CVHW2

team members:0510134 王泓仁、0516006 賴奕善、0516218 軒轅照雯

Work assignment

Task 1. Hybrid image - 軒轅照雯
Task 2. Image pyramid - 王泓仁
Task 3. Colorizing the Russian Empire - 賴奕善

Task 1. Hybrid image

I. Introduction

Hybrid images are a form of illusion which exploits the multiscale visual perspective of human vision. These images take advantage of the fact that our vision relies on different frequencies in order to make sense of an image from different distances (or images of different sizes). When we are further away from the image (or when the image is smaller) we rely more on the lower frequencies of the image and the human eye attempts to "fill in the gaps" with additional details, creating groupings that collectively represent an object in an image. From closer distances (or when the image is larger) we use high frequency details in the image to enhance our understanding. By superimposing the high frequency portion of one image with the low-frequency portion of another, you get a hybrid image that leads to different interpretations at different distances.

II. Implementation procedure

(1) Check image sizes

If the two input images have different sizes, for each coordinate, resize the larger one to the same size as the smaller one with least center movement. (imgResize) Here we omit the alignment, since the given data are already aligned.

```
# Resize if different image sizes
if np.any(img1.shape!=img2.shape):
    img1, img2=imgResize(img1, img2)
```

```
def imgResize(img1,img2):
    diff_x=abs(img1.shape[0]-img2.shape[0])
    diff_y=abs(img1.shape[1]-img2.shape[1])

    if img1.shape[0]>img2.shape[0]:
        lower=int(diff_x/2)
        upper=img1.shape[0]-lower
        if (diff_x%2)==1:
            upper=upper-1
        img1=img1[lower:upper, :,:]
    else:
        lower=int(diff_x/2)
        upper=img2.shape[0]-lower
        if (diff_x%2)==1:
            upper=upper-1
        img2=img2[lower:upper, :,:]

    if img1.shape[1]>img2.shape[1]:
        lower=int(diff_y/2)
        upper=img1.shape[1]-lower
        if (diff_y%2)==1:
            upper=upper-1
        img1=img1[:,lower:upper,:]
    else:
        lower=int(diff_y/2)
        upper=img2.shape[1]-lower
        if (diff_y%2)==1:
            upper=upper-1
        img2=img2[:,lower:upper,:]
    return img1, img2
```

(2) Fourier transformation

Since we are applying frequency domain filtering, for each channel of the two images, multiply the input images by $(-1)^{x+y}$ (shift) to center the transform , then do the 2D Fourier transformation (ft).

```
# Multiply each channel of the input image by (-1)^(x+y)
# Do 2D Fourier transformation on each channel
f_img1=ft(shift(img1))
f_img2=ft(shift(img2))
```

```
def shift(img):
    shift_img=np.zeros((img.shape))
    for x in range(img.shape[0]):
        for y in range(img.shape[1]):
            for z in range(img.shape[2]):
                shift_img[x,y,z]=img[x,y,z]*((-1)**(x+y))
    return shift_img
```

```
def ft(img):
    f_img=np.zeros((img.shape), dtype = complex)
    for z in range(img.shape[2]):
        f_img[:,:,z]=np.fft.fft2(img[:,:,z])
    return f_img
```

(3) Filtering

1. Decide the cutoff frequencies (L_D0, H_D0) for each pair of input images.
2. Generate the corresponding low pass filters by giving the third parameter “True” and high pass filters by giving the third parameter “False”.
3. Multiply each channel of one of the Fourier transformed images by the low pass filters and the other by the high pass filters.

```
L_D0=[15,10,10,15,10,10,10,15,20]
H_D0=[30,20,25,15,25,25,30,20,25]
```
(index 0~6 for given data pairs, 78 for our own data)

```
# Ideal filter
iLP_fimg=idealFilter(L_D0[i], f_img1, lowPass=True)
iHP_fimg=idealFilter(H_D0[i], f_img2, lowPass=False)      #highPass

# Gaussian filter
gLP_fimg=gaussianFilter(L_D0[i], f_img1, lowPass=True)
gHP_fimg=gaussianFilter(H_D0[i], f_img2, lowPass=False)   #highPass
```

```
def dist(x, y):
    return math.sqrt((x[0]-y[0])**2+(x[1]-y[1])**2)
```

```
def idealFilter(D0, img, lowPass):
    filtered=np.zeros((img.shape), dtype = complex)
    center=(img.shape[0]/2, img.shape[1]/2)
    if lowPass:
        f=np.zeros((img.shape[:2]))
    else:
        f=np.ones((img.shape[:2]))
    for x in range(img.shape[0]):
        for y in range(img.shape[1]):
            if dist(center, (x,y))<D0:
                f[x,y]=int(lowPass)
    for z in range(img.shape[2]):
        filtered[:,:,z]=img[:,:,z]*f
    return filtered
```

```
def gaussianFilter(D0, img, lowPass):
    filtered=np.zeros((img.shape), dtype = complex)
    center=(img.shape[0]/2, img.shape[1]/2)
    f=np.zeros((img.shape[:2]))
    for x in range(img.shape[0]):
        for y in range(img.shape[1]):
            f[x,y]=math.exp(((-(dist((x,y),center))**2)/(2*(D0**2))))
    if lowPass==False:
        f=1-f
    for z in range(img.shape[2]):
        filtered[:,:,z]=img[:,:,z]*f
    return filtered
```

(4) Inverse Fourier transformation

1. For each channel, do the 2D inverse Fourier transformation on the filtered images. (ift)

2. After Fourier transformations, complex values are formed; however, the imaginary parts are too small that we can discard them, that is, obtain the real parts in each pixel value of the results in 1.

3. Multiply each channel of the results in 2. by $(-1)^{x+y}$. (shift)

```
# Do 2D inverse Fourier transformation on each channel
# Obtain the real parts
# Multiply each channel of the filtered image by (-1)^(x+y)
iLP_img=shift(ift(iLP_fimg).real)
iHP_img=shift(ift(iHP_fimg).real)
gLP_img=shift(ift(gLP_fimg).real)
gHP_img=shift(ift(gHP_fimg).real)
```

```
def ift(img):
    if_img=np.zeros((img.shape), dtype = complex)
    for z in range(img.shape[2]):
        if_img[:,:,z]=np.fft.ifft2(img[:,:,z])
    return if_img
```

(5) Create the hybrid image

Add the low passed image and high passed image to create the hybrid image.

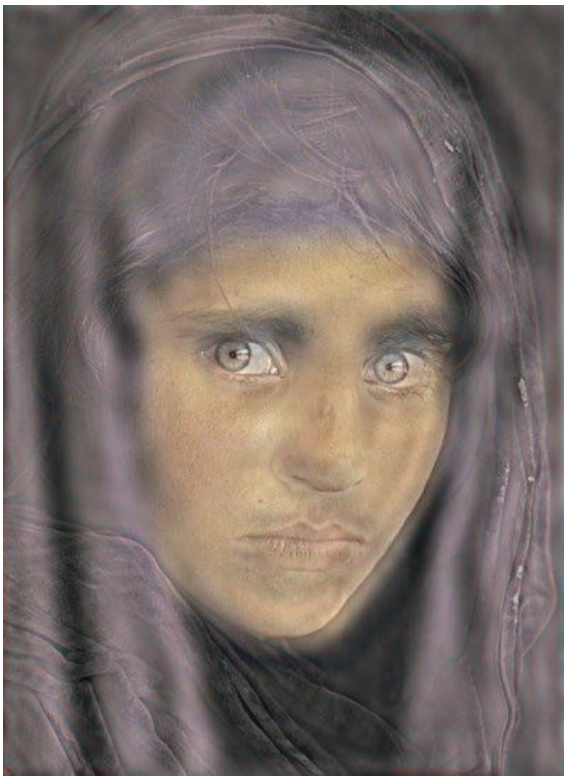
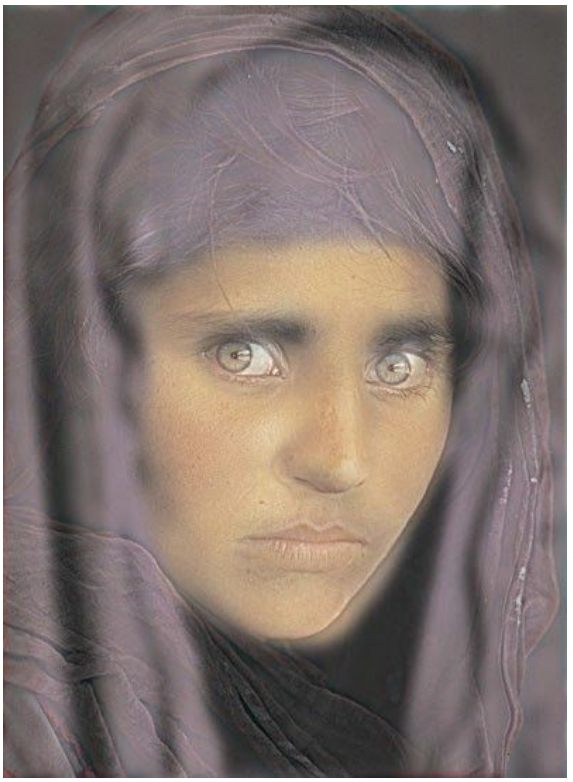
```
# Hybrid
ihybrid_img=iLP_img+iHP_img
ghybrid_img=gLP_img+gHP_img
```

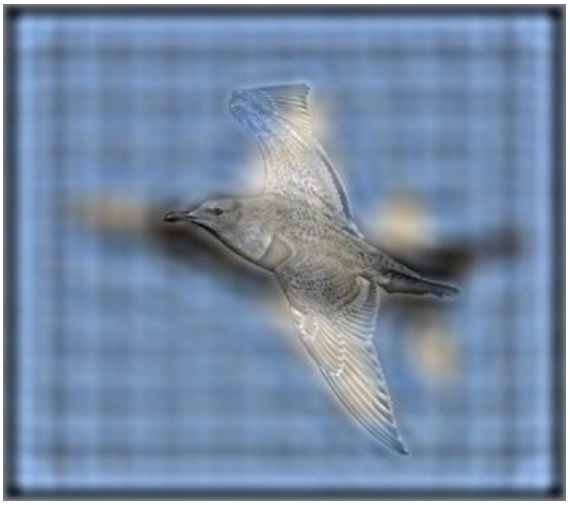
III. Experimental results

(1) Hybrid images generated with ideal filters v.s gaussian filter

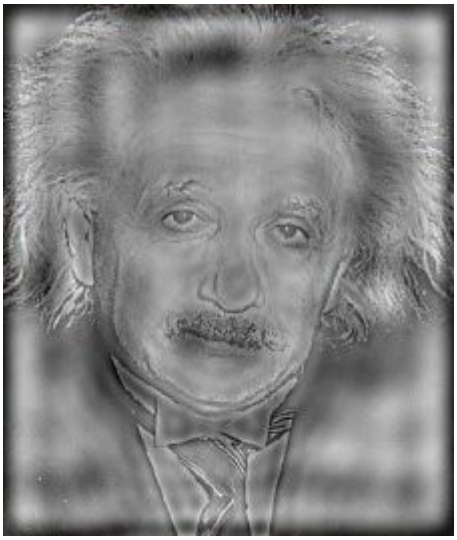
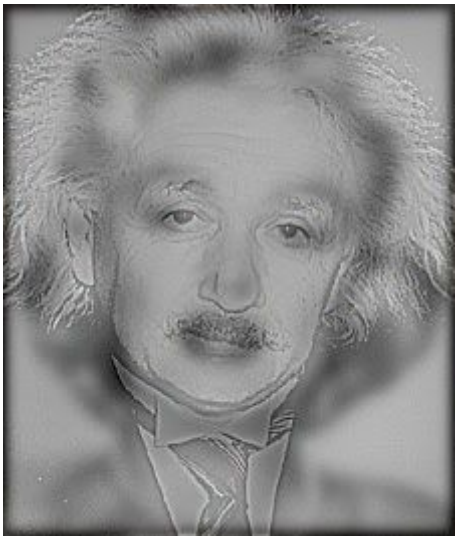
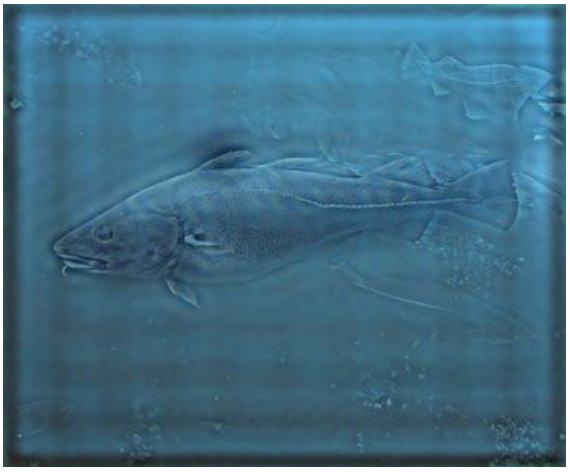
- same cutoff frequencies for different filters

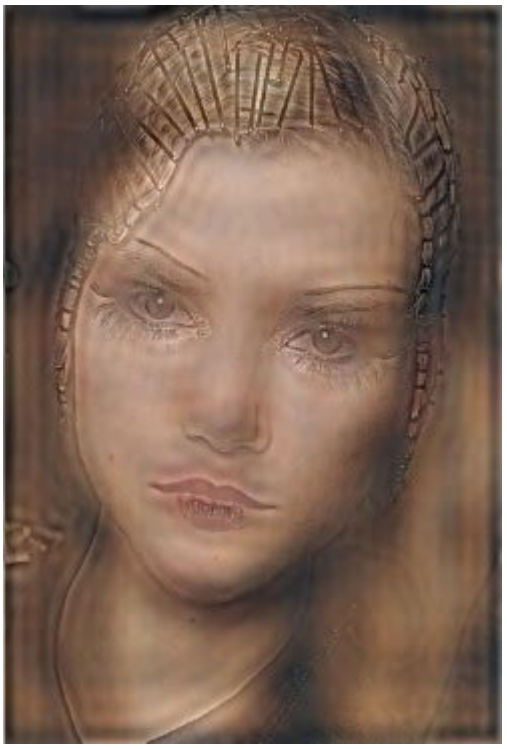
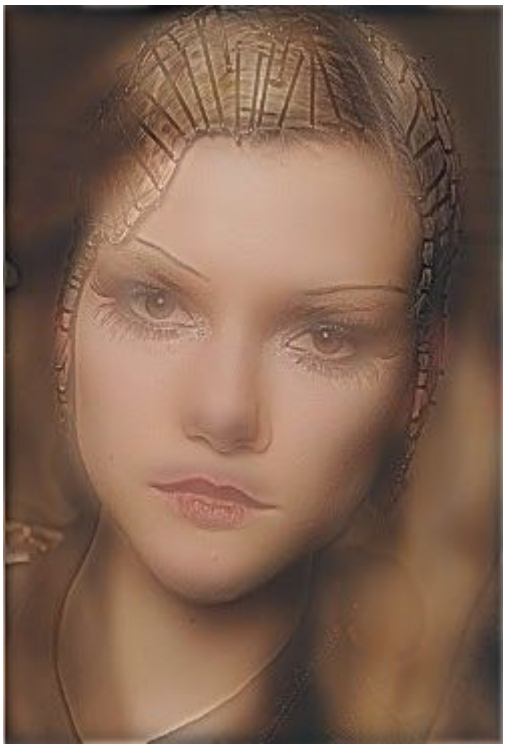
Ideal filter: Gaussian filter:

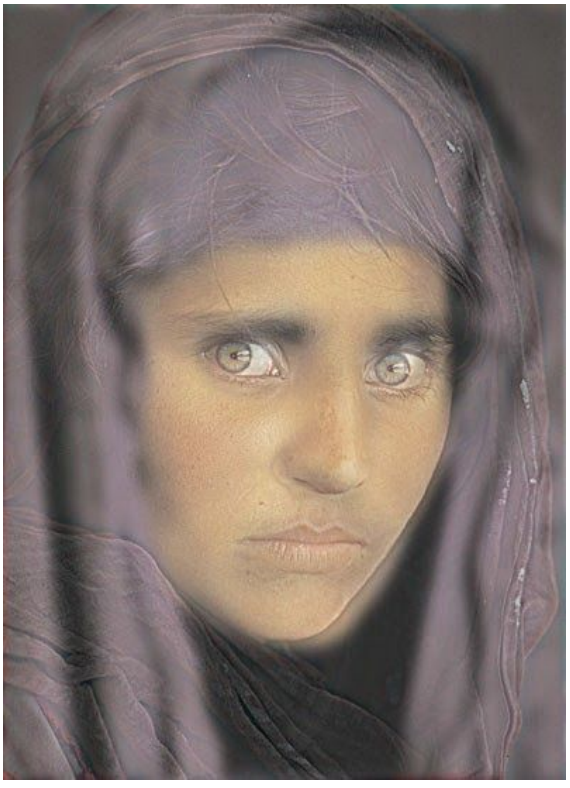





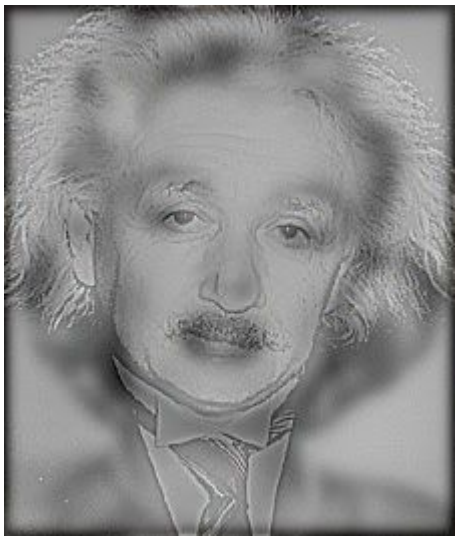
(2) Hybrid images generated by same pairs of images but with opposite filters
- same gaussian filters and same cutoff frequencies in (1) but applying to opposite images

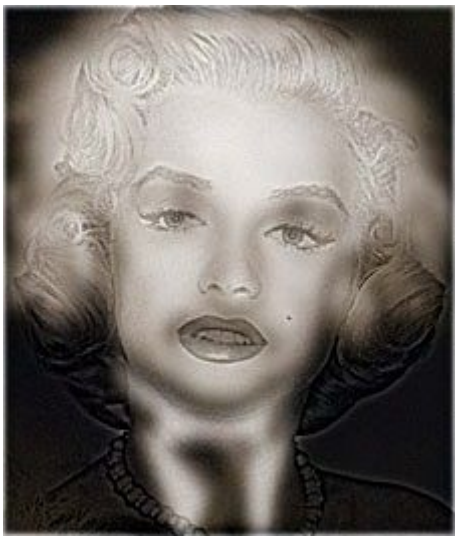


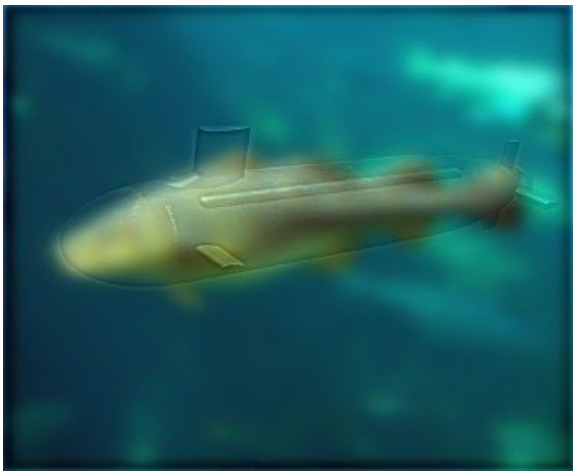












(3) Hybrid images generated by our own images

a. images of a curling tool and medicine


- results:


b. images of a person with eyes opened and closed

- results:

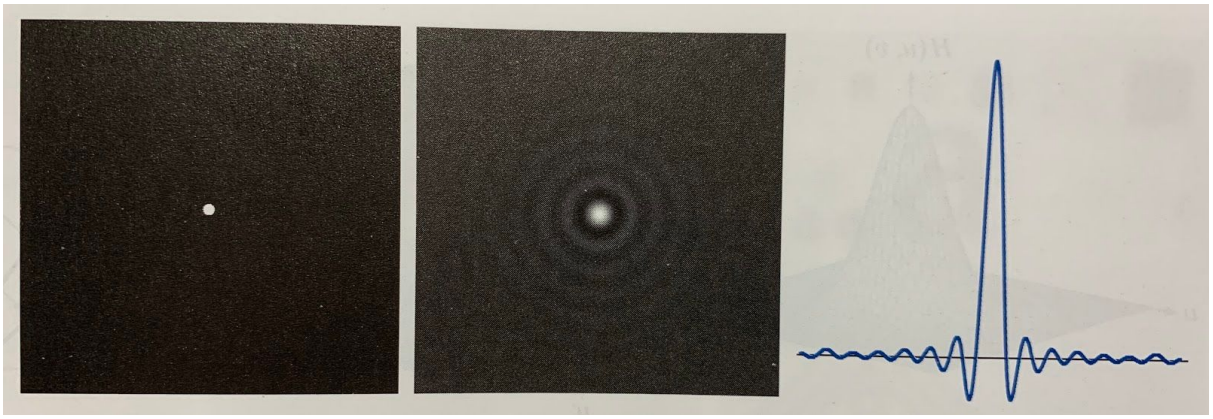
IV. Discussion

(1) Filters

We can observe significant ringing effects in the hybrid images generated by ideal filters. It can be explained using the convolution theorem.

Picture (a) shows an image of a frequency-domain ideal low pass filter and picture (b) is the spatial representation of ideal low pass filter obtained by taking the inverse Fourier transformation of (a). Picture (c) shows the intensity profile of a line passing through the center of (b), it resembles a sinc function. Filtering in spatial domain is done by convolving the function in (b) with an image.

(a) (b) (c)



Since the inverse Fourier transform of a frequency-domain Gaussian function is Gaussian also, a spatial Gaussian filter kernel obtained by computing the inverse Fourier transformation of $H(u,v) = e^{-D^{2(u,v)/2D0^2}}$, will have no ringing.

(2) Good hybrid images

It is clear that hybrid images are effective only in certain situations. As we can see, the images need to be aligned, but the colors and structure of the two images need to provide a strong illusion. In some cases, it also matters which image has the high pass band filter applied, due to the fact that some objects are simply more understandable based on either their low frequency or high frequency characteristics.

Good low-pass filtered images should "lack a precise definition of object shapes and region boundaries". In other words, one should be able to determine what the object is from a far distance just by looking at its approximate shapes and regions. Examples of such objects used in the above experiments are the plane, the dog with dark nose, Einstein with a tie and mustache, the submarine, the girl before makeup and our own data of a person with eyes closed.

Lastly, cutoff frequencies also play an important role. Since every pair of images has different details and structures, different cutoff frequencies must be used to provide better results.

(3) Difficulties

We encountered some type errors, since we applied Fourier transformation on images and some complex numbers are generated, just make sure to specify the correct type of each variable.

We also met a color channel problem on image I/O which made images become “bluish”, due to the different channel orders of each module, use identical modules for both input and output instead to solve it.

V. Conclusion

This task explores the realm of image frequencies and shows how one can manipulate the interpretation of an image by deliberately changing its characteristics. Hybrid images do not have much application to the real world but very effectively show how our vision system processes and understands images and objects around us.

Task 2. Image pyramid

I. Introduction

The Gaussian pyramid

The Gaussian pyramid is a technique in image processing that breaks down an image into successively smaller groups of pixels, in repeated steps, for the purpose of blurring it. It is named after German mathematician Johann Carl Friederich Gauss. This type of precise mathematical blurring is used extensively in artificially intelligent computer vision as a pre-processing step. For instance, when a digital photograph is blurred in this way, edges of objects are much easier to detect, enabling a computer to identify them automatically.

The ``pyramid_gaussian`` function takes an image and yields successive images shrunk by a constant scale factor. Image pyramids are often used, e.g., to implement algorithms for denoising, texture discrimination, and scale-invariant detection.

The Laplacian Pyramid

The general class of linear transform decomposes an image into various components by multiplication with a set of transform functions. Some examples are the Discrete Fourier and Discrete Cosine Transforms, the Singular Value

Decomposition, and finally, the Wavelet Transform, of which the Laplacian Pyramid and other subband transforms are simple ancestors.

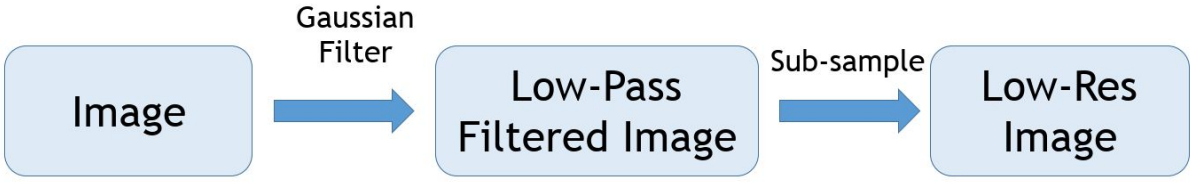
The magnitude spectrum:

Fourier Transform is used to analyze the frequency characteristics of various filters. For images, 2D Discrete Fourier Transform (DFT) is used to find the frequency domain. A fast algorithm called Fast Fourier Transform (FFT) is used for calculation of DFT. Details about these can be found in any image processing or signal processing textbooks.

More intuitively, for the sinusoidal signal, if the amplitude varies so fast in short time, you can say it is a high frequency signal. If it varies slowly, it is a low frequency signal.

II. Procedure

To generate a Gaussian pyramid, iterate between these two steps:



a. Smoothing:

use gaussian filter to smooth the image to prevent images from alias artifacts

while down sampling.

1.

```
def _gaussian_kernel1d(sigma, radius):
    """
    Computes a 1D Gaussian convolution kernel.
    """
    sigma2 = sigma * sigma
    x = np.arange(-radius, radius+1)
    phi_x = np.exp(-0.5 / sigma2 * x ** 2)
    phi_x = phi_x / phi_x.sum()


    return phi_x
```

2.

```
def gaussian_filter1d(input, sigma, axis=-1, order=0, output=None,
                      mode="reflect", cval=0.0, truncate=4.0):
    """One-dimensional Gaussian filter.
    """
```

```
sd = float(sigma)
# make the radius of the filter equal to truncate standard deviations
lw = int(truncate * sd + 0.5)

weights = _gaussian_kernel1d(sigma,lw)

return convolve1d(input, weights, axis, output, mode, cval, 0)
```

3.

```
def gaussian_filter(input, sigma, order=0, output=None,
                    mode="reflect", cval=0.0, truncate=4.0):
    """Multidimensional Gaussian filter.
    """
```

```
if len(axes) > 0:
    for axis, sigma, order, mode in axes:
        gaussian_filter1d(input, sigma, axis, order, output,
                          mode, cval, truncate)
        input = output
else:
    output[...] = input[...]
return output
```

All in the _smooth function:

```
def _smooth(image, sigma, mode, cval, multichannel=None):
    """Return image with each channel smoothed by the Gaussian filter."""
```

```
smoothed = np.empty(image.shape, dtype=np.double)

# apply Gaussian filter to all channels independently
if multichannel:
    sigma = (sigma, ) * (image.ndim - 1) + (0, )
gaussian_filter(image, sigma, output=smoothed,
                    mode=mode, cval=cval)
return smoothed
```

Sigma for Gaussian filter. Default is `2 * downscale / 6.0` which corresponds to a filter mask twice the size of the scale factor that covers more than 99% of the Gaussian distribution.

b. down sampling:

Divided block in the image matrix,calculate each block's mean, and leave the mean and remove other elements,by doing so we can't get the downsize version of the input.

```
def downscale_local_mean(image, factors, cval=0, clip=True):
    """Down-sample N-dimensional image by local averaging.
    The image is padded with `cval` if it is not perfectly divisible by the
    integer factors.this function calculates the local mean of
    elements in each block of size `factors` in the input image.
    """
```

Detailed steps:

Ex:

for a matrix A:

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

1.

```
def view_as_blocks(arr_in, block_shape):
    """Block view of the input n-dimensional array (using re-striding).
    Blocks are non-overlapping views of the input array.
```

use **numpy.lib.stride_tricks.as_strided** to create a view into the array with the given shape and strides.

```
if not isinstance(block_shape, tuple):
    #raise TypeError('block needs to be a tuple')
    block_shape=np.asarray(block_shape, dtype=int)
    block_shape=tuple(block_shape)

block_shape = np.array(block_shape)
if (block_shape <= 0).any():
    raise ValueError("'block_shape' elements must be strictly positive")

if block_shape.size != arr_in.ndim:
    raise ValueError("'block_shape' must have the same length "
                     "as 'arr_in.shape'")

arr_shape = np.array(arr_in.shape)
if (arr_shape % block_shape).sum() != 0:
    raise ValueError("'block_shape' is not compatible with 'arr_in'")

# -- restride the array to build the block view
new_shape = tuple(arr_shape // block_shape) + tuple(block_shape)
new_strides = tuple(arr_in.strides * block_shape) + arr_in.strides

arr_out = as_strided(arr_in, shape=new_shape, strides=new_strides)

return arr_out
```

we divide A into 4 block, and we can implement whatever numpy function to process these blocks:

(actually, the 4 blocks are not overlapping like the example below(on 7), we will use padding to fulfil the same calculation result as below.)

```
[ 0,  1,  2,  2,  3,  4] [ 5,  6,  7,  7,  8,  9],
[ 5,  6,  7,  7,  8,  9] [10, 11, 12, 12, 13, 14]]
```

```
                        "image.")
    if image.shape[i] % block_size[i] != 0:
        after_width = math.ceil(block_size[i] - (image.shape[i] % block_size
    else:
        after_width = 0
    pad_width.append((0, after_width))

image = np.pad(image, pad_width=pad_width, mode='constant',
               constant_values=cval)

blocked = view_as_blocks(image, block_size)

return func(blocked, axis=tuple(range(image.ndim, blocked.ndim)),
            **func_kwargs)
```

2.

calculate each block's mean, and leave the mean and remove other elements, if the image size can't perfectly divided by integer, use np.pad to block the result will be:

```
def block_reduce(image, block_size, func=np.sum, cval=0, func_kwargs=None):
    """Downsample image by applying function `func` to local blocks.
    This function is useful for max and mean pooling, for example.
```

we use **np.mean** to calculate the mean of the block

```
return block_reduce(image, factors, np.mean, cval
```

```
array([[3.5, 4. ],
       [5.5, 4.5]])
```

the whole process including gaussian filtering and downscaling is done by below:

```
def resize(image, output_shape, order=None, mode='reflect', cval=0, clip=True,
           preserve_range=False, anti_aliasing=None, anti_aliasing_sigma=None):
    """Resize image to match a certain size.
```

let **anti_aliasing** to be **True** to smooth the image to prevent artifacts.

```
if anti_aliasing:
```

```
image = gaussian_filter(image, anti_aliasing_sigma,
                        cval=cval, mode=ndi_mode)
out=downscale_local_mean(image, factors, cval=0, clip=True)
```

step 1 and step 2 conclusion:

step 1 and step 2 will be implemented in this function, which included downsampling and smoothing:

```
def pyramid_reduce(image, downscale=2, sigma=None, order=1,
                   mode='reflect', cval=0, multichannel=False,
                   preserve_range=False):
    """Smooth and then downsample image.
```

```
smoothed = _smooth(image, sigma, mode, cval, multichannel)
out = resize(smoothed, out_shape, order=order, mode=mode, cval=cval,
             anti_aliasing=True)
```

3.bulid gaussian pyramid layer by layer:

```
def pyramid_gaussian(image, max_layer=-1, downscale=2, sigma=None, order=1,
                     mode='reflect', cval=0, multichannel=False,
                     preserve_range=False):
    """Yield images of the Gaussian pyramid formed by the input image.
    Recursively applies the `pyramid_reduce` function to the image, and yields
    the downscaled images.
```

Note that the first image of the pyramid will be the original, unscaled image. The total number of images is `max_layer + 1`. In case all layers are computed, the last image is either a one-pixel image or the image where the reduction does not change its shape

```
while layer != max_layer:
    layer += 1

    layer_image = pyramid_reduce(prev_layer_image, downscale, sigma, order,
                                 mode, cval, multichannel=multichannel)

    prev_shape = np.asarray(current_shape)
    prev_layer_image = layer_image
    current_shape = np.asarray(layer_image.shape)

    # no change to previous pyramid layer
    if np.all(current_shape == prev_shape):
        break

    yield layer_image
```

The laplacian pyramid:

1.calculate the difference between smoothed and unsmoothed version of previous layers to generate the next laplacian layer:

```
def pyramid_laplacian(image, max_layer=-1, downscale=2, sigma=None, order=1,
                      mode='reflect', cval=0, multichannel=False,
                      preserve_range=False):
    """Yield images of the laplacian pyramid formed by the input image.
    Each layer contains the difference between the downsampled and the
    downsampled, smoothed image::
        layer = resize(prev_layer) - smooth(resize(prev_layer))
    Note that the first image of the pyramid will be the difference between the
    original, unscaled image and its smoothed version. The total number of
    images is `max_layer + 1`. In case all layers are computed, the last image
    is either a one-pixel image or the image where the reduction does not
    change its shape.
```

```
_check_factor(downscale)

# cast to float for consistent data type in pyramid
image = convert_to_float(image, preserve_range)

if sigma is None:
    # automatically determine sigma which covers > 99% of distribution
    sigma = 2 * downscale / 6.0

current_shape = image.shape

smoothed_image = _smooth(image, sigma, mode, cval, multichannel)
yield image - smoothed_image
```

2.build the laplacian pyramid

```
for layer in range(max_layer):

    out_shape =tuple([math.ceil(d / float(downscale)) for d in current_shape])

    if multichannel:
        out_shape = out_shape[:-1]

    resized_image = resize(smoothed_image, out_shape, order=order,
                           mode=mode, cval=cval, anti_aliasing=True)
    smoothed_image = _smooth(resized_image, sigma, mode, cval,
                             multichannel)
    current_shape = np.asarray(resized_image.shape)

    yield resized_image - smoothed_image
```

The magnitude spectrum of gaussian and laplacian:

use fourier transform to gain the image information in frequency domain and show it in the picture.

```
f = np.fft.fft2(image)
fshift = np.fft.fftshift(f)
magnitude_spectrum = 20*np.log(np.abs(fshift))
```

```
def spectrum_gaussian(image, max_layer=-1, downscale=2, sigma=None, order=1,
                      mode='reflect', cval=0, multichannel=False,
                      preserve_range=False):
```

```
def spectrum_laplacian(image, max_layer=-1, downscale=2, sigma=None, order=1,
                       mode='reflect', cval=0, multichannel=False,
                       preserve_range=False):
```

show the result:

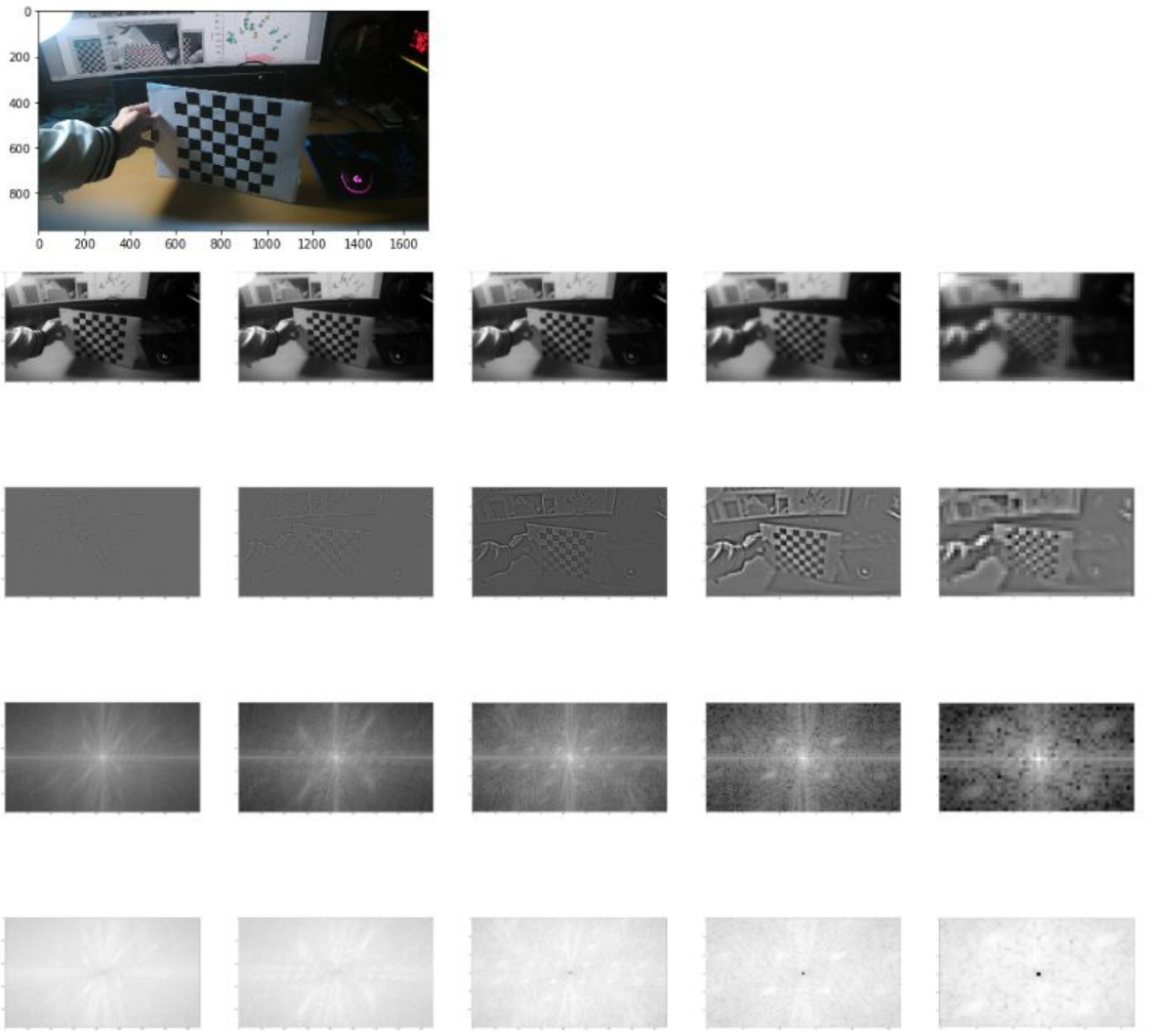
use matplotlib function: **plt.subplots** to properly put all the result together :

```
image1 = plt.imread('hw2_data/task1and2_hybrid_pyramid/data4.jpg')


fig=plt.figure()
plt.imshow(image1)
image1=color.rgb2gray(image1)

g_pyramid =tuple(pyramid_gaussian(image1, downscale=2, multichannel=False))
g_spec=tuple(spectrum_gaussian(image1, downscale=2, multichannel=False))
l_pyramid=tuple(pyramid_laplacian(image1, downscale=2, multichannel=False))
l_spec=tuple(spectrum_laplacian(image1, downscale=2, multichannel=False))


fig,ax=plt.subplots(4,5,figsize=[128,96])
y=0
for g in g_pyramid[1:6]:

    ax[0,y].imshow(g)
    y+=1
y=0
for l in l_pyramid[1:6]:
    ax[1,y].imshow(l)
    y+=1
y=0
for gs in g_spec[1:6]:
    ax[2,y].imshow(gs)
    y+=1
y=0
for ls in l_spec[1:6]:
    ax[3,y].imshow(ls)
    y+=1

plt.show()
```

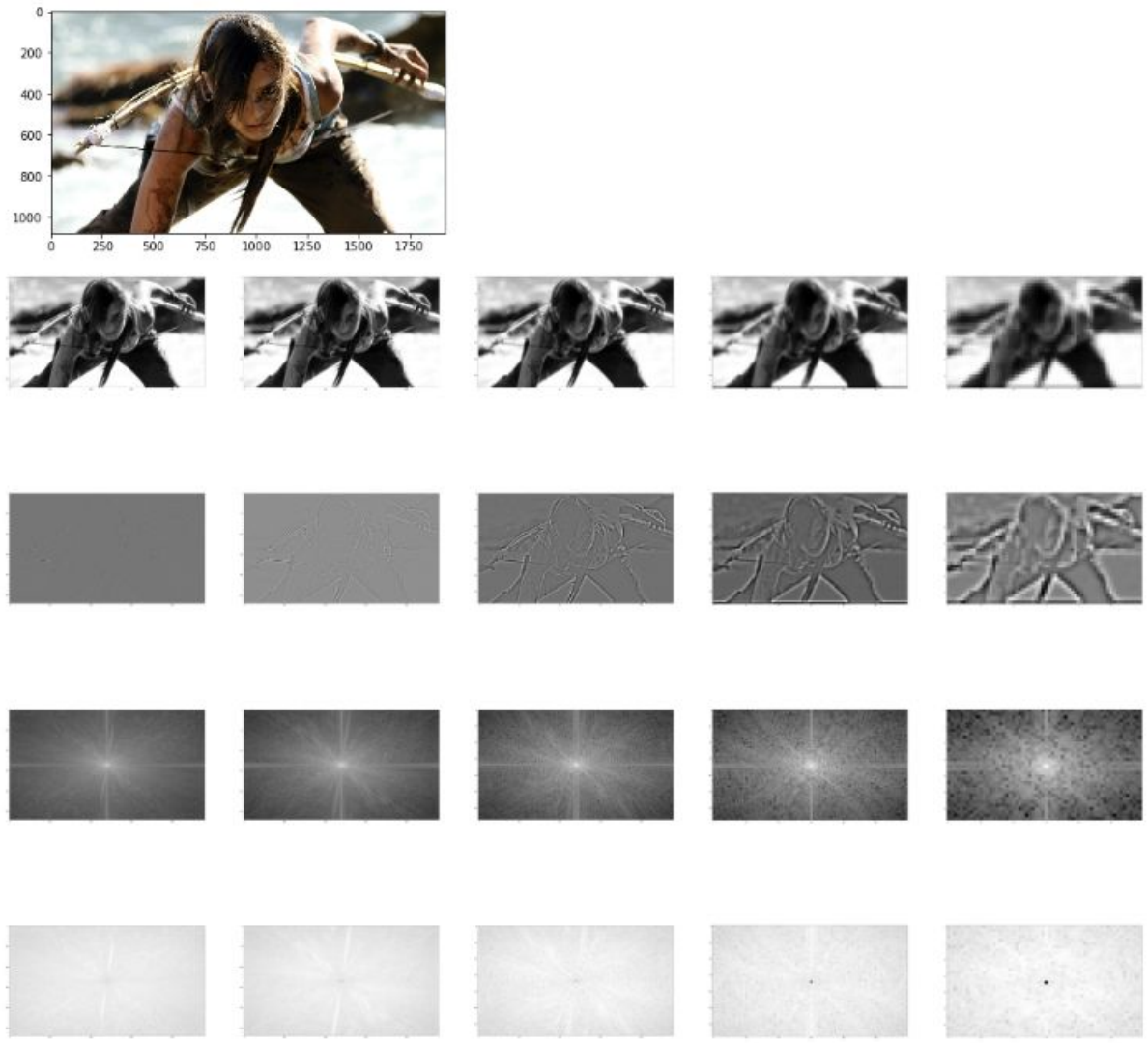
III. Result

From top row to bottom row is **the gaussian pyramid、 the Laplacian pyramid、magnitude spectrum of gaussian、magnitude spectrum of laplacian**, respectively:

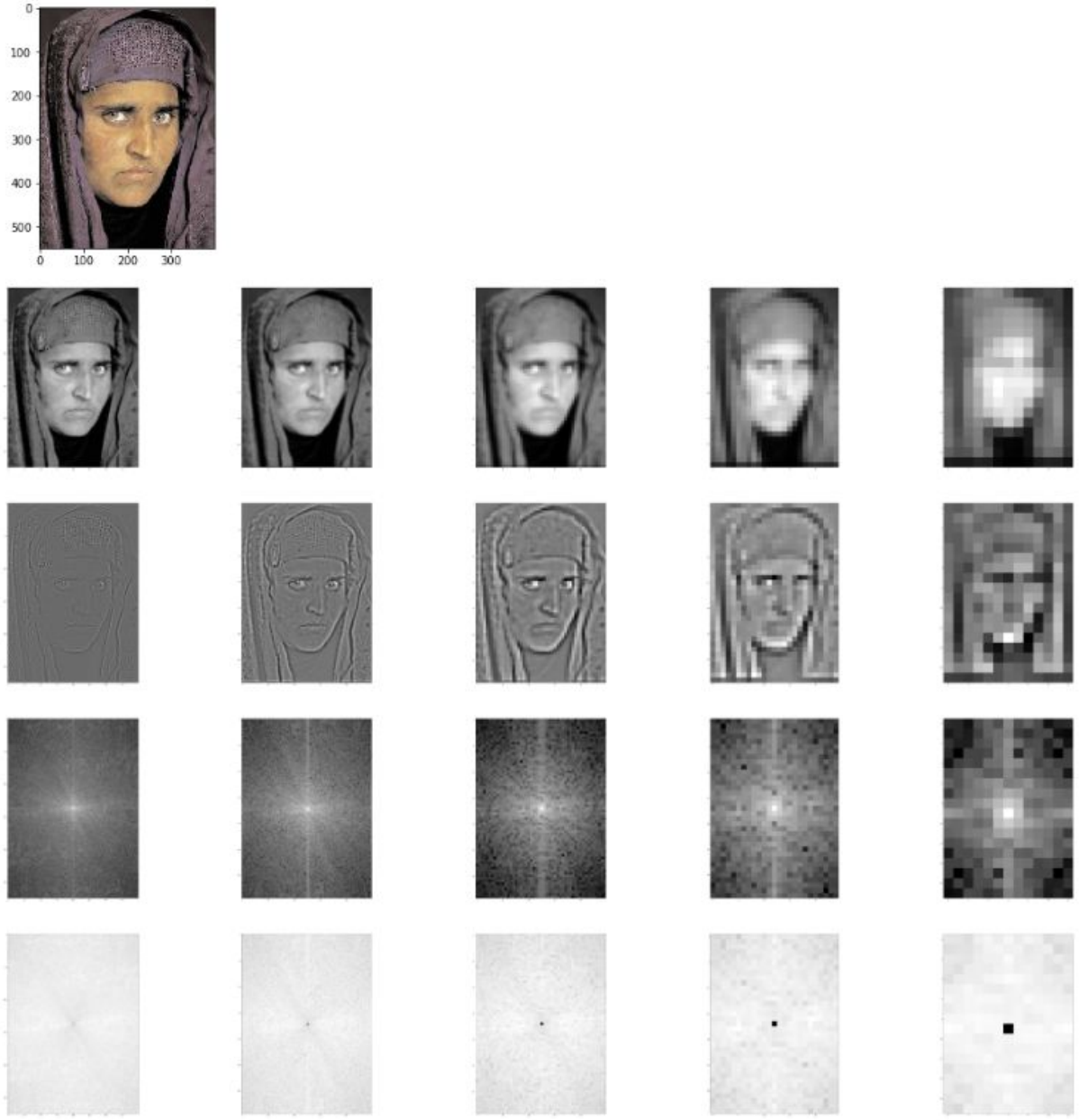
Data4(own image):



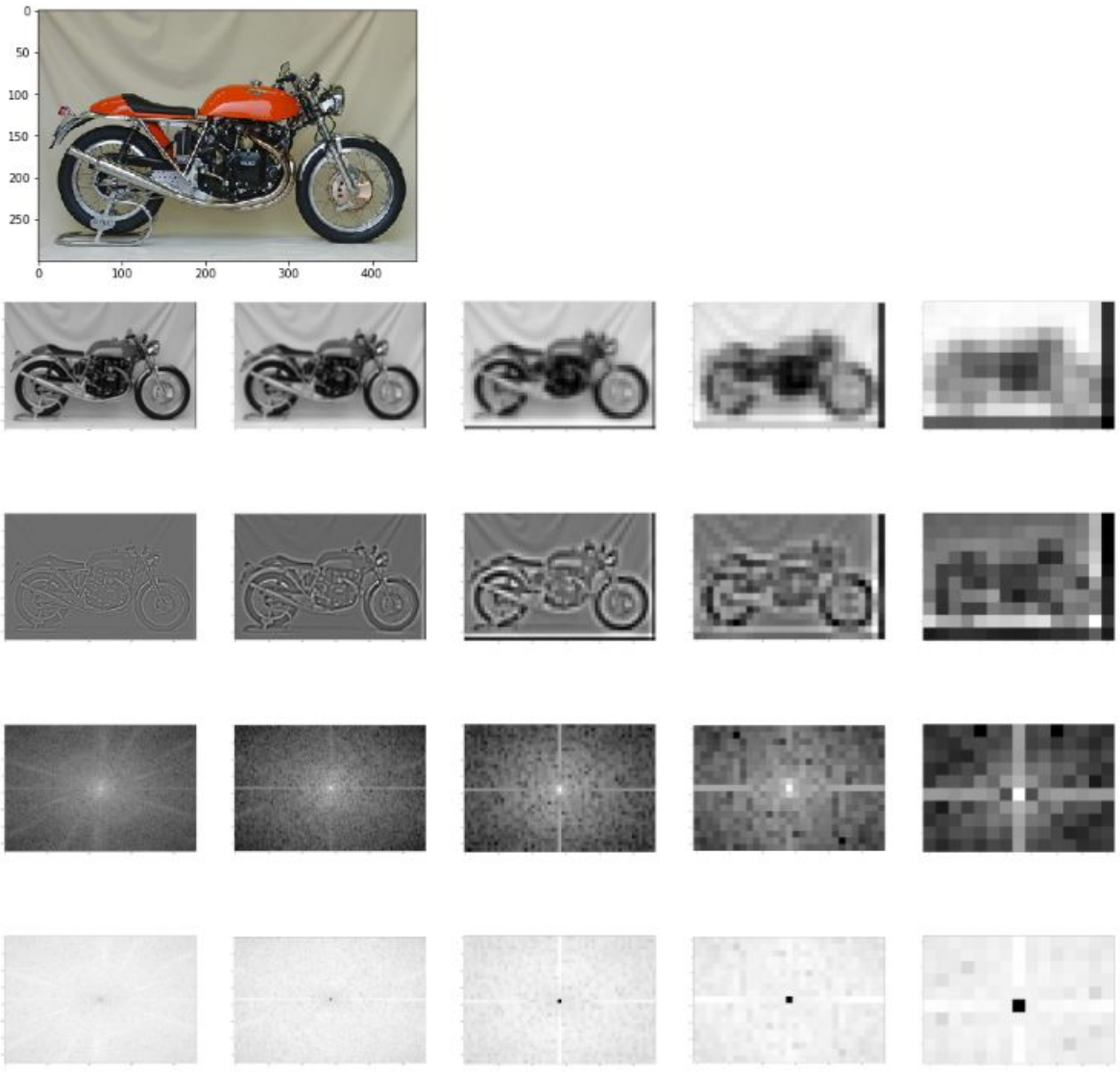
Lara(own image):



0_Afghan_girl_after(from data):



1_motorcycle.bmp(from data):



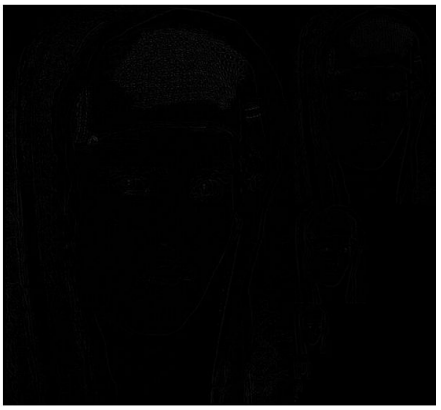
IV. Discussion:

Problem:

1.Meet some type error but we solve it by checking the code thoroughly.

2.display problem:

If we use rgb channel to display the laplacian pyramid the edge and the texture of the image will become almost unrecognisable:



while transforming the image from rgb channel to grayscale channel,if we didn't use **plt.gray()** before, we may get the weird grayscale result like below:

if we use cv2 to read the image while use plt to show it, it will cause weird result because the two functions use different channel order, so we should use the same series of reading and displaying images

Result observation:

The Gaussian pyramid can smooth the image, which has the effect of reducing the image's high-frequency components, you can compare pictures at different scales to prevent the content you are looking for from having different sizes on the picture

The Laplacian pyramid is a versatile data structure with many attractive features for image processing. It represents an image as a series of quasi-bandpassed images, each sampled at successively sparser densities. By appropriately choosing the parameters of the encoding and quantizing scheme, one can substantially reduce the entropy in the representation, and simultaneously extract the feature of the object in the image.

Real-world digital images are in general both scale-variant and highly nonstationary in space. They contain a variety of objects and features (lines, shapes, patterns, edges) at different scales, orientations, and spatial locations; features which the ideal image transformation should independently extract into easily manipulable components .

V. Conclusion:

gaussianGaussian pyramid

In a Gaussian pyramid, subsequent images are weighted down using a Gaussian average (Gaussian blur) and scaled down. Each pixel containing a local average corresponds to a neighborhood pixel on a lower level of the pyramid. This technique is used especially in texture synthesis.

Laplacian pyramid

A Laplacian pyramid is very similar to a Gaussian pyramid but saves the difference image of the blurred versions between each levels. Only the smallest level is not a difference image to enable reconstruction of the high resolution image using the difference images on higher levels. This technique can be used in image compression.

magnitude spectrum:

We can see that amplitude varies drastically in images at the edge points, or noises. So we can say, edges and noises are high frequency contents in an image. If there is no much changes in amplitude, it is a low frequency component.

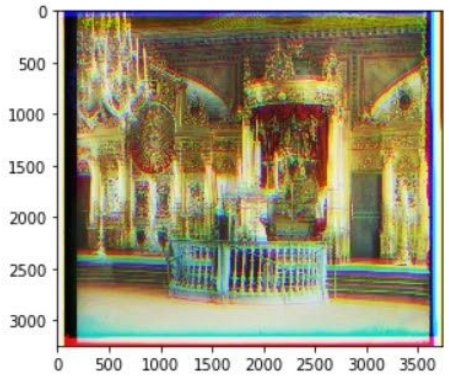
Task 3. Colorizing the Russian Empire

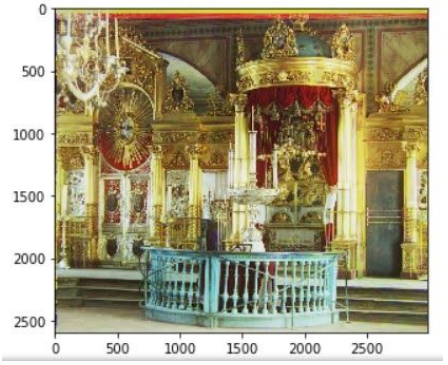
I. Introduction

In this case, our goal is to automatically produce a color image from the digitized Prokudin-Gorskii glass plate images with as few visual artifacts as possible. To produce quality images, we need to crop image borders and use image matching techniques.

The image matching method includes a template matching based on Sum of Squared Differences(SSD) metric and a speed-up trick based on Gaussian pyramid.

without matching: matching:





II. Implementation procedure

First of all, the images include “.jpg” and “.tif” types, we have to use different methods.

What is the difference between “.jpg” and “.tif”? The size of “.tif” is much more large than ‘.jpg’ which means it takes more time in execution.

For jpg:

1. Separate into three pieces

```
def Get_three_channel(img):

    img = sk.img_as_float(img)


    h = int(img.shape[0] / 3)

    # separate color channels
    blue = img[:h]
    green = img[h: 2 * h]
    red = img[2 * h: 3 * h]
    return [blue, green, red]
```

2. Crop 10% to get higher quality

```
def Crop_Img(image, m=0.1):

    h, w = image.shape
    y1, y2 = int(m * h), int((1 - m) * h)
    x1, x2 = int(m * w), int((1 - m) * w)
    return image[y1:y2, x1:x2]
```

3. Align green and red channel to blue channel. Then reconstruct a RGB picture.

```
Shift_result = alignGandRtoB(reconstruct_r, reconstruct_g, reconstruct_b,k_size)
Shift_g = Shift_result[0]
Shift_r = Shift_result[1]
#print(Shift_result)

g_shift_row = Shift_g[0]
g_shift_col = Shift_g[1]
r_shift_row = Shift_r[0]
r_shift_col = Shift_r[1]

reconstruct_g = np.roll(reconstruct_g, g_shift_row, axis=0)
reconstruct_g = np.roll(reconstruct_g, g_shift_col, axis=1)
reconstruct_r = np.roll(reconstruct_r, r_shift_row, axis=0)
reconstruct_r = np.roll(reconstruct_r, r_shift_col, axis=1)

output = np.dstack([reconstruct_r, reconstruct_g, reconstruct_b])
```

detail in align:
In this case, I choose SSD to implement. (Other method such as NCC need much more computation). We search over a window of possible deplacements, score each one with SSD, and find the offset of G and R channels between B. After that, use np.rolling to match three channel and get wonderful images.

For tif:

With the above method, it takes lots of time to handle high resolution image like ‘tif’, so we applied image pyramid to speed up.

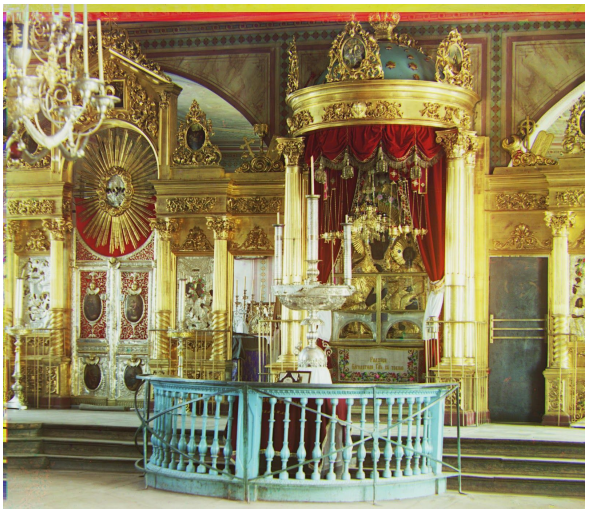
```
while (down_scale >= 1):

    smaller_b = sktr.downscale_local_mean(Three_Channel[0], (down_scale, down_scale))
    smaller_g = sktr.downscale_local_mean(Three_Channel[1], (down_scale, down_scale))
    smaller_r = sktr.downscale_local_mean(Three_Channel[2], (down_scale, down_scale))



    Shift_result = alignGandRtoB(smaller_r, smaller_g, smaller_b,k_size)
    Shift_g = Shift_result[0]
    Shift_r = Shift_result[1]


    g_shift_row += (Shift_g[0] * down_scale)
    g_shift_col += (Shift_g[1] * down_scale)
    r_shift_row += (Shift_r[0] * down_scale)
    r_shift_col += (Shift_r[1] * down_scale)

    Three_Channel[1] = np.roll(Three_Channel[1], Shift_g[0] * down_scale, axis=0)
    Three_Channel[1] = np.roll(Three_Channel[1], Shift_g[1] * down_scale, axis=1)
    Three_Channel[2] = np.roll(Three_Channel[2], Shift_r[0] * down_scale, axis=0)
    Three_Channel[2] = np.roll(Three_Channel[2], Shift_r[1] * down_scale, axis=1)

    down_scale = down_scale // 2
    k_size = k_size // 2
```

Compute the offset through SSD from top to bottom. The images in upper layer are small that take less time in computation and passing the result to lower layer that can save lots of time.

III. Experimental results

tif without speed up: about 502 sec/image

with speed up : about 35 sec/image







IV. Discussion

Execution time is long in this task, image pyramid provide a better result. In coding : when read a image, remember to change the dimension from 3 to 2, due to each

piece in glass plate is only one channel.

V. Conclusion

In this case, we implement the concept from previous task to be fast and efficient. Moreover, we could try to use parallelism to get better result.