CS 696 - Applied Computer Vision

Problem Set 1 : Filtering in Frequency Domain

INTRODUCTION:

We can perform linear spatial filters as a simple component-wise multiplication in the frequency domain.This suggests that we could use Fourier transforms to speed up spatial filters. This only works for large images that are correctly padded, where multiple transformations are applied in the frequency domain before moving back to the spatial domain.

BASIC STEPS IN FREQUENCY DOMAIN FILTERING :

Image filtering in spatial domain can be time consuming, because it involves sequentially operating on each pixel via for-loop. Faster results can be obtained using filtering in the frequency domain. The key idea is to :

Step 1: Convert both the image, and filter in the frequency domain using Matlab function (fft2).
Step 2: The result can be obtained by simply multiplying the image and the filter.
Step 3: Finally, the filtered image can be obtained by taking the inverse of the fft i.e. ifft. The core idea is shown in the code below.

Lowpass Filter :

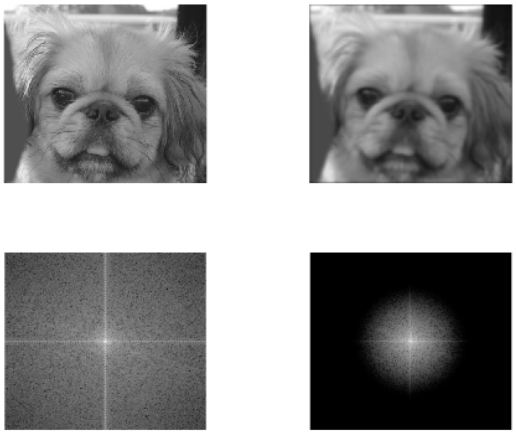
- Create a blurred image by attenuating the high frequencies and leave the low frequencies of the Fourier transform relatively unchanged.

Highpass Filter:

- Creates a sharpened image by attenuating the low frequencies and leave the high frequencies of the Fourier transform relatively unchanged

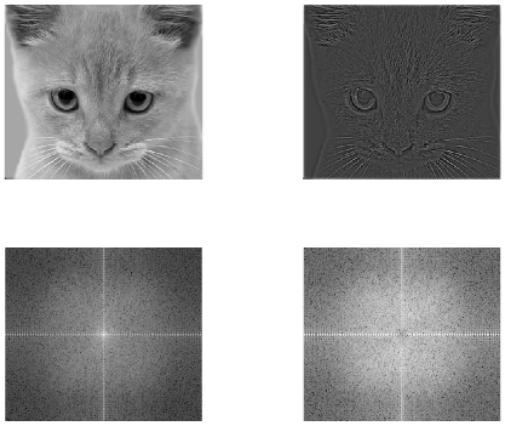
RESULTS:

(1) **D0 of 7%** the width of the Fourier transform (D0 = 0.07*PQ(1))



a)**Lowpass filtering of 'dog.bmp'**

Figure 1. *image filtering in frequency domain. top-left: input image; top-right: smoothed images (low-pass); bottom-left: original frequency spectrum; bottom-right: low-passed frequency spectrum.*



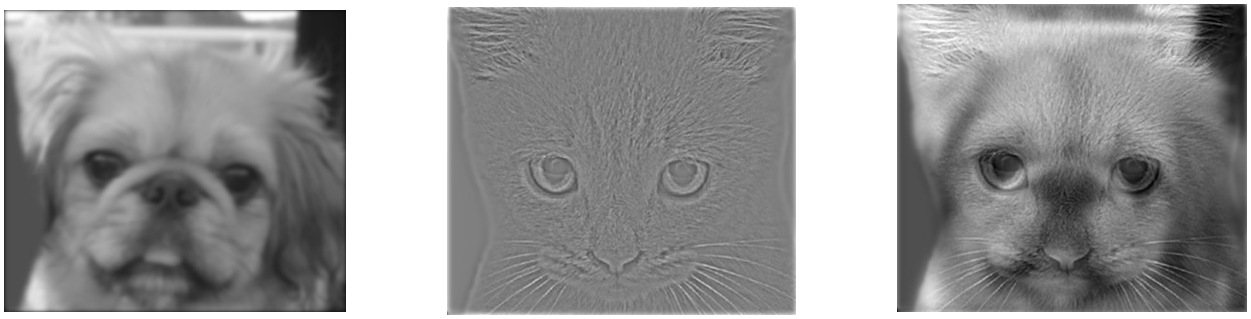
***b)*Highpass filtering of 'cat.bmp'**

Figure 2. *image filtering in frequency domain. top-left: input image; top-right: sharpened images (high-pass); bottom-left: original frequency spectrum; bottom-right: high-passed frequency spectrum.*

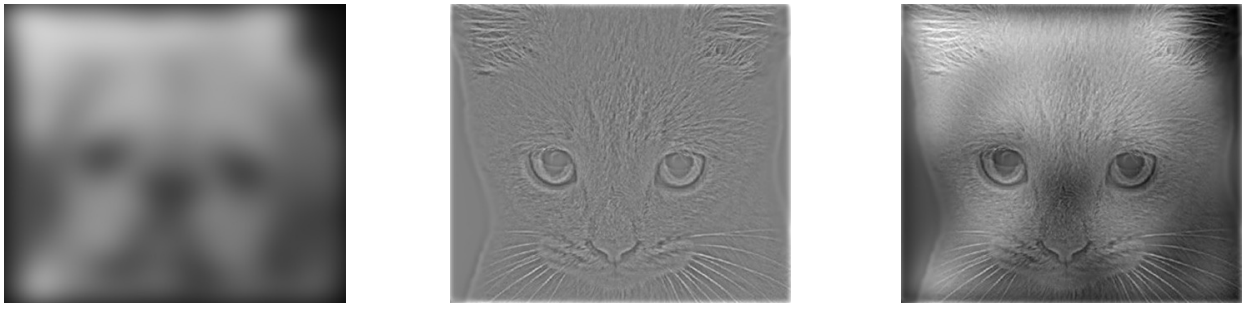
COMPARISON BETWEEN DIFFERENT D0 (CUT OFF FREQUENCIES) :

Low pass Image(left) High pass Image(center) HybridImage(right)

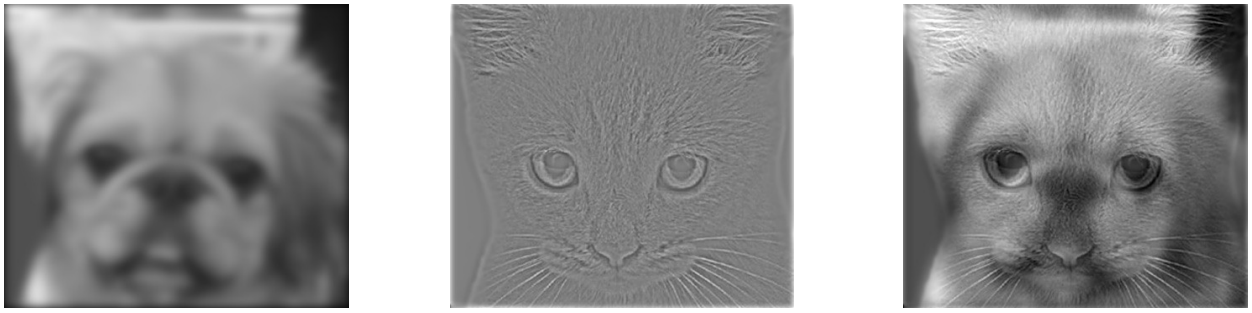
(1) D0 = 0.07*PQ(1); -> Decent hybrid image



(2) D0 = 0.01*PQ(1); -> Poor hybrid image : Dominated by the high pass filtered image



(3) D0 = 0.03*PQ(1); -> balanced hybrid image



CODE SNIPPETS:

Lowpass filtered image :

```
% Convert the image to grayscale format
image1 = im2single(imread('dog.bmp'));
Img1 = rgb2gray(image1);
%Determine good padding for Fourier transform
PQ = paddedsize(size(Img1));
%Create a Gaussian Lowpass filter 5% the width of the Fourier transform
D0 = 0.03*PQ(1);
H = lpfilter('gaussian', PQ(1), PQ(2), D0);
% Calculate the discrete Fourier transform of the image
F=fft2(double(I),size(H,1),size(H,2));
% Apply the highpass filter to the Fourier spectrum of the image
LPFS_I = H.*F;
% convert the result to the spacial domain.
LPF_I=real(ifft2(LPFS_I));
% Crop the image to undo padding
LPF_I=LPF_I(1:size(I,1), 1:size(I,2));
%Display the blurred image
subplot(2,2,2), imshow(LPF_I, [])
% Display the Fourier Spectrum
% Move the origin of the transform to the center of the frequency rectangle.
Fc=fftshift(F);
Fcf=fftshift(LPFS_I);
% use abs to compute the magnitude and use log to brighten display
S1=log(1+abs(Fc));
S2=log(1+abs(Fcf));
output = LPF_I;
```

Highpass filtered image :

```
% Convert the image to grayscale format
image1 = im2single(imread('cat.bmp'));
Img1 = rgb2gray(image1);
%Determine good padding for Fourier transform
PQ = paddedsize(size(Img1));
%Create a Gaussian Lowpass filter 5% the width of the Fourier transform
D0 = 0.03*PQ(1);
H = hpfilter('gaussian', PQ(1), PQ(2), D0);
% Calculate the discrete Fourier transform of the image
F=fft2(double(I),size(H,1),size(H,2));
% Apply the highpass filter to the Fourier spectrum of the image
HPFS_I = H.*F;
% convert the result to the spacial domain.
HPF_I=real(ifft2(HPFS_I));
% Crop the image to undo padding
HPF_I=HPF_I(1:size(I,1), 1:size(I,2));
%Display the blurred image
subplot(2,2,2), imshow(HPF_I, [])
% Display the Fourier Spectrum
% Move the origin of the transform to the center of the frequency rectangle.
Fc=fftshift(F);
Fcf=fftshift(HPFS_I);
% use abs to compute the magnitude and use log to brighten display
S1=log(1+abs(Fc));
S2=log(1+abs(Fcf));
output = HPF_I;
```

EXTRA CREDIT :
NOTCH FILTERS :

- are used to remove repetitive "Spectral" noise from an image
- are like a narrow highpass filter, but they "notch" out frequencies other than the dc component
- attenuate a selected frequency (and some of its neighbors) and leave other frequencies of the Fourier transform relatively unchanged

Repetitive noise in an image is sometimes seen as a bright peak somewhere other than the origin. We can suppress such noise effectively by carefully erasing the peaks. One way to do this is to use a notch filter to simply remove that frequency from the picture.

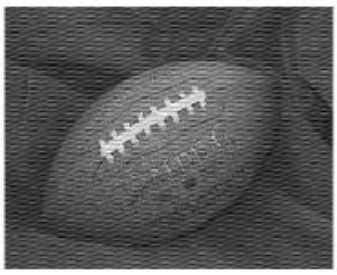
notch.m

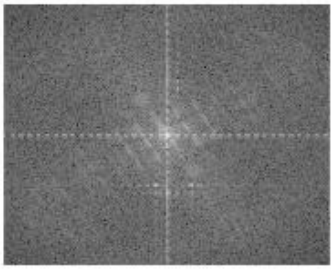
```
function H = notch(type, M, N, D0, x, y, n)
%notch Computes frequency domain notch filters
%   H = NOTCH(TYPE, M, N, D0, x, y, n) creates the transfer function of
%   a notch filter, H, of the specified TYPE and size (M-by-N). centered at
%   Column X, Row Y in an unshifted Fourier spectrum.
%   Valid values for TYPE, D0, and n are:
%
%   'ideal'     Ideal highpass filter with cutoff frequency D0.  n
%               need not be supplied.  D0 must be positive
%
%   'btw'       Butterworth highpass filter of order n, and cutoff D0.
%               The default value for n is 1.0.  D0 must be positive.
%
%   'gaussian'  Gaussian highpass filter with cutoff (standard deviation)
%               D0.  n need not be supplied.  D0 must be positive.
%

% The transfer function Hhp of a highpass filter is 1 - Hlp,
% where Hlp is the transfer function of the corresponding lowpass
% filter.  Thus, we can use function lpfilter to generate highpass
% filters.

if nargin == 6
   n = 1; % Default value of n.
end

% Generate highpass filter.
Hlp = lpfilter(type, M, N, D0, n);
H = 1 - Hlp;
H = circshift(H, [y-1 x-1]);
```

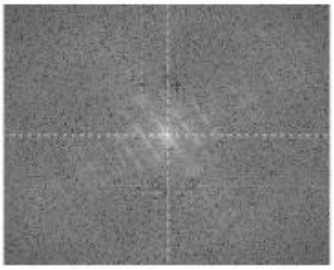
notch_filter_demo.m

```
image=imread('noiseball.png');
subplot(2,2,1), imshow(image)

%Determine good padding for Fourier transform
PQ = paddedsize(size(image));
```

```
%Create Notch filters corresponding to extra peaks in the Fourier transform
H1 = notch('btw', PQ(1), PQ(2), 10, 50, 100);
H2 = notch('btw', PQ(1), PQ(2), 10, 1, 400);
H3 = notch('btw', PQ(1), PQ(2), 10, 620, 100);
H4 = notch('btw', PQ(1), PQ(2), 10, 22, 414);
H5 = notch('btw', PQ(1), PQ(2), 10, 592, 414);
H6 = notch('btw', PQ(1), PQ(2), 10, 1, 114);

% Calculate the discrete Fourier transform of the image
F=fft2(double(image),PQ(1),PQ(2));

% Apply the notch filters to the Fourier spectrum of the image
FS_image = F.*H1.*H2.*H3.*H4.*H5.*H6;

% convert the result to the spacial domain.
F_image=real(ifft2(FS_image));

% Crop the image to undo padding
F_image=F_image(1:size(image,1), 1:size(image,2));

%Display the noise removed image
subplot(2,2,2), imshow(F_image,[])

% Display the Fourier Spectrum
% Move the origin of the transform to the center of the frequency rectangle.
Fc=fftshift(F);
Fcf=fftshift(FS_image);

% use abs to compute the magnitude and use log to brighten display
S1=log(1+abs(Fc));
S2=log(1+abs(Fcf));
subplot(2,2,3), imshow(S1,[])
subplot(2,2,4), imshow(S2,[])
```







Problem Set 2 : Template Matching

Case-1: template matching with fixed scale: match the cropped template to the query image

Algorithm 1 : Sum Square Difference (SSD)

$$h[m,n] = \sum_{k,l} (g[k,l] - f[m+k,n+l])^2$$

Problems : Sensitive to average intensity

SSDTemplateMatching.m

```
function [MatchingPosition] = SSDTemplateMaching(TemplateImage,QueryImage)
%Get the size of the image and template
[TemplateHeight,TemplateWidth]=size(TemplateImage);
[QueryImageHeight,QueryImageWidth]=size(QueryImage);
%Loop through to perform the SSD algorithm
for i=1:QueryImageHeight-TemplateHeight+1
    for j=1:QueryImageWidth-TemplateWidth+1
            SSDMatrix(i,j)=sum(sum((double(QueryImage(i:i+TemplateHeight-1,j:j
+TemplateWidth-1))-double(TemplateImage)).^2));
    end
end
MinSSD=0;
% Get the best possible match of the template image
for i=1:QueryImageHeight-TemplateHeight+1
    for j=1:QueryImageWidth-TemplateWidth+1
        if (i == 1) && (j==1)
            MinSSD=SSDMatrix(i,j);
        else
            if SSDMatrix(i,j)<MinSSD
                MatchingPosition=[i;j];
                MinSSD=SSDMatrix(i,j);
            elseif (SSDMatrix(i,j) == MinSSD) && (j<MatchingPosition(2))
                MatchingPosition=[i;j];
                MinSSD=SSDMatrix(i,j);
            end
        end
    end
end
end
```

Result:

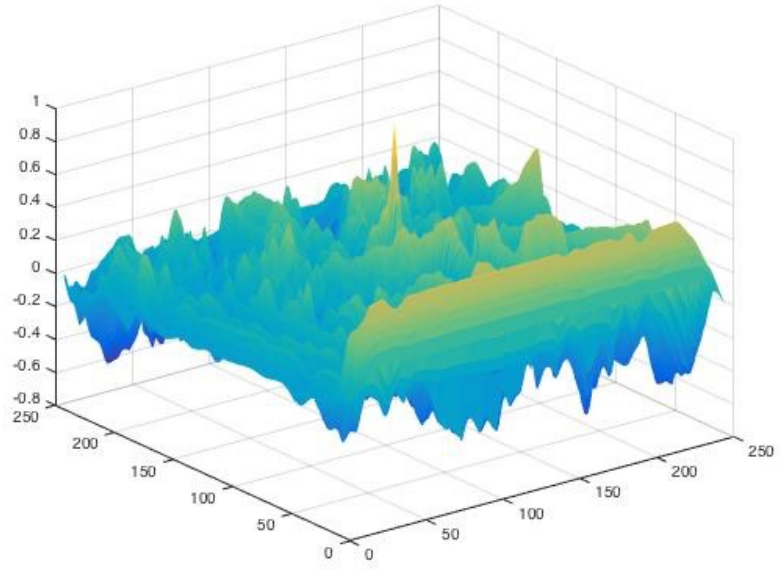
Figure : left : query image and template image ; right: template image matched with SSD

Algorithm 2 : Normalized Cross Correlation (NCC)

Function : c = normxcorr2(template,image);

$$h[m,n] = \frac{\sum_{k,l}(g[k,l]-\bar{g})(f[m-k,n-l]-\bar{f}_{m,n})}{\left(\sum_{k,l}(g[k,l]-\bar{g})^2\sum_{k,l}(f[m-k,n-l]-\bar{f}_{m,n})^2\right)^{0.5}}$$

Figure : left : query image and template image ; right: template image matched with NCC



Algorithm 3 : Zero Mean Correlation (ZMC)

$$h[m,n] = \sum_{k,l} (f[k,l] - \bar{f})\,(g[m+k,n+l])$$

ZMCTemplateMatching.m (only algorithm part)

```
function [MatchingPosition] = ZMCTemplateMatching(template, image)

  temp = template - mean(template(:));
  img = image;
    [TemplateHeight,TemplateWidth]=size(temp);
    [QueryImageHeight,QueryImageWidth]=size(img);
% Zero- mean algorithm
for i=1:QueryImageHeight-TemplateHeight+1
    for j=1:QueryImageWidth-TemplateWidth+1
        ZMCMatrix(i,j)=sum(sum((double(img(i:i+TemplateHeight-1,j:j
      +TemplateWidth-1)).*double(temp)));
    end
end
.
.
.
end
```

Result:

Figure : left : query image and template image ; right: template image false matched with zero-mean correlation

Error is calculated using euclidean distance : EuclideanDistance.m
or using norm( ) function.

Case 2: Image Pyramid

buildPyramid.m

```
function [ img_pyramid ] = buildPyramid( img,PyLevel )
%BUILDPYRAMID Summary of this function goes here
%   Detailed explanation goes here
%   Input:
%   img:     input image
%   PyLevel: level of Pyramids needed.  default :4

if nargin<2
    PyLevel=4;
end
img_pyramid=cell(PyLevel,1);
img_pyramid{1}=img;

for i=2:1:PyLevel
    img_pyramid{i}=impyramid(img_pyramid{i-1},'reduce');

end
end
```