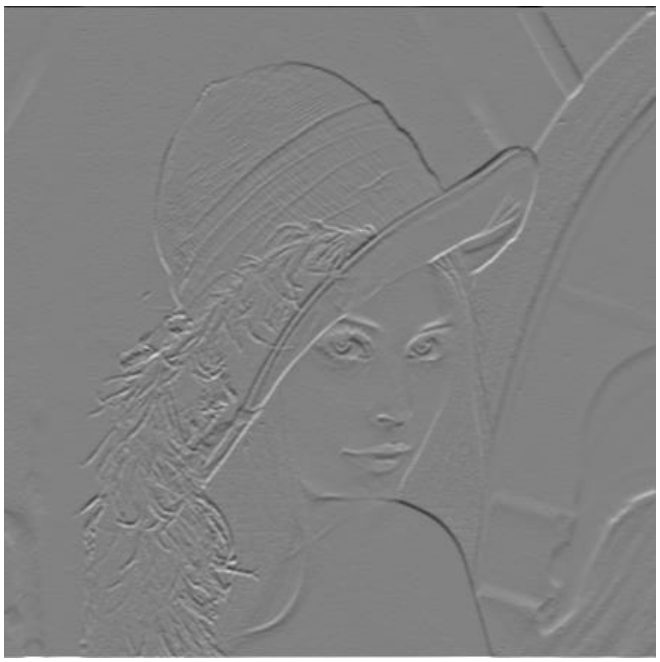


**Convolution result (zoomed)**

**Prewitt filter (Y):**

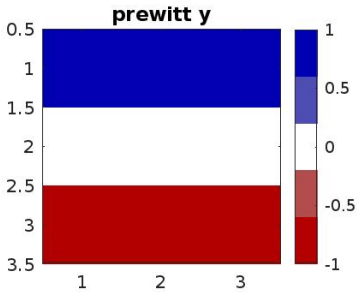



**Convolution result (zoomed)**



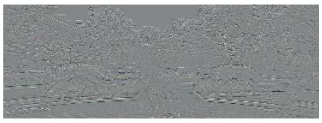
Convolution with prewitt y

original image

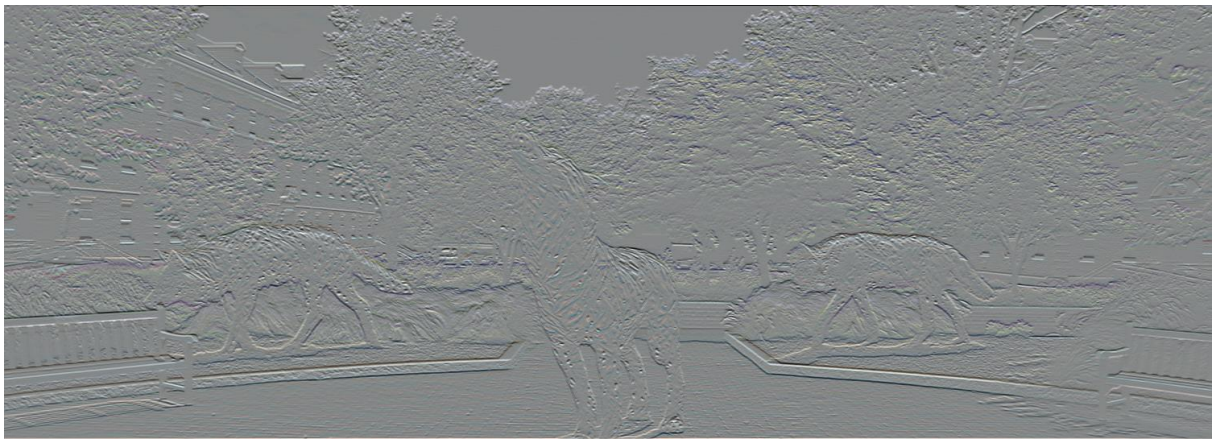



zero padded image

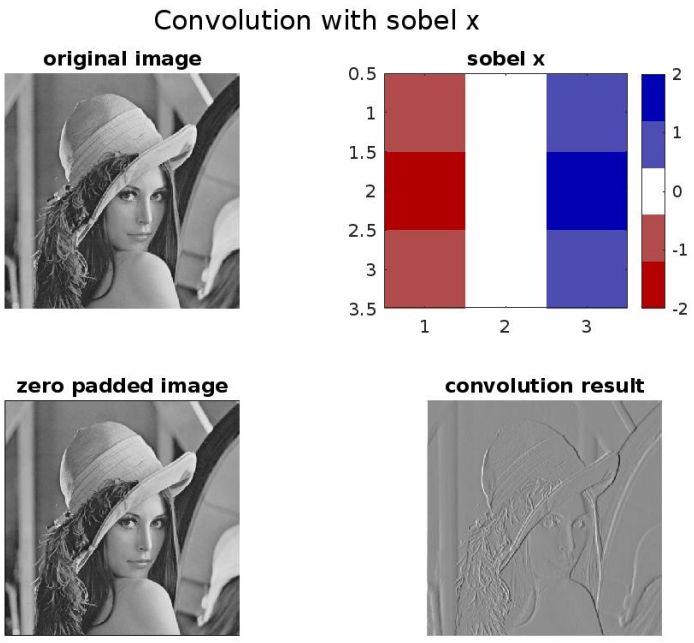
convolution result



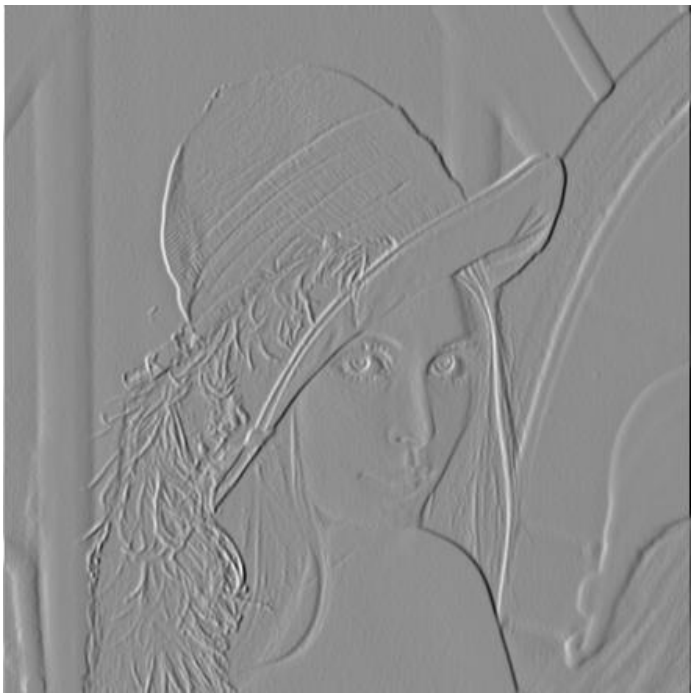
**Convolution result (zoomed)**

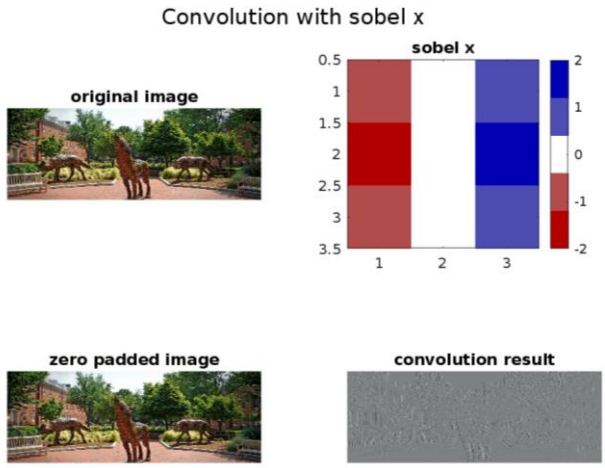


**Sobel filter (X):**



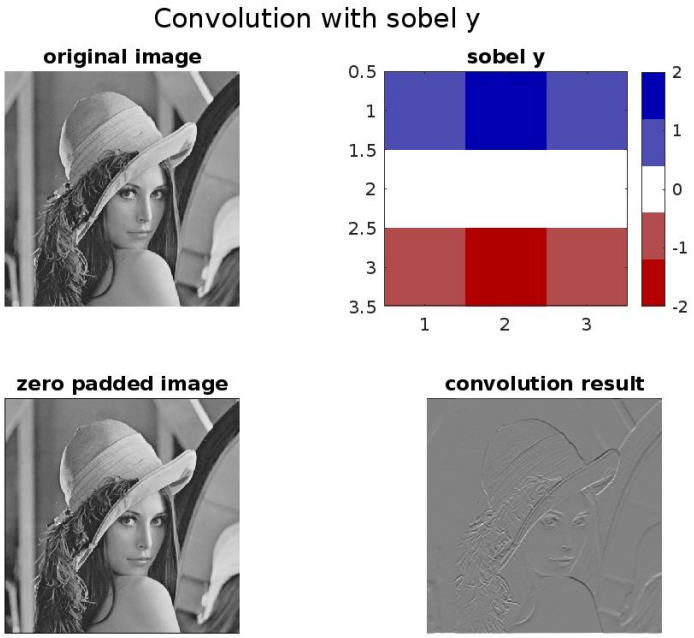

**Convolution result (zoomed)**



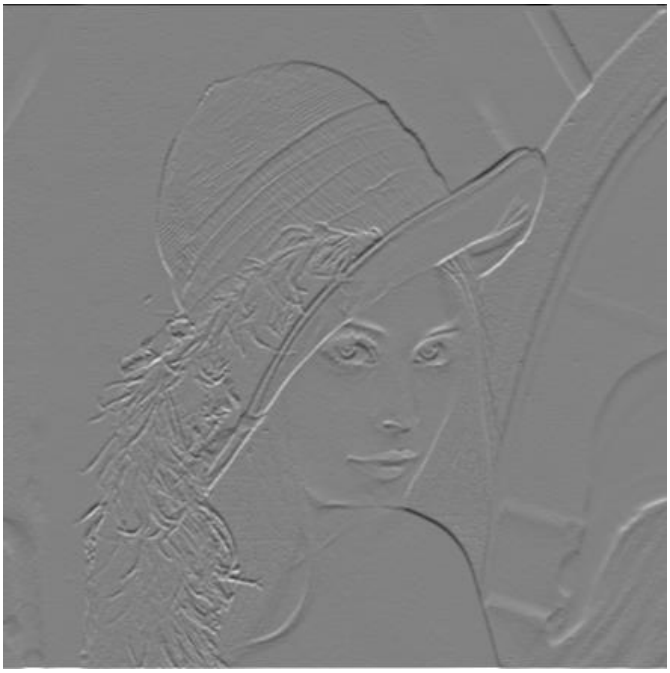



**Convolution result (zoomed)**

**Sobel filter (Y):**



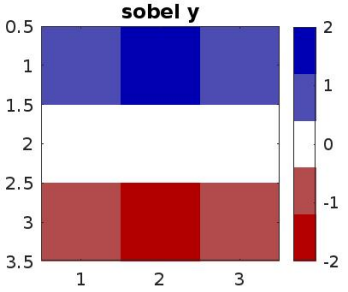

**Convolution result (zoomed)**



Convolution with sobel y

**original image**

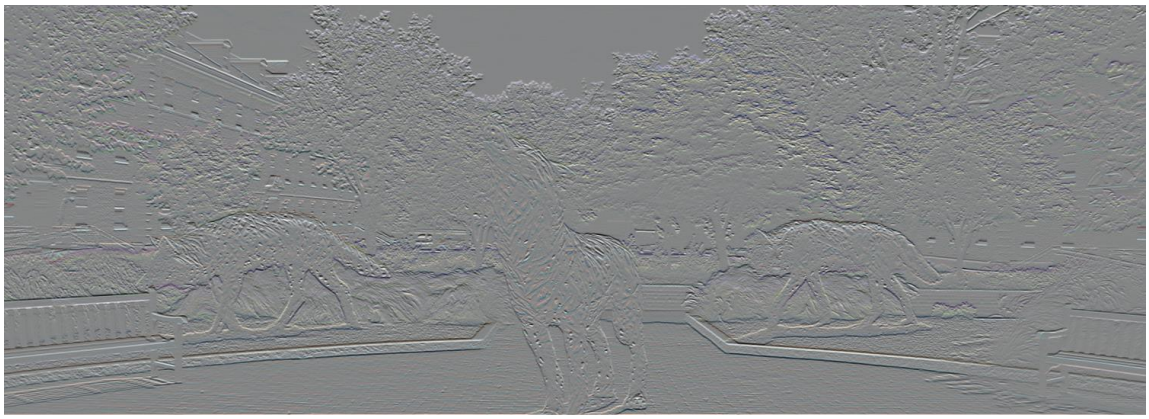
**sobel y**



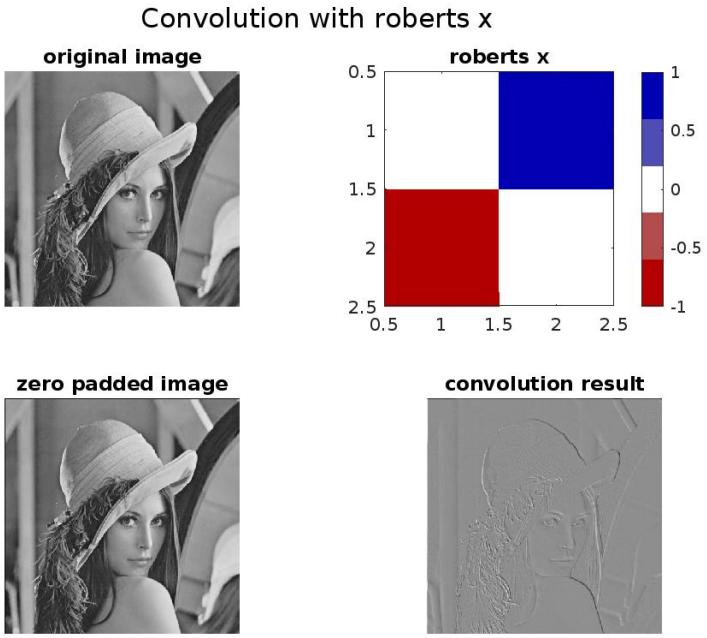

**zero padded image**

**convolution result**

**Convolution result (zoomed)**



**Roberts filter (X):**

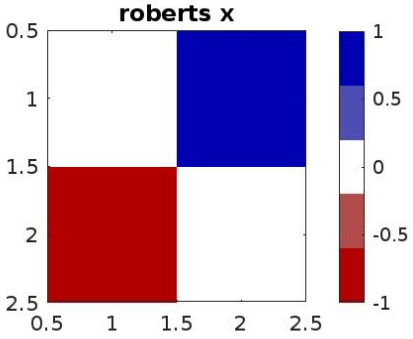



**Convolution result (zoomed)**

Convolution with roberts x

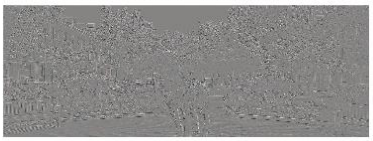
**original image**

**roberts x**

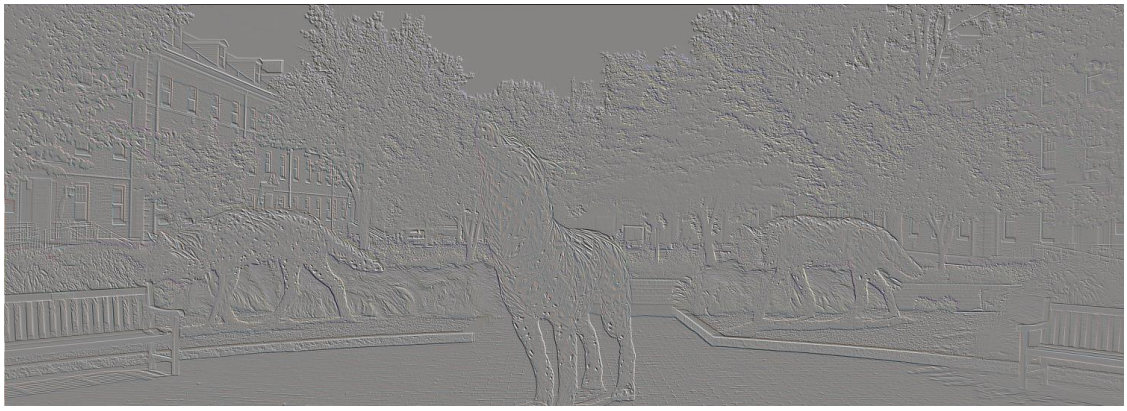



**zero padded image**

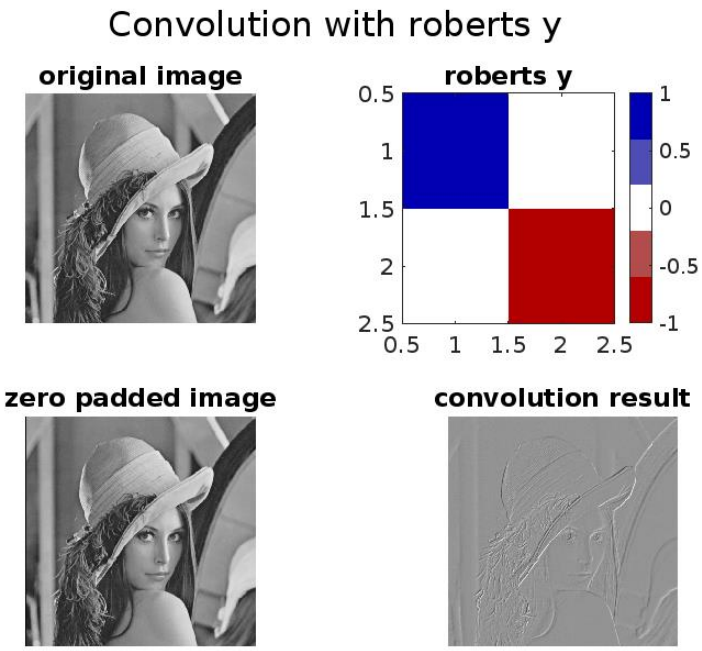
**convolution result**



**Convolution result (zoomed)**



**Roberts filter (Y):**

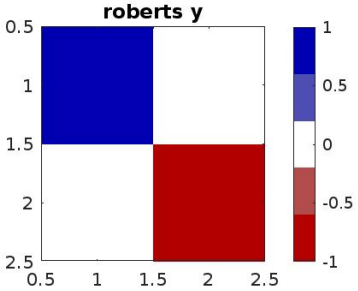



**Convolution result (zoomed)**

Convolution with roberts y

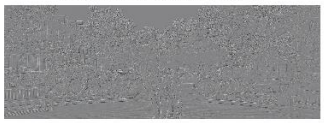
**original image**

**roberts y**

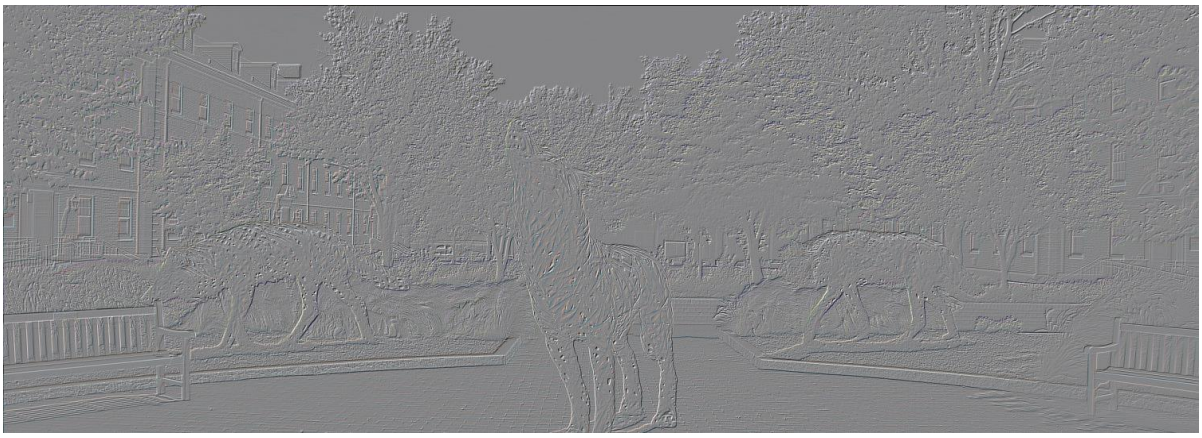



**zero padded image**

**convolution result**



**Convolution result (zoomed)**



**Problem 1(b):**

Creating an impulse image and viewing its convolution result (I picked sobel Y filter):

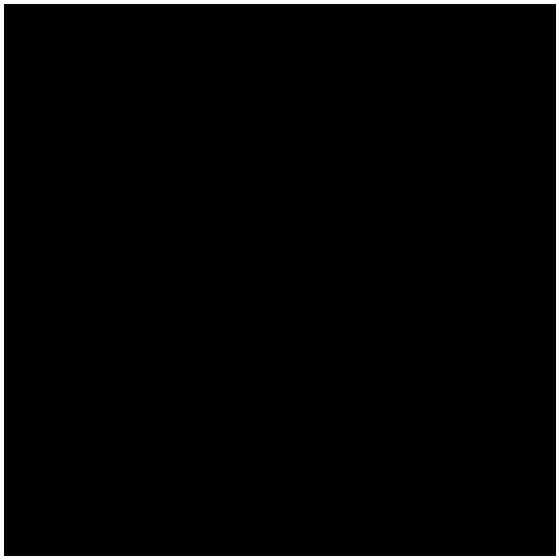
**Code:**

```
%% impulse image (1b)
impulse_im = uint8(zeros(1024,1024));
impulse_im(512,512) = 255;

figure()
imshow(impulse_im)
title('impulse image')
imwrite(impulse_im, "output/prob_1/impulse_image.png")

[padded_f, res] = my_conv2(impulse_im, w_sobely, "clip", "same");
fgr = figure();
imagesc(res)
colormap('jet'); colorbar;
grid on;
title('convolution result')
saveas(fgr, "output/prob_1/impulse_image_convolution.png")

% crop the middle 24X24 image for visibility
fgr = figure();
imagesc(res(501:524, 501:524))
colormap('jet'); colorbar;
```

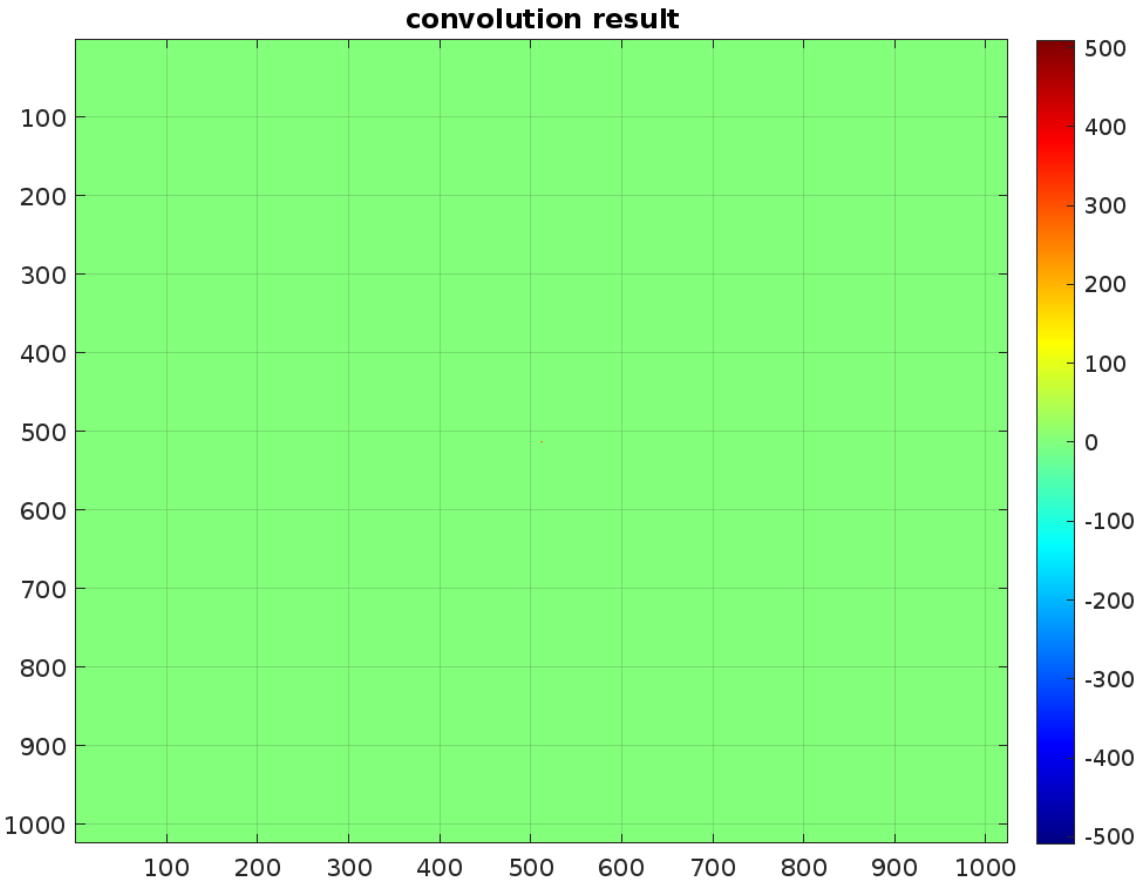
```
grid on;
title('convolution result')
saveas(fgr, "output/prob_1/impulse_image_convolution_cropped.png")
```

**Result:**

Impulse image

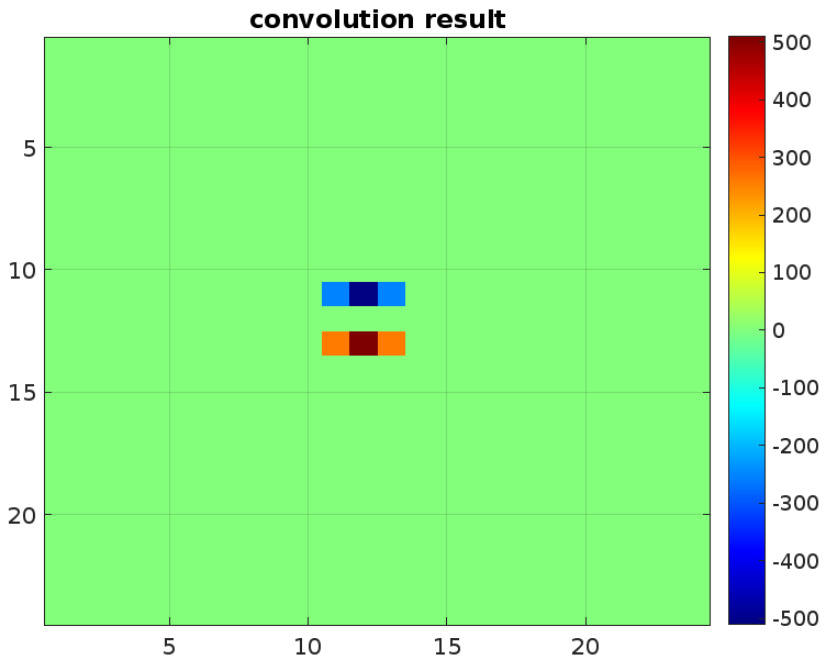


Convolution result




We can see a dot near (512, 512) above. To see more clearly, we plot only the sub-image of (501: 524, 501: 524).

Convolution result for (501: 524, 501: 524)




**Explanation:**

What we have got in the above convolution is that the kernel filter has appeared centering the pixel position (512,512) of the main image or (12,12) of the sub-image.

This is a property of convolution: when an impulse image is convoluted with a kernel we get the kernel as the output. Mathematically, $\delta(x,y) * k(x,y) = k(x,y)$

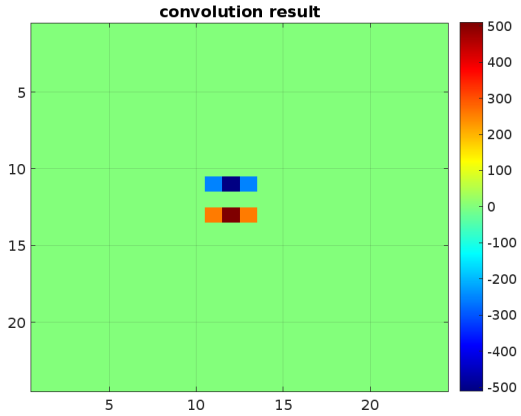
Here k is the sobel kernel and our input image has impulse value of 255.

Sobel (y) = [1,2,1; 0,0,0; -1,-2,-1];

So, in our convolution result we have got [255,510,255; 0,0,0; -255,-510,-255];

If we use conv2 function instead of my_conv2, we get the same result.

Convolution result using built-in conv2 function




That's the proof that my_conv2 is indeed performing convolution.

**Problem 2(a):**

Implementing DTFT2 from scratch using built-in 1D FFT:

Code:

```
function F = my_fft2(f)
   % input f 2D image of size (M,N)
   M = size(f,1);
   N = size(f,2);
   % output F is fft2 of f of same size
   F = zeros(M,N);

   for i=1:N % for each column
      F(:,i) = fft(f(:,i)); % complexity M*log(M)
   end
   for i=1:M % for each row
      F(i,:) = fft(F(i,:)); % complexity M*log(M)
   end
end
```

Testing the image with provided lena.png and wolves.png:

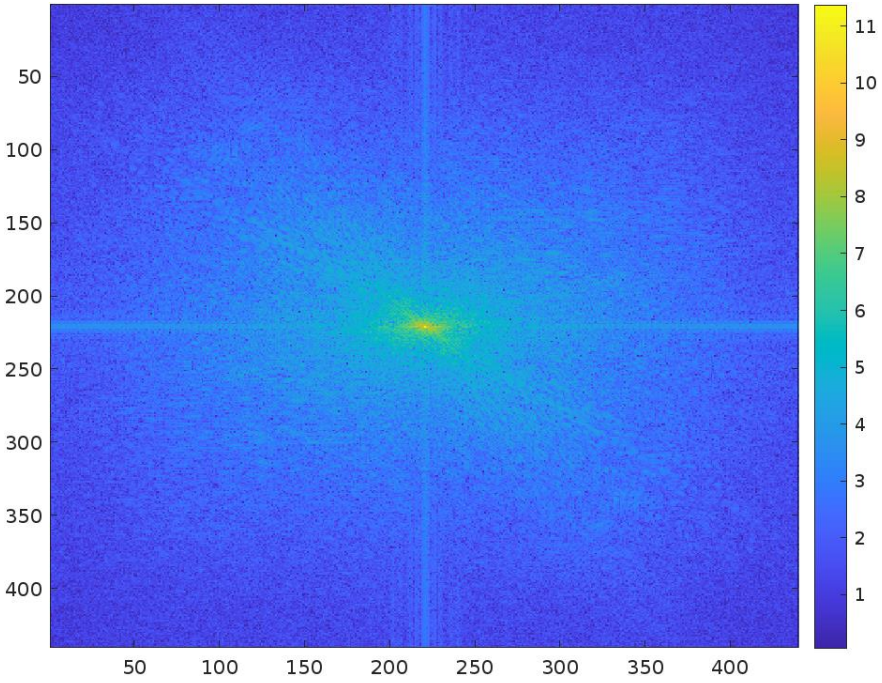
Code:

```
%% read
im1 = imread("lena.png");
im2 = imread("wolves.png");

im3 = rgb2gray(im1);
im4 = rgb2gray(im2);

scaled_img = scale_image(im3);

figure()
imshow(scaled_img)
colorbar()

%% my function
F = my_fft2(scaled_img);
% fft (magnitude)
figure()
imagesc(log(1+abs(fftshift(F))))
colorbar();
% fft (phase)
figure()
imagesc(angle(fftshift(F)))
colorbar();
```

**Result:**

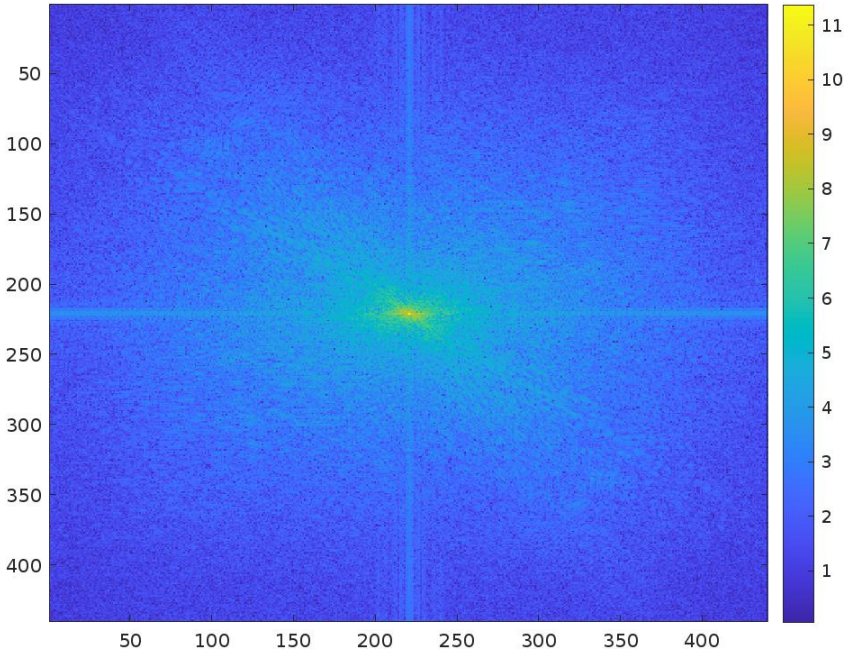
**Magnitude:** Using log(1+abs(fftshift(F))) formula

Magnitude of fft2 using my_fft2 function (lena.png)



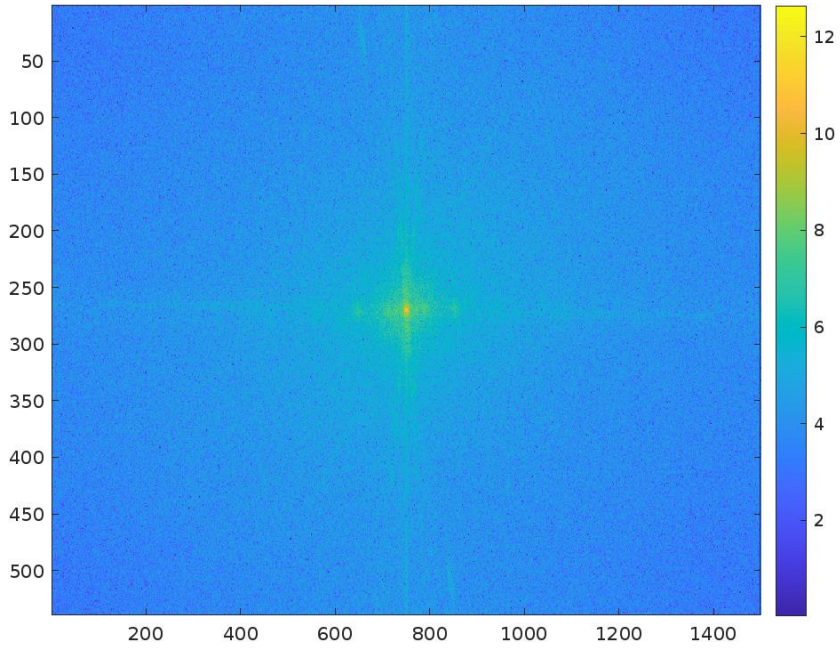

For comparison, if I use builtin fft2 function in Matlab, I get the following result:

Magnitude of fft2 using fft2 function (lena.png)



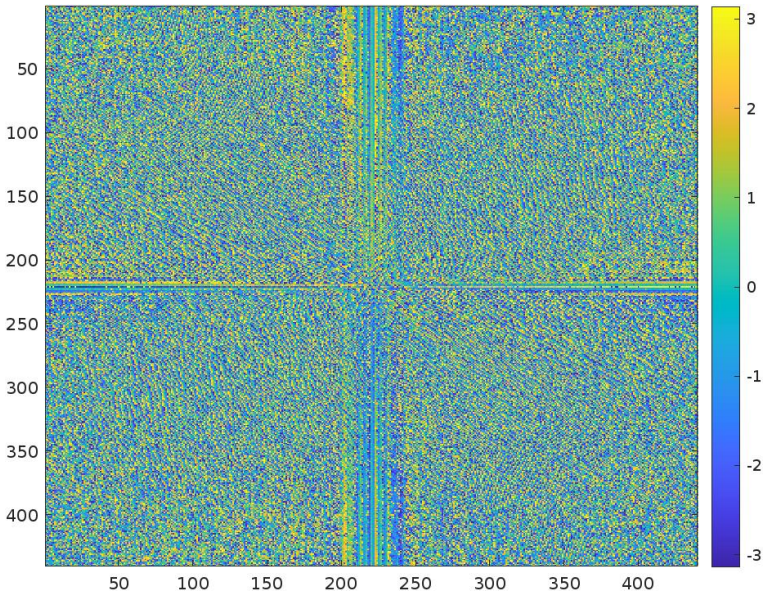

The above two results show that, my_fft2 function is valid.

Magnitude of fft2 using my_fft2 function (wolves.png)

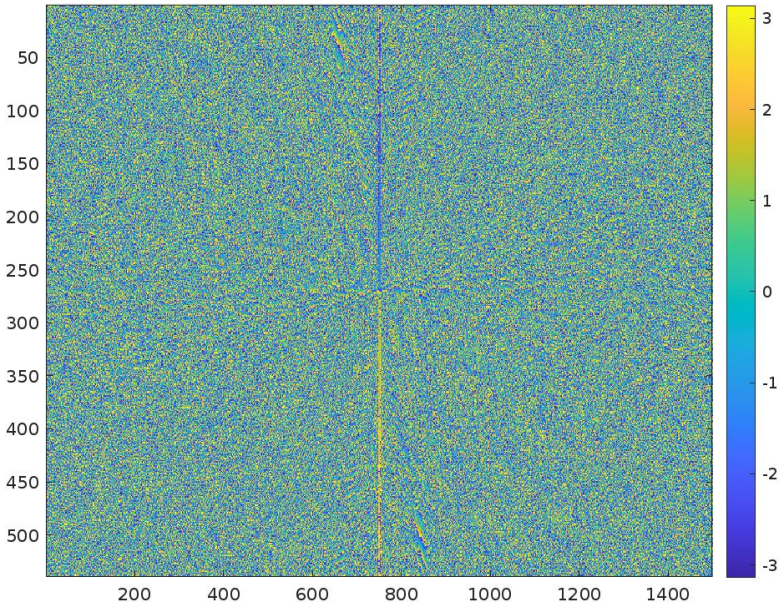



**Angle:** Using log(angle(fftshift(F))) formula

Angle of fft2 using my_fft2 function (lena.png)



Angle of fft2 using my_fft2 function (wolves.png)




**Problem 2(b):**

Implementation of ifft2 using my_fft2 function:

Code:

```
function f = my_ifft2(F)
   % input F 2D FFT of size (M,N)
   M = size(F,1);
   N = size(F,2);
   % output f is ifft2 of f of same size

   f = conj(my_fft2(conj(F)))/(M*N);
end
```

Then using this function to get the inverse fft of my_fft(grayscaled “wolves.png” or “lena.png”)

```
% inverse fft
I = my_ifft2(F);
figure()
% recovered image
imshow(abs(I));
colorbar();
```

**Result:**

Recovered image by my_iift(my_fft(“lena”))


Recovered image by my_iift(my_fft(“wolves”))


**Difference image:**

We want to check if the recovered image is exactly the original image or not.

Code:

```
% distance from original
figure()
imshow(abs(abs(I)-scaled_img))
colorbar();
```

Result:

Difference image for “lena”


Difference image for “wolves”


Final comment: We can use fast implementation of 1D DFT to implement 2D DFT (FFT2) using separability of 2D DFT equation. Again, we can implement IDFT2 using DFT2 block using conjugate property of complex numbers. If we implement correctly, the result of DFT and IDFT will nullify each other and we will get the original image.