**OpenCV学习笔记-傅里叶变换**

**一、**[傅里叶](https://so.csdn.net/so/search?q=%E5%82%85%E9%87%8C%E5%8F%B6&spm=1001.2101.3001.7020)**变换**

傅里叶变换可以将一副图片分解为正弦和余弦两个分量，换言之，它可以将一幅图像从空间域（spatial domain）转换为频域（frequency domain）。这种变换的思想是任何函数可以很精确的接近无穷个sin()函数和cos()函数的和。

我们来梳理一下概念：

**空间域**

一般情况下，空间域的图像是f(x, y) = 灰度级（0-255），形象一点就是一个二维矩阵，每一个坐标对应一个颜色值。

**频率域**

频率：对于图像来说就是指图像颜色值的梯度，即灰度级的变化速度

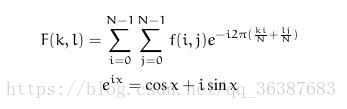
幅度：可以简单的理解为是频率的权，即该频率所占的比例

对于一个正弦信号，如果它的幅度变化很快，我们称之为高频信号，如果变化非常慢，我们称之为低频信号。迁移到图像中，图像哪里的幅度变化非常大呢？边界点或者噪声。所以我们说边界点和噪声是图像中的高频分量（这里的高频是指变化非常快，不是出现的次数多），图像的主要部分集中在低频分量。

由于图像变换的结果原点在边缘部分，不容易显示，所以将原点移动到中心部分。

那么结果便是中间一个亮点朝着周围发散开来，越远离中心位置的能量越低（越暗）。

对于二位图像其傅里叶变换公式如下：



式中 f(i, j)是图像空间域的值，F(k ,l)是频域的值。傅里叶转换的结果是复数，这也显示了傅里叶变换是一幅实数图像和虚数图像叠加或者是幅度图像和相位图像叠加的结果。

**二、Numpy中的傅里叶变换**

函数np.fft.fft2()可以对信号进行频率转换，输出结果是一个复杂的数组。函数的第一个参数是输入图像，要求是灰度格式。第二个参数是可选的，决定输出数组的大小。 输出数组的大小和输入大小一样。如果输出结果比输入图像大，输入图像就需要在进行FFT之前补0.如果输出结果比输入图像小，输入图像就会被切割。

*#快速傅里叶变换算法得到频率分布*

f = np.fft.fft2(img)

*#默认结果中心点位置在左上角，转移到中间*

fshift = np.fft.fftshift(f)

*#fshift是复数，求绝对值结果才是振幅*

magnitude\_spectrum = np.log(np.abs(fshift))

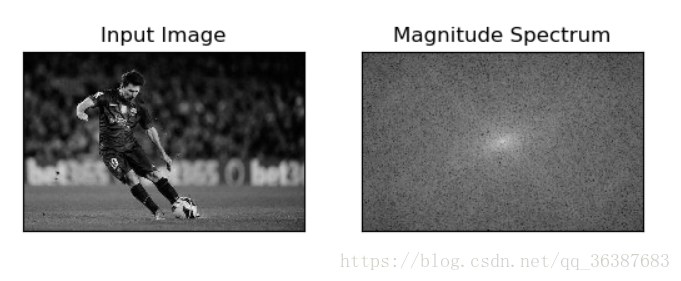
plt.subplot(121),plt.imshow(img, cmap=**'gray'**)

plt.title(**'Input Image'**),plt.xticks([]),plt.yticks([])

plt.subplot(122),plt.imshow(magnitude\_spectrum, cmap=**'gray'**)

plt.title(**'Magnitude Spectrum'**),plt.xticks([]),plt.yticks([])

plt.show()



注意的是，上图其实并没有什么含义，显示出来的可以看成是频域后图像的振幅信息，并没有相位信息，图像的相位  
ϕ = a t a n ( 实部虚部 )，numpy包自带一个angle函数可以直接根据复数的实部和虚部求出角度（默认出来的角度是弧度。）像上述的程序中的f和fshift都是复数，就可以直接用angle直接求解相位，我们和振幅图对比一下：

f = np.fft.fft2(img)

fshift = np.fft.fftshift(f)

*#取绝对值：将复数变成实数*

*#取对数的目的为了将数据变化到0-255*

s1 = np.log(np.abs(fshift))

*#求相位*

s1\_angle = np.angle(fshift)

plt.subplot(221),plt.imshow(img, cmap=**'gray'**),plt.title(**'original'**)

plt.subplot(222),plt.imshow(s1, **'gray'**),plt.title(**'center'**)

plt.subplot(223),plt.imshow(s1\_angle, **'gray'**),plt.title(**'angel'**)

*#逆变换*

f1shift = np.fft.ifftshift(fshift)

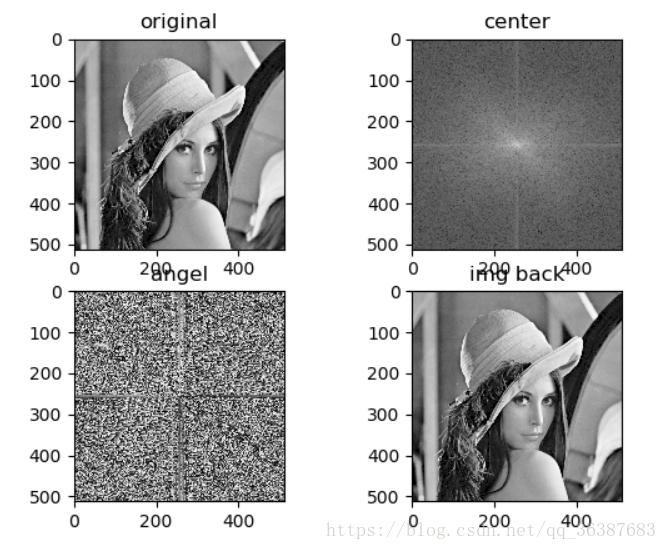
img\_back = np.fft.ifft2(f1shift)

*#出来的复数，无法显示，转成实数*

img\_back = np.abs(img\_back)

plt.subplot(224),plt.imshow(img\_back, **'gray'**),plt.title(**'img back'**)

plt.show()



图三就是图像上每个像素点对应的相位图，其实是毫无规律的，理解就是偏移的角度。

图像变换到频域后，我们怎么转回时域呢？这就是傅里叶逆变换，先反中心化，再逆变换，上面我们一并提到了，并做了一个对比，和原图像一样。

我们说恢复一个频域图像需要图像的振幅和相位，而一个复数也正好包含这些，振幅就是复数的实数部分的绝对值，相位就是  
a t a n ( 实部虚部 )，现在假设我们只用一幅图像的振幅或者相位来将频域内图像恢复到时域会怎么样？

下面给出只用振幅、只用相位、和两者结合的程序：

f = np.fft.fft2(img)

fshift = np.fft.fftshift(f)

*# 取绝对值： 将复数变成实数*

*# 取对数是为了将数据变换到0-255*

s1 = np.log(np.abs(fshift))

plt.subplot(221),plt.imshow(img, **'gray'**),plt.title(**'original'**)

plt.xticks([]), plt.yticks([])

*#-------------------------------------*

*#逆变换 -- 取绝对值就是振幅*

f1shift = np.fft.ifftshift(np.abs(fshift))

img\_back = np.fft.ifft2(f1shift)

img\_back = np.abs(img\_back)

plt.subplot(222),plt.imshow(img\_back, **'gray'**),plt.title(**'only Amplitude'**)

plt.xticks([]),plt.yticks([])

*#------------------------------------*

*#逆变换 -- 取相位*

f2shift = np.fft.ifftshift(np.angle(fshift))

img\_back = np.fft.ifft2(f2shift)

*#结果是复数，无法显示*

img\_back = np.abs(img\_back)

plt.subplot(223), plt.imshow(img\_back, **'gray'**), plt.title(**'only phase'**)

plt.xticks([]), plt.yticks([])

*#------------------------------------*

*#逆变换--将两者结合看看*

s1 = np.abs(fshift) *# 取振幅*

s1\_angle = np.angle(fshift) *# 取相位*

s1\_real = s1\*np.cos(s1\_angle) *# 取实部*

s1\_imag = s1\*np.sin(s1\_angle) *# 去虚部*

s2 = np.zeros(img.shape, dtype=complex)

s2.real = np.array(s1\_real) *# 重新赋值给s2*

s2.imag = np.array(s1\_imag)

*#对新的进行逆变换*

f3shift = np.fft.ifftshift(s2)

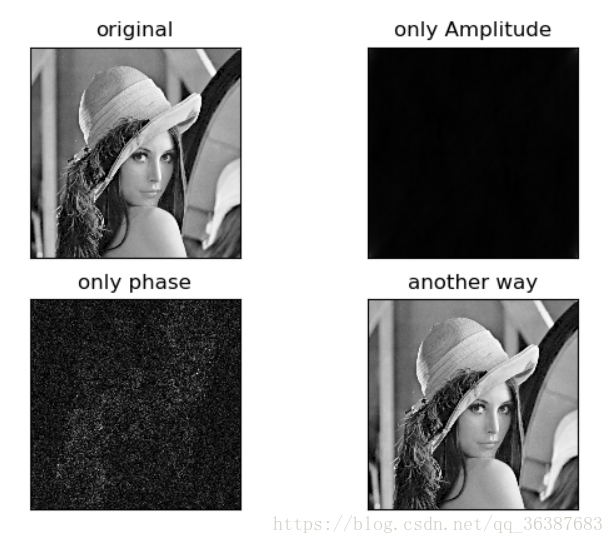
img\_back = np.fft.ifft2(f3shift)

img\_back = np.abs(img\_back)

plt.subplot(224), plt.imshow(img\_back, **'gray'**), plt.title(**'another way'**)

plt.xticks([]), plt.yticks([])

plt.show()



可以看到，仅仅振幅的恢复图什么都看不来，仅仅的相位图虽然有东西，但是也看不出来什么，最后是把振幅和相位分别作为频域内复数的实部和虚部，得到的恢复图才和原来一样。

基于此，我们来做一个实验，假设有两幅图像，将这两幅图像进行傅里叶变换到频域，然后用一幅图像的振幅和另一幅图像的相位结合生成一幅新的图像，然后我们再转回时域，那么生成的图像更像取振幅的图像，还是取相位的图像呢？我们两张大小一样的图片：

img\_1 = cv.imread(**'img/WindowsLogo.jpg'**, 0)

img\_2 = cv.imread(**'img/LinuxLogo.jpg'**, 0)

plt.subplot(221), plt.imshow(img\_1, **'gray'**), plt.title(**'original\_1'**)

plt.xticks([]), plt.yticks([])

plt.subplot(222), plt.imshow(img\_2, **'gray'**), plt.title(**'original\_2'**)

plt.xticks([]), plt.yticks([])

*#----------------------------*

f1 = np.fft.fft2(img\_1)

f1shift = np.fft.fftshift(f1)

f1\_A = np.abs(f1shift) *# 取振幅*

f1\_P = np.angle(f1shift) *# 取相位*

*#-----------------------------*

f2 = np.fft.fft2(img\_2)

f2shift = np.fft.fftshift(f2)

f2\_A = np.abs(f2shift) *# 取振幅*

f2\_P = np.angle(f2shift) *# 取相位*

*#--------图1的振幅--图2的相位-----*

img\_new\_1 = np.zeros(img\_1.shape, dtype=complex)

img\_new\_1\_real = f1\_A \* np.cos(f2\_P) *# 取实部*

img\_new\_1\_imag = f1\_A \* np.sin(f2\_P) *# 取虚部*

img\_new\_1.real = img\_new\_1\_real

img\_new\_1.imag = img\_new\_1\_imag *# 对新图像赋值*

f3shift = np.fft.ifftshift(img\_new\_1) *# 对新的图像进行逆变换*

img\_new\_1\_back = np.fft.ifft2(f3shift)

*#结果是复数，无法显示*

img\_new\_1\_back = np.abs(img\_new\_1\_back)

plt.subplot(223), plt.imshow(img\_new\_1\_back, **'gray'**), plt.title(**'img1\_A img2\_P'**)

plt.xticks([]), plt.yticks([])

*#-------图1的相位--图2的振幅---------*

img\_new\_2 = np.zeros(img\_2.shape, dtype=complex)

img\_new\_2\_real = f2\_A \* np.cos(f1\_P) *# 取实部*

img\_new\_2\_imag = f2\_A \* np.sin(f1\_P) *# 取虚部*

img\_new\_2.real = img\_new\_2\_real

img\_new\_2.imag = img\_new\_2\_imag *#对新图像赋值*

f4shift = np.fft.ifftshift(img\_new\_2) *# 对新的图像进行逆变换*

img\_new\_2\_back = np.fft.ifft2(f4shift)

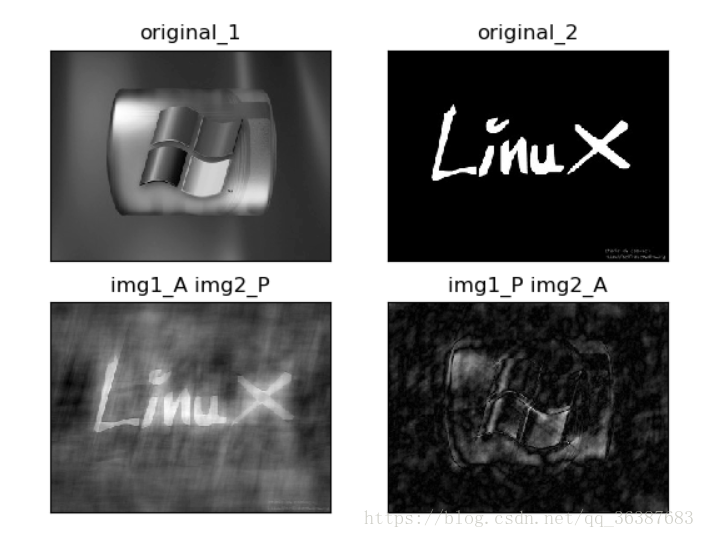
*#结果是复数，无法显示*

img\_new\_2\_back = np.abs(img\_new\_2\_back)

plt.subplot(224), plt.imshow(img\_new\_2\_back, **'gray'**), plt.title(**'img1\_P img2\_A'**)

plt.xticks([]), plt.yticks([])

plt.show()



图三是图一的振幅加图二的相位，图四是图一的相位加图二的振幅。很明显的可以看出，新图像由相位决定。为什么会这样呢？可以理解为振幅不过是描述图像灰度的亮度，占用谁的振幅不过使得结果哪部分偏亮或者哪部分偏暗而已。而图像最终是什么样子，由相位来决定。相位描述的是一个方向，只要方向正确了，那么最终的结果离你的目的也不会远。

现在我们可以进行频域变换了，例如高通滤波和重建图像（DFT的逆变换）。比如我们使用一个60 x 60的矩形窗口对图像进行掩模操作去除低频分量。然后再使用函数np.fft.ifftshift()进行逆平移操作，现在直流分量又回到了左上角，然后使用函数np.ifft2()进行FFT逆变换。同样又得到一堆复杂的数字，我们取绝对值：

rows, cols = img.shape

*#宽高的1/2 取中心坐标*

crow, ccol = int(rows / 2), int(cols / 2)

*#60 \* 60 的矩形窗口，去除低频分量*

fshift[crow-30:crow+30,ccol-30:ccol+30] = 0

*#进行逆平移操作，直流分量又回到左上角*

f\_shift = np.fft.ifftshift(fshift)

*#进行FFT逆变换*

img\_back = np.fft.ifft2(f\_shift)

*#取绝对值*

img\_back = np.abs(img\_back)

plt.subplot(131),plt.imshow(img, cmap=**'gray'**)

plt.title(**'Input Image'**),plt.xticks([]),plt.yticks([])

plt.subplot(132), plt.imshow(img\_back, cmap=**'gray'**)

plt.title(**'Image after HPF'**), plt.xticks([]), plt.yticks([])

plt.subplot(133), plt.imshow(img\_back)

plt.title(**'Result in JET'**), plt.xticks([]), plt.yticks([])

plt.show()

效果如下：



上图的结果显示高通滤波其实是一种边界检测操作。我们把中心区域的低频分量全部去除了，只留下了一些高频分量，高频分量是什么？边界和噪声啊，所以剩下的就是边界了。

下面再来试试低通滤波器什么效果，构造个低通滤波器也很简单，把上述模板中的1变成0，0变成1，其实就是低通了：

plt.subplot(131), plt.imshow(img, **'gray'**), plt.title(**'original'**)

plt.xticks([]), plt.yticks([])

rows, cols = img.shape

crow, ccol = int(rows/2), int(cols/2)

*# -------高通滤波---------*

mask = np.ones(img.shape, np.uint8)

mask[crow - 30:crow + 30, ccol - 30:ccol + 30] = 0

f1 = np.fft.fft2(img)

f1shift = np.fft.fftshift(f1)

f1shift = f1shift \* mask

f2shift = np.fft.ifftshift(f1shift) *# 对新得到图像进行逆变换*

img\_new = np.fft.ifft2(f2shift)

img\_new = np.abs(img\_new)

plt.subplot(132), plt.imshow(img\_new, **'gray'**), plt.title(**'Highpass'**)

plt.xticks([]), plt.yticks([])

*# --------低通滤波---------*

mask2 = np.zeros(img.shape, np.uint8)

mask2[crow - 30:crow + 30, ccol - 30:ccol + 30] = 1

f1 = np.fft.fft2(img)

f1shift = np.fft.fftshift(f1)

f1shift = f1shift \* mask2

f2shift = np.fft.ifftshift(f1shift) *# 对新得到图像进行逆变换*

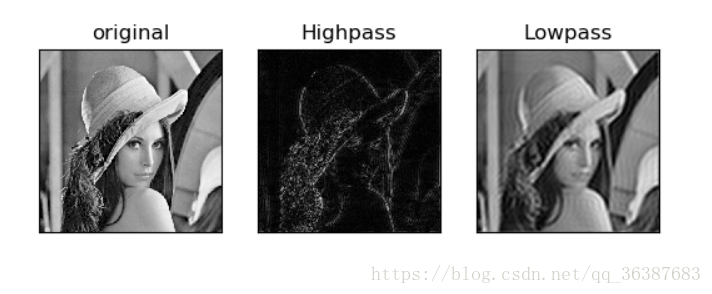
img\_new = np.fft.ifft2(f2shift)

img\_new = np.abs(img\_new)

plt.subplot(133), plt.imshow(img\_new, **'gray'**), plt.title(**'Lowpass'**)

plt.xticks([]), plt.yticks([])

plt.show()



可以看出低通滤波后图像的轮廓模糊了，从原理上看，图像的主要信息都集中在低频上，所以低通滤波器相当于执行了迷糊操作。

除了高通、低通滤波器，还有其他滤波器，如高斯滤波器，butterworth滤波器等。还有就是把高通低通都结合一部分到一个模板上形成的带通滤波器，这在一些场合会很有用，还是以理想的带通滤波器试验下：

plt.subplot(121), plt.imshow(img, **'gray'**), plt.title(**'original'**)

plt.xticks([]), plt.yticks([])

*#-----------------------------*

rows, cols = img.shape

crow, ccol = int(rows/2), int(cols/2)

*#高通滤波器*

mask1 = np.ones(img.shape, np.uint8)

mask1[crow-8:crow+8, ccol-8:ccol+8] = 0

*#低通滤波器*

mask2 = np.zeros(img.shape, np.uint8)

mask2[crow-80:crow+80, ccol-80:ccol+80] = 1

*#带通滤波器*

mask = mask1 \* mask2

*#----------------------------------*

f1 = np.fft.fft2(img)

f1shift = np.fft.fftshift(f1)

f1shift = f1shift \* mask

*#对新得到的图像进行逆变换*

f2shift = np.fft.ifftshift(f1shift)

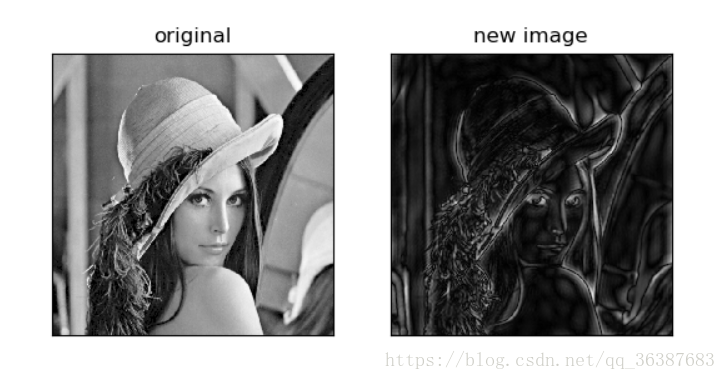
img\_new = np.fft.ifft2(f2shift)

img\_new = np.abs(img\_new)

plt.subplot(122), plt.imshow(img\_new, **'gray'**), plt.title(**'new image'**)

plt.xticks([]), plt.yticks([])

plt.show()



这就是带通的效果，即可以保留一部分低频，也可以保留一部分高频，至于保留多少，就要视情况而定。

在文章的最后，我们对一些算子做一下傅里叶变换，看看他们属于HPF还是LPH

*#对一些算子进行比较*

**def** demo5():

mean\_filter = np.ones((3, 3))

x = cv.getGaussianKernel(5, 10)

guassian = x \* x.T

scharr = np.array([[-3, 0, -3],

[-10, 0, -10],

[-3, 0, 3]])

sobel\_x = np.array([[-1, 0, 1],

[-2, 0, -2],

[-1, 0, 1]])

sobel\_y = np.array([[-1, -2, -1],

[0, 0, 0],

[1, 2, 1]])

laplacian = np.array([[0, 1, 0],

[1, -4, 1],

[0, 1, 0]])

*#我看有两个版本的laplacian算子，这个博客给了解释https://www.cnblogs.com/german-iris/p/4840647.html*

filters = [mean\_filter, guassian, laplacian, sobel\_x, sobel\_y, scharr]

filter\_name = [**'mean\_filter'**, **'guassian'**, **'laplacian'**, **'sobel\_x'**, **'sobel\_y'**, **'scharr'**]

fft\_filters = [np.fft.fft2(x) **for** x **in** filters]

fft\_shift = [np.fft.fftshift(x) **for** x **in** fft\_filters]

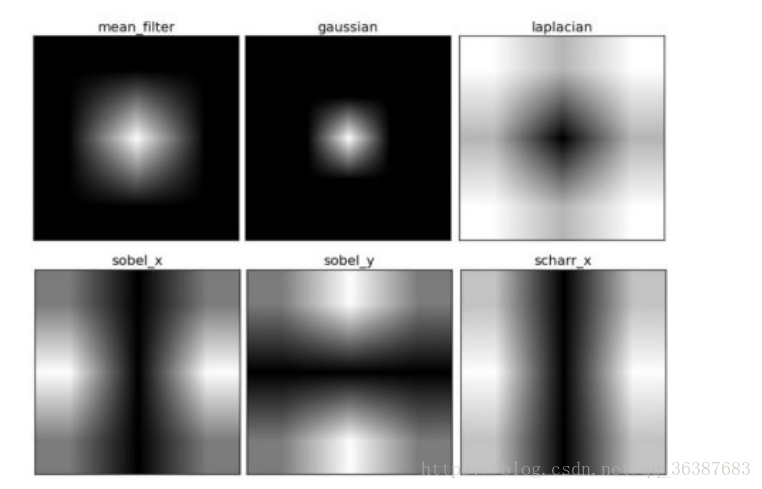
mag\_spectrum = [np.log(np.abs(x)+1) **for** z **in** fft\_shift]

**for** i **in** range(6):

plt.subplot(2, 3, i+1),plt.imshow(mag\_spectrum[i], **'gray'**)

plt.title(filter\_name[i]),plt.xticks([]),plt.yticks([])

plt.show()



从结果我们可以看出：mean\_filter、guassian算子是LPH，laplacian、sobel、scharr算子是HPF