# Computer Vision Homework 2

Yao Cai

23533599

## Hybrid Images

### 1\ Code

```
import matplotlib.pyplot as plt
from align_image_code import align_images
from scipy import signal
from math import pi, sqrt, exp
from skimage import io
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from skimage import color
import os

def gaussian_kernel(sigma, truncate=4.0):

    n = int(2*sigma*truncate + 1)
    gaussian = np.zeros((n, n))
    mean = np.array([n//2, n//2])
    for i in range(n):
        for j in range(n):
            gaussian[i, j] = 1/(2*pi*sigma**2)*exp(-((i-mean[0])**2+(j-mean[1])**2)/2/sigma**2)
    return gaussian/gaussian.sum()

def LoG(sigma, truncate=4.0):
        lowpass = gaussian_kernel(sigma, truncate)
```

```
    highpass = np.zeros(lowpass.shape)
    i = lowpass.shape[0]//2
    highpass[i, i] = 1
    highpass = highpass - lowpass
    return highpass


def hybrid_image(im1, im2, sigma1, sigma2):
    highpass = LoG(sigma = sigma1)
    im1_filter = np.zeros(im1.shape)
    for i in range(3):
        im1_filter[:, :, i] = signal.convolve2d(im1[:, :, i], highpass, boundary='symm',
mode='same')
    im1_filter = (im1_filter-im1_filter.min())/(im1_filter.max()-im1_filter.min())

    lowpass2 = gaussian_kernel(sigma = sigma2)
    im2_filter = np.zeros(im2.shape)
    for i in range(3):
        im2_filter[:, :, i] = signal.convolve2d(im2[:, :, i], lowpass2, boundary='symm',
mode='same')

    return im1_filter, im2_filter, im1_filter*0.5+im2_filter*0.5
```

**2\ Results**

Lowpass the first image(convolve with lowpass filter), highpass the second image(convolve with highpass filter), and linearly combine them, so when we take a closer look we recognize the second image and when we take several step back, we recognize the first image.

Lowpass filter: Gaussian, the only parameter is the standard deviation sigma. Larger sigma, more blurred image, which means more high frequency components have been removed. Generally, we want a filtered image with some high frequency removed, but not too much, so that we can still be able to recognize the image. So sigma for lowpass filter can not be too large.

$$G_\sigma = \frac{1}{2\pi\sigma^2} e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$

Highpass filter:

**M-G**

Where **G** is the Gaussian filter defined above, **M** is a unit impulse filter, a matrix with the same size as G and all zero except the center pixel. The parameter to tune is sigma in G, the larger sigma, sharper the image, which means more high frequency components have been kept. Generally, we want the sigma used in highpass filter **smaller** or equal to sigma used in lowpass filter so the frequency

components from the two images doesn't interfere with each other. (Larger sigma in real space means smaller sigma in frequency space)

Below is my favorite hybrid image of Taylor Swift and Anne Hathaway:

Original Images:

Filtered images(highpass sigma=5, lowpass sigma=5):

The hybrid(at two scales):

Below is the frequency analysis for both sigma = 5, we can see that indeed highpass filtered image doesn’t have low frequency components, and low pass filtered image doesn’t have high frequency components.

image1
highpass of image1, sigma=5
image2
lowpass of image2, sigma=5
hybrid
0
200
400
600
800
1000
0.0
0.2
0.4
0.6
0.8
1.0

**More results!**

Generally good sigma should be compatible with the features on the images. If the features are small, we would need small sigmas. First choose the sigma for the low pass image, so that even blurred we can still recognize it. Then tune the sigma for the high pass image.

Higher highpass sigma means more low frequency component, and the filtered highpass image is more colorful with higher saturation. This is good for the big image, but not good for the small image, since the low frequency components in highpass image interfere with low frequency components in the lowpass image.

(By small image I mean viewed from a larger distance, and large image I mean viewed from a smaller distance)

Lower highpass sigma means less low frequency component, this is good for recognition of the small image, but results in the low saturation of the high pass image, so that the colors in the big image mostly come from the low pass image, and when the two images cannot be well aligned, we will end up with a big image not so real.

This is why alignment is so important, alignment means that one feature in one image can be understood as a similar or compatible feature in another image. In all of my images, they generally have “OK” alignment but not perfect. If the alignment is super good, then we can just use a lower highpass sigma, and get both good small and big images, as discussed before. But since in all my images, they are not super good, so the sweet spot that I found for all my hybrid images is that when the sigma of the high pass filter is the same as the low pass filter.

0
50
100
150
200
250
300
350
0
100
200
300
400
500

0
50
100
150
200
250
300
350
0
100
200
300
400
500

0
50
100
150
200
250
300
350
0
100
200
300
400
500

0
100
200
300
400
0
100
200
300

0
100
200
300
400
0
100
200
300

0
100
200
300
400
0
100
200
300

0
100
200
300
400
0
100
200

0
100
200
300
400
0
100
200

0
100
200
300
400
0
100
200

# 2 Edge Detection

One of the state-of-art edge detector examples is shown below as benchmark:

**1\ Finite operator**

I use arctan2 function to calculate the angle along each RGB channel and then take an average of the three channels. We cannot use norm here, since norm are positive, but angles can be positive or negative from -180 to 180 degree. Average makes more sense than norm for angles.

Below are gradient magnitude and gradient orientation mappings for several different thresholds, for an example image:

gradient magnitude
gradient orientation
0
100
200
300
400
0
100
200
300
150
100
50
0
−50
−100
−150

gradient magnitude
gradient orientation
0
100
200
300
400
0
100
200
300
150
100
50
0
−50
−100
−150

**2\ Derivative of Gaussian**

Differentiation can be calculated as convolution, and since convolution are commutative and associative, we can convolve any two of the matrices as a first step. But we want to begin with small matrices, and leave the large image matrix as last operator, so that we don't have to deal with large matrix multiple times. In this way, the speed of calculation is much faster.

Below are the plots for Derivative of Gaussian along x and y direction, respectively.

Below are gradient magnitude and gradient orientation mappings for several different thresholds, for an example image:

We can see that using derivative of gaussians, the resulting filtered images are less noisy with clear edges. But we don't want the details in some of the textures detected as edges, for example, the details in the trees at riverside. By using a higher threshold, we somehow remove the details in the textures, but at the same time remove some weaker edges as well.

**3\ Oriented Filters**

I use 18 Elongated Gaussian Derivative filter along 6 different directions(0, 30, 60, 90, 120, 150 degree) and 3 different sizes(sigma2 = sqrt(2), 2* sqrt(2), 3* sqrt(2), sigma1 = 3*sigma2). These are the 18 elongated Gaussian 1$^{st}$ derivative used in Leung-Malik Filter Bank.

Then at each pixel, I choose the largest response from each filter as element in the gradient magnitude matrix, and I use the angle of the oriented filter with the largest response as element in the gradient orientation matrix.

The mapping are shown below for several threshold. We can observe that Oriented Filters are better at smoothing the details in textures than the DoG filters. For example, in the filtered image without using threshold, there are still some details in the leaves on his arm, but the details are somewhat smoothed.

gradient magnitude
gradient orientation
0
100
200
300
400
0
100
200
300
0
20
40
60
80
100
120
140

gradient magnitude
gradient orientation
0
100
200
300
400
0
100
200
300
0
20
40
60
80
100
120
140

**4\ Compare with state-of-art result and human annotation**

Human labelers will ignore any texture. The start-of-art results are better at ignoring textures than my best algorithm. My best algorithm is good at finding edges of high contrast, but struggles with ignoring the details in textures. Human labelers will probably ignore the shadows on clothes, but my algorithm are not able to ignore them since sometimes they have large contrast. Human will probably ignore stripes on clothes or on zebras, but these are all high contrast and hard to ignore by my algorithms.

We may need some algorithm that can recognize textures and and remove them from edge response. I think this is the biggest challenge in edge detection.

Below I will attach some images and their filtered images with best performance.

gradient magnitude
0
200
400
600
800
0
200
400
600
gradient orientation
0
200
400
600
800
0
200
400
600
140
120
100
80
60
40
20
0

LITTLE
WILD
CHILD

gradient magnitude
0
200
400
600
800
0
200
400
600

gradient orientation
0
200
400
600
800
0
200
400
600
140
120
100
80
60
40
20
0

www.Disney.com.cn
星际宝贝

gradient magnitude
gradient orientation
0
100
200
300
400
500
0
200
400
600
140
120
100
80
60
40
20
0

gradient magnitude
0
100
200
300
0
100
200
300
400
gradient orientation
0
100
200
300
0
100
200
300
400
150
100
50
0
−50
−100
−150

gradient magnitude
0
100
200
300
0
100
200
300
400
gradient orientation
0
100
200
300
0
100
200
300
400
150
100
50
0
−50
−100
−150

gradient magnitude
gradient orientation
0
100
200
300
400
140
120
100
80
60
40
20

gradient magnitude
gradient orientation
0
100
200
300
0
100
200
300
400
140
120
100
80
60
40
20
0

```
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt
import os
from numpy.linalg import norm, inv
from matplotlib import cm
from math import pi, sqrt, exp
from mpl_toolkits.mplot3d import Axes3D
from math import cos, sin


def gaussian_kernel(sigma, truncate=4.0):
	n = int(2*sigma*truncate + 1)
	gaussian = np.zeros((n, n))
	mean = np.array([n//2, n//2])
	for i in range(n):
		for j in range(n):
			gaussian[i, j] = 1/(2*pi*sigma**2)*exp(-((i-mean[0])**2+(j-
			mean[1])**2)/2/sigma**2)
	return gaussian/gaussian.sum()

def elongated_gaussian_derivative(sigma1, sigma2, theta, truncate=4.0):
	n = int(sigma1*truncate)
	x, y = np.meshgrid(np.arange(-n, n+1), np.arange(-n, n+1))
	orgpts = np.vstack((x.reshape(-1), y.reshape(-1)))

	theta = theta/180*pi
	rotation = np.array([[cos(theta), sin(theta)], [-sin(theta), cos(theta)]])
	rotpts = rotation @ orgpts

	gx = np.exp(-1/2*(rotpts[0, :])**2/sigma1**2)
	gy = np.exp(-1/2*(rotpts[1, :])**2/sigma2**2) * (-(rotpts[1, :])/sigma2**2)
	egd = (gx * gy).reshape(2*n+1, 2*n+1)
	egd_plus = egd * (egd>0)
	return egd/egd_plus.sum()

def difference_filter(I):
	dx = np.array([[1, -1]])
	dy = np.array([[1], [-1]])

	return get_gradient_images(I, dx, dy)

def derivative_gaussian_filter(I, sigma):
	dx = np.array([[1, -1]])
	dy = np.array([[1], [-1]])
	dogx = signal.convolve2d(gaussian_kernel(sigma), dx, boundary='symm', mode='same')
	dogy = signal.convolve2d(gaussian_kernel(sigma), dy, boundary='symm', mode='same')
```

```
    plot_filter(dogx, 'derivative_gaussian_x.jpg')
    plot_filter(dogy, 'derivative_gaussian_y.jpg')

    return get_gradient_images(I, dogx, dogy)

def get_gradient_images(I, filterx, filtery):
    img_x = np.zeros(I.shape)
    img_y = np.zeros(I.shape)
    for i in range(3):
        img_x[:, :, i] = signal.convolve2d(I[:, :, i], filterx, boundary='symm', mode='same')
        img_y[:, :, i] = signal.convolve2d(I[:, :, i], filtery, boundary='symm', mode='same')
    img_x_norm = norm(img_x, axis = 2)
    img_y_norm = norm(img_y, axis = 2)

    img_mag = np.concatenate((img_x, img_y), axis = 2)
    img_mag = norm(img_mag, axis = 2)

    img_orient = np.arctan2(img_y, img_x).mean(axis=2)/pi*180
    return img_x_norm, img_y_norm, img_mag, img_orient

def oriented_filter(I):
    thetas = np.array([0, 30, 60, 90, 120, 150])
    sigma1s = 3*sqrt(2) ** np.array([1, 2, 3])
    theta, sigma1 = np.meshgrid(thetas, sigma1s)
    theta = theta.reshape(-1)
    sigma1 = sigma1.reshape(-1)

    img_mag = np.zeros((I.shape[0], I.shape[1], len(theta)))
    img_orient = np.zeros((I.shape[0], I.shape[1], len(theta)))

    for i in range(len(theta)):
        egd = elongated_gaussian_derivative(sigma1[i], sigma1[i]/3, theta[i])
        img_mag[:, :, i] = convolve3channel(I, egd)
        img_orient[:, :, i] = theta[i]

    img_max_repeat= np.repeat(img_mag.max(axis=2)[:, :, np.newaxis], len(theta), axis=2)
    indexes = np.equal(img_mag, img_max_repeat)
    img_orient = (img_orient * indexes).max(axis=2)

    img_mag = img_mag.max(axis=2)

    return img_mag, img_orient

def convolve3channel(I, filter_):
    filtered = np.zeros(I.shape)
    for i in range(3):
        filtered[:, :, i] = signal.convolve2d(I[:, :, i], filter_, boundary='symm', mode='same')
    filtered = norm(filtered, axis = 2)
    return filtered
```