

（一）实现方法

1. 人脸检测

因为 opencv 自带人脸特征数据，可以简单借助 opencv 中的 CascadeClassifier 函数实现检测，需要指定使用的人脸特征数据的路径，我使用的是 haarcascade_frontalface_alt2.xml，可在该网站下载现有的.xml 文件：<https://github.com/opencv/opencv/tree/master/data/haarcascades>

alalek fix files permissions		
..		
haarcascade_eye.xml		some attempts to tune the performance
haarcascade_eye_tree_eyeglasses.xml		some attempts to tune the performance
haarcascade_frontalcatface.xml		fix files permissions
haarcascade_frontalcatface_extended.xml		fix files permissions
haarcascade_frontalface_alt.xml		some attempts to tune the performance
haarcascade_frontalface_alt2.xml		some attempts to tune the performance
haarcascade_frontalface_alt_tree.xml		some attempts to tune the performance

.xml 属于分类器文件，作用是根据已有类别的训练数据，判断一个新样本是否同属该类别，如人脸、人体、车牌、猫脸等会对应不同的.xml。

定义 face_detect 人脸检测函数：

```
1. def face_detect(img_path):
2.     color = (0, 255, 0)
3.     img_bgr = cv2.imread(img_path)
4.     classifier = cv2.CascadeClassifier(CADES_PATH)
5.     img_gray = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2GRAY)
6.     facerects = classifier.detectMultiScale(img_gray)
7.     if len(facerects) > 0:
8.         for rect in facerects:
9.             x, y, w, h = rect
10.            if w > 200:
11.                cv2.rectangle(img_bgr, (x, y), (x + w, y + h), color, 2)
```

以下是算法思路：

a. 导入 cv2 库

- b. 读入要检测的图像
- c. 创建人脸检测器
- d. 对图像进行灰度处理，进行人脸检测。(提高人脸检测的速度)
- e. 根据检测到的人脸数据，用矩形框标注人脸
- f. 显示标注人脸后的图像

选择 CascadeClassifier 函数算法的原因在于追求简洁明了，提高速度，可以直接借助 opencv，不用使用深度学习预处理数据以及多次训练。

2. 滤波器

2.1 引导滤波

引导滤波思想在于用一张引导图像产生权重，从而对输入图像进行处理，实际上是假设某像素 k 为中心的窗口存在局部线性关系，通过对局部线性关系求导可以判断哪些边缘需要保留，给出的 a, b 两个参数则是以像素 k 为窗口的周围的权重的均值。基于此原理，自己实现了一个引导滤波去噪的算法。

2.1.1 定义单通道函数

首先定义 my_guidedFilter_oneChannel 函数，用于单通道图像（灰度图）的引导滤波函数：

```
1. def my_guidedFilter_oneChannel(srcImg, guidedImg, rad=9, eps=0.01):
```

以下是参数解释：

srcImg: 输入图像，为单通道图像；

guidedImg: 引导图像，为单通道图像，尺寸与输入图像一致；

rad: 滤波器大小，应该保证为奇数，默认值为 9；

eps : 防止 a 过大的正则化参数。

2.1.2 归一化与求均值

然后转换数据类型，将其归一化：

```
1. srcImg = srcImg / 255.0
2. guidedImg = guidedImg / 255.0
```

然后再求引导图像和输入图像的均值图，其中求图像的均值滤波图采用 OpneCV 的 boxfliter 函数

```
1. P_mean = cv2.boxFilter(srcImg, -1, (rad, rad), normalize=True)
2. I_mean = cv2.boxFilter(guidedImg, -1, (rad, rad), normalize=True)
3.
```

```

4. I_square_mean = cv2.boxFilter(np.multiply(guidedImg, guidedImg), -
    1, (rad, rad), normalize=True)
5. I_mul_P_mean = cv2.boxFilter(np.multiply(srcImg, guidedImg), -
    1, (rad, rad), normalize=True)
6.
7. var_I = I_square_mean - np.multiply(I_mean, I_mean)
8. cov_I_P = I_mul_P_mean - np.multiply(I_mean, P_mean)
9.
10. a = cov_I_P / (var_I + eps)
11. b = P_mean - np.multiply(a, I_mean)
12.
13. a_mean = cv2.boxFilter(a, -1, (rad, rad), normalize=True)
14. b_mean = cv2.boxFilter(b, -1, (rad, rad), normalize=True)
15.
16. dstImg = np.multiply(a_mean, guidedImg) + b_mean

```

2.1.3 定义三通道函数

定义 `my_guidedFilter_threeChannel` 函数，用于三通道图像（RGB 彩色图）的引导滤波函数：

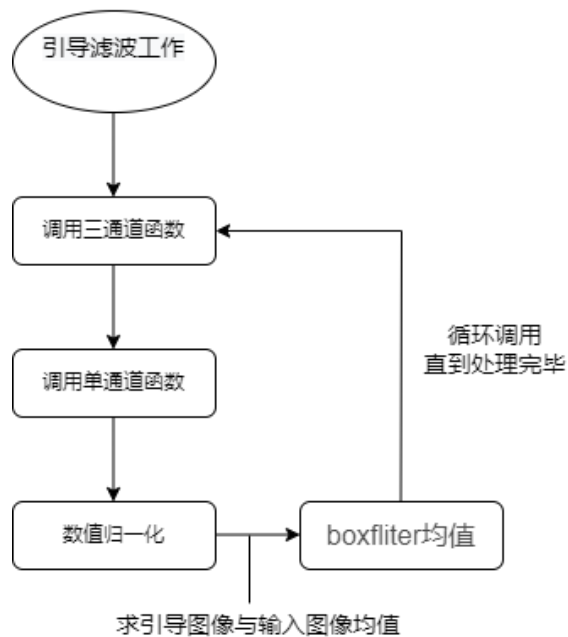
```

1. def my_guidedFilter_threeChannel(srcImg, guidedImg, rad=9, eps=0.01):
2.     img_shape = np.shape(srcImg)
3.
4.     dstImg = np.zeros(img_shape, dtype=float)
5.
6.     for ind in range(0, img_shape[2]):
7.         dstImg[:, :, ind] = my_guidedFilter_oneChannel(srcImg[:, :, ind],
8.                                                         guidedImg[:, :, ind]
9.                                                         , rad, eps)
10.
11.     dstImg = dstImg.astype(np.uint8)
12.
13.     return dstImg

```

以上返回的 `dstImg` 即是引导滤波图。

算法的具体流程图如下：



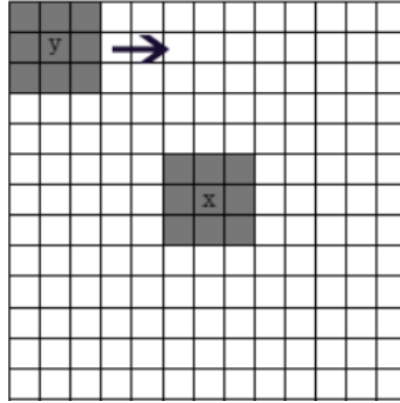
因为引导滤波可以很好地克服双边滤波中出现的梯度翻转的现象，在滤波后图像的细节上更优。而双边滤波对于梯度变化大的地方，由于周围没有相似的像素，高斯函数的权重不稳定，导致最终梯度出现反转现象。而且引导滤波线性的计算量，可以显著提高处理的效率，比较适合我们这种轻量级的美颜应用。

2.2 NLM 滤波

某些基于邻域像素的滤波方法，基本上只考虑了有限窗口范围内的像素灰度值信息，没有考虑该窗口范围内像素的统计信息如方差，也没有考虑整个图像的像素分布特性。

因此选择 NLM (Non-Local Means) 算法，它利用了整幅图像来去噪，以图像块为单位在图像中寻找相似区域，再对这些区域求平均，能够比较好的去掉图像中存在的高斯噪声，基本思想是：当前像素的估计值由图像中与它具有相似邻域结构的像素加权平均得到。

下图是 NL-means 算法执行过程，大窗口是以目标像素为中心的搜索窗口，两个灰色小窗口分别是以 x, y 为中心的邻域窗口。其中以 y 为中心的邻域窗口在搜索窗口中滑动，通过计算两个邻域窗口间的相似程度为 y 赋以权值：



传统 NLM 算法:

opencv 自带有传统的 NLM 算法，例如:

`cv2.fastNLMMeansDenoising()` 使用对象为灰度图像

`cv2.fastNLMMeansDenoisingColored()` 使用对象为彩色图像 等等

但是它是基于对应像素点间的误差，并未考虑到人眼的视觉特性。

自定义 NLM 算法:

考虑到人眼的视觉特性，为了适应人眼的主观印象，下面自定义一个 NLM 算法:

2.2.1 自定义 NLM 算法思想:

设含噪声图像为 v ，去噪后的图像为 u 。 u 中像素点 x 处的灰度值通过如下方式得到:

$$\tilde{u}(x) = \sum_{y \in I} w(x, y) * v(y)$$

其中权值 w 表示像素点 x 和 y 间的相似度，它的值由以 $V(x)$ 、 $V(y)$ 为中心的矩形邻域间的距离决定:

$$w(x, y) = \frac{1}{Z(x)} \exp\left(-\frac{\|V(x) - V(y)\|^2}{h^2}\right)$$

其中

$$\|V(x) - V(y)\|^2 = \frac{1}{d^2} \sum_{\|z\|_{\infty} \leq ds} \|v(x+z) - v(y+z)\|^2$$

$$Z(x) = \sum_y \exp\left(-\frac{\|V(x) - V(y)\|^2}{h^2}\right)$$

$Z(x)$ 为归一化系数， h 为平滑参数，控制高斯函数的衰减程度。 h 越大，高斯函数变化越平缓，去噪水平越高，但同时也会导致图像越模糊。 h 越小，边缘细节成分保持得越多，但会残留过多

的噪声点。 h 的具体取值应当以图像中的噪声水平为依据。

2.2.2 自定义 NLM 算法实现:

1) 给图像加高斯噪声 (可不添加, 起到增加颗粒感作用)

加高斯噪声就是在原图像矩阵上面加一个符合高斯或者叫正态分布特征的矩阵。

如果我们的目标矩阵是一个 $r \times c$ 的矩阵, 要生成一个均值是 $mean$, 标准差 $sigma$ 的随机噪声矩阵, 那么可以这样:

```
1. sigma*np.random.randn(r,c)+mean, #输入是两个参数, 一个mean, 一个sigma
2. sigma*np.random.standard_normal((r,c)) #输入是一个tuple, 里面包含两个元素。
3. numpy.random.normal(mean,sigma, size=(r,c)) #mean, sigma 和大小都作为输入放进函数
```

生成噪声图像的代码:

```
1. def double2unit8(I, L, ratio=1.0, sigma=20.0):
2.     I = I.astype(np.float64) # 转换成float 形式便于计算
3.     noise = np.random.randn(*I.shape) * sigma # 生成的时形状与I 相同, 均值为0, 标准差为sigam 的随机高斯噪声矩阵
4.     noisy = I + noise # 噪声图像
5.     return np.clip(np.round(noisy * ratio), 0, 255).astype(L.dtype) # 转换回图像的unit8 均值
```

2) 生成高斯核

```
1. def make_kernel(f): # 计算得到一个高斯核, 用于后续的计算
2.     kernel = np.zeros((2 * f + 1, 2 * f + 1))
3.     for d in range(1, f + 1):
4.         kernel[f - d:f + d + 1, f - d:f + d + 1] += (1.0 / ((2 * d + 1) ** 2))
5.     return kernel / kernel.sum()
```

3) NLM

```
1. def NLmeansfilter(I,L,h_=10,templateWindowSize=5,searchWindowSize=11):
2.     I=I.astype(np.float64)
3.     f=int(templateWindowSize/2)
4.     t=int(searchWindowSize/2)
5.     height,width=I.shape[:2] # 利用ndarray 的索引得到长宽
6.     padLength=t+f
```

```

7.     I2=np.pad(I,padLength,'symmetric')    # symmetric—表示对称填充，每侧填充
      均为padLength
8.     kernel=make_kernel(f)
9.     h=(h_**2)
10.    I_=I2[padLength-f:padLength+f+height,padLength-
      f:padLength+f+width]    # 在原图大小的基础上，上下左右均延拓了f 大小的图，固定的领
      域窗口y
11.
12.    average=np.zeros(I.shape)
13.    sweight=np.zeros(I.shape)
14.    wmax=np.zeros(I.shape)
15.    for i in range(-t, t+1):
16.        for j in range (-t,t+1):
17.            if i==0 and j==0:
18.                continue
19.            I2_=I2[padLength+i-f:padLength+i+f+height,padLength+j-
      f:padLength+j+f+width]    # 移动的领域窗口x
20.            w = np.exp(-cv2.filter2D((I2_ - I_)**2, -
      1, kernel)/h)[f:f+height, f:f+width]    # 计算权重w，保持形状
21.            sweight += w    # 将所有的权值相加
22.            wmax=np.maximum(wmax,w)    # 求最大
23.            average += (w*I2_[f:f+height,f:f+width])    # 得到权值与窗口x 相乘的值
      并求和
24.    I2=(average+wmax*I)/(sweight+wmax)    # 权值最大的是自身
25.    return np.clip(np.round(I1),0,255).astype(L.dtype)

```

2.2.2 自定义 NLM vs 传统 NLM:

对一张原图进行处理，生成四张图像：



原图



噪声图



传统 NLM 图



自定义 NLM 图

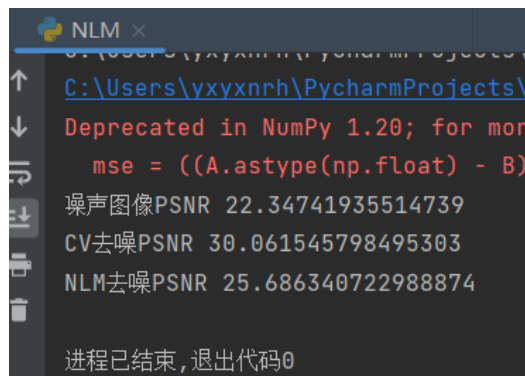
为了比较自定义的 NLM 算法与传统算法的性能，借助 PSNR（峰值信噪比，即峰值信号的能量与噪声的平均能量之比，越大表示图片质量越好）指标，定义 `psnr` 函数

```
1. def psnr(A, B):  
2.     val = 255  
3.     mse = ((A.astype(np.float) - B) ** 2).mean()  
4.     return 10 * np.log10((val * val) / mse)
```

对超过 30 张图片进行测试，输出生成图像的平均 PSNR 值：

```
1. print('噪声图像 PSNR', psnr(I, I1))  
2. print('CV 去噪 PSNR', psnr(I, R2))  
3. print('NLM 去噪 PSNR', psnr(I, R1))
```

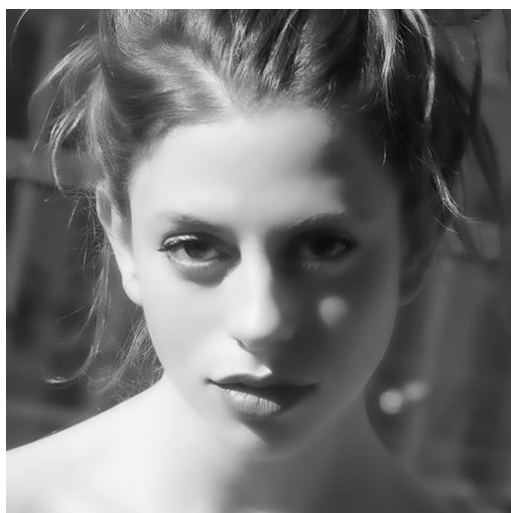
以下是输出结果：



```
NLM x  
C:\Users\yxyxnrh\PycharmProjects\  
Deprecated in NumPy 1.20; for mor  
mse = ((A.astype(np.float) - B)  
噪声图像PSNR 22.34741935514739  
CV去噪PSNR 30.061545798495303  
NLM去噪PSNR 25.686340722988874  
进程已结束,退出代码0
```


凭肉眼来看，我觉得自定义的 NLM 效果更好一点，但是从 PSNR 结果上来看 NLM 虽然起到了去噪的效果，但效果还是不如 CV 自带的传统 NLM 函数。这是因为 PSNR 是最普遍和使用最为广泛的一种图像客观评价指标，是基于对应像素点间的误差，即基于误差敏感的图像质量评价。但由于并未考虑到人眼的视觉特性，因而经常出现评价结果与人的主观感觉不一致的情况。

2.3 两种滤波器算法比较



引导滤波图



自定义 NLM 图

由上图效果可知，比起引导滤波，NLM 特点在于较好保持图像整体清晰度。缺点在于计算起来很慢，如果图像像素点比较多，而且计算的时候取的框还比较大的话，那么计算时间就会过长。

引导滤波优点在于既可以去噪（磨平）和保留边缘，而且时间上快，效率高，时间复杂度为 $O(N)$ ， N 是像素个数，也就是说与掩膜窗口尺寸无关。

3. 皮肤检测

基于 RGB 空间的肤色检测，RGB 是生活中常见的颜色模型，可以通过简单的阈值计算获得肤色数据，能有效的降低程序的执行时间。

只需满足以下式子：

$R > 95$ And $G > 40$ And $B > 20$ And $R > G$ And $R > B$ And $\text{Max}(R, G, B) - \text{Min}(R, G, B) > 15$ And $\text{Abs}(R - G) > 15$

函数里判断阈值条件即可：

```

1. for r in range(rows):
2.     for c in range(cols):
3.         # get pixel value
4.         B = img5.item(r, c, 0)
5.         G = img5.item(r, c, 1)
6.         R = img5.item(r, c, 2)
7.         skin = 0
8.         if (abs(R - G) > 15) and (R > G) and (R > B):
9.             if (R > 95) and (G > 40) and (B > 20) and (max(R, G, B) - m
in(R, G, B) > 15):
10.                 skin = 1
11.                 elif (R > 220) and (G > 210) and (B > 170):
12.                     skin = 1
13.
14.         if 0 == skin:
15.             imgSkin.itemset((r, c, 0), 0)
16.             imgSkin.itemset((r, c, 1), 0)
17.             imgSkin.itemset((r, c, 2), 0)

```

4. 融合

根据调查,选择基于小波变换的图像融合方法,因为图像的小波分解具有方向性和塔状结构,在融合处理时,根据需要针对不同频率分量、不同方向、不同分解层或针对同一分解层的不同频率分量采用不同的融合规则进行融合处理,这样能充分利用图像的互补和冗余信息来达到良好的融合效果。

基于以上原理,对引导滤波处理后的图像和原图像进行融合,希望既能保留原图细节,又能起到较好的磨皮效果。

以下是**算法思路**:

- 1) 对原始图像进行预处理和图像配准;
- 2) 对处理过的图像分别进行小波分解,得到低频和高频分量;
- 3) 对低频和高频分量采用不同的融合规则进行融合;
- 4) 进行小波逆变换;
- 5) 得到融合图像。

定义方差函数:

```

1. def fangcha(img):
2.     row=img.shape[0]
3.     col=img.shape[1]
4.     varImg=np.zeros([row,col])

```

```

5.     for i in range(row):#求取方差范围
6.         for j in range(col):
7.             if i-5>0:
8.                 up=i-5
9.             else:
10.                up=0
11.            if i+5<row:
12.                down=i+5
13.            else:
14.                down=row
15.            if j-5>0:
16.                left=j-5
17.            else:
18.                left=0
19.            if j+5<col:
20.                right=j+5
21.            else:
22.                right=col
23.            window=img[up:down,left:right]
24.            mean,var=cv.meanStdDev(window)#调用OpenCV 函数求取均值和方差
25.            varImg[i,j]=var
26.    return varImg

```

定义求图像权值函数：

```

1. def qiuquan(img1,img2):
2.     row=img1.shape[0]
3.     col=img1.shape[1]
4.     array1=fangcha(img1)#调用求方差函数
5.     array2=fangcha(img2)
6.     for i in range(row):#求权
7.         for j in range(col):
8.             weight1=array1[i,j]/(array1[i,j]+array2[i,j])
9.             weight2=array2[i,j]/(array1[i,j]+array2[i,j])
10.            array1[i,j]=weight1
11.            array2[i,j]=weight2
12.    return array1,array2

```

定义融合函数，对两幅图像融合：

```

1. def ronghe(img1,img2):
2.     cc = img1.copy()
3.     b,g,r=cv.split(img1)#分通道处理
4.     b1,g1,r1=cv.split(img2)

```

```

5.     weight1,weight2=qiuquan(b,b1)#调用求权重函数
6.     weight11,weight22=qiuquan(g,g1)
7.     weight111,weight222=qiuquan(r,r1)
8.     new_img=img1*1
9.     row=new_img.shape[0]
10.    col=new_img.shape[1]
11.    b2,g2,r2 = cv.split(cc)
12.    for i in range(row):#图像融合
13.        for j in range(col):
14.            b2[i,j]=(weight1[i,j]*b[i,j]+weight2[i,j]*b1[i,j]).astype(int)
15.            g2[i,j]=(weight11[i,j]*g[i,j]+weight22[i,j]*g1[i,j]).astype(int)
16.            r2[i,j]=(weight111[i,j]*r[i,j]+weight222[i,j]*r1[i,j]).astype(int)
17.    new_img=cv.merge([b2,g2,r2])#通道合并
18.    return new_img

```

5. 锐化

使用 Gabor 滤波来提取图像特征，锐化图像。

原因在于 Gabor 小波与人类视觉系统中简单细胞的视觉刺激响应非常相似。它在提取目标的局部空间和频率域信息方面具有良好的特性。Gabor 是一个用于边缘提取的线性滤波器，其频率和方向表达与人类视觉系统类似，能够提供良好的方向选择和尺度选择特性，而且对于光照变化不敏感，因此十分适合纹理分析。在人脸识别等领域有着很广泛的应用。

定义一个 Gabor 滤波器：Gabor_filter，对每个像素坐标进行处理，设置核函数：

```

1. def Gabor_filter(K_size=111, Sigma=10, Gamma=1.2, Lambda=10, Psi=0, angle=0):
2.     # get half size
3.     d = K_size // 2
4.     # prepare kernel
5.     gabor = np.zeros((K_size, K_size), dtype=np.float32)
6.     # each value
7.     for y in range(K_size):
8.         for x in range(K_size):
9.             # distance from center
10.            px = x - d
11.            py = y - d
12.            # degree -> radian
13.            theta = angle / 180. * np.pi
14.            # get kernel x

```

```

15.         _x = np.cos(theta) * px + np.sin(theta) * py
16.         # get kernel y
17.         _y = -np.sin(theta) * px + np.cos(theta) * py
18.         # fill kernel
19.         gabor[y, x] = np.exp(-
            (_x**2 + Gamma**2 * _y**2) / (2 * Sigma**2)) * np.cos(2*np.pi*_x/Lambda + P
            si)
20.         # kernel normalization
21.         gabor /= np.sum(np.abs(gabor))
22.         return gabor

```

定义 Gabor_filtering, 使 Gabor 滤波器作用于图像上, 对(x,y)变换

```

1. def Gabor_filtering(gray, K_size=111, Sigma=10, Gamma=1.2, Lambda=10, Psi=0
    , angle=0):
2.     # get shape
3.     H, W = gray.shape
4.     # padding
5.     gray = np.pad(gray, (K_size//2, K_size//2), 'edge')
6.     # prepare out image
7.     out = np.zeros((H, W), dtype=np.float32)
8.     # get gabor filter
9.     gabor = Gabor_filter(K_size=K_size, Sigma=Sigma, Gamma=Gamma, Lambda=La
        mbda, Psi=0, angle=angle)
10.    # filtering
11.    for y in range(H):
12.        for x in range(W):
13.            out[y, x] = np.sum(gray[y : y + K_size, x : x + K_size] * gabor
                )
14.    out = np.clip(out, 0, 255)
15.    out = out.astype(np.uint8)
16.    return out

```

使用 6 个不同角度的 Gabor 滤波器对图像进行特征提取, 保留边缘

```

1. def Gabor_process(img):
2.     # get shape
3.     H, W = img.shape
4.     As = [0,30,60,90,120,150]
5.     # prepare pyplot
6.     plt.subplots_adjust(left=0, right=1, top=1, bottom=0, hspace=0, wspace=
        0.2)
7.     out = np.zeros([H, W], dtype=np.float32)
8.     # each angle

```

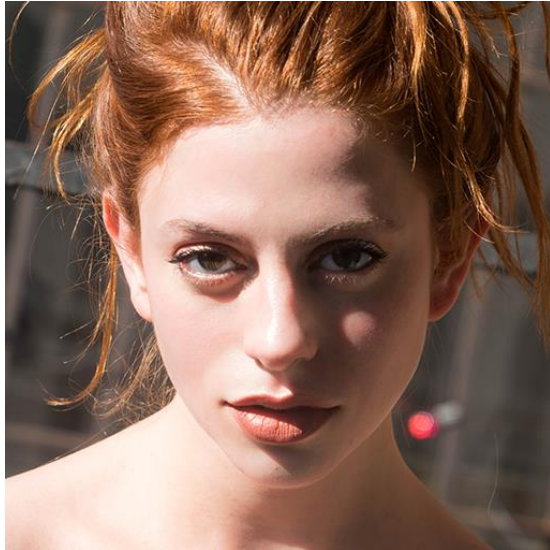
```

9.     for i, A in enumerate(As):
10.         # gabor filtering
11.         _out = Gabor_filtering(img, K_size=11, Sigma=1.5, Gamma=1.2, Lambda
            =3, angle=A)
12.         # add gabor filtered image
13.         out += _out
14.         # scale normalization
15.         out = out / out.max() * 255
16.         out = out.astype(np.uint8)
17.     return out

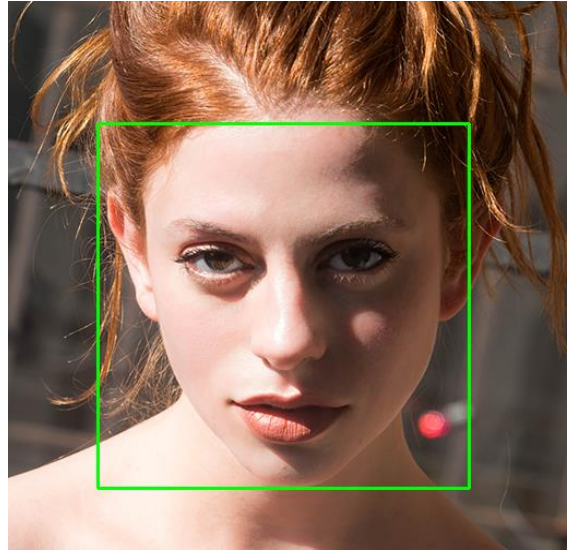
```

（二）实验结果

1. 人脸检测



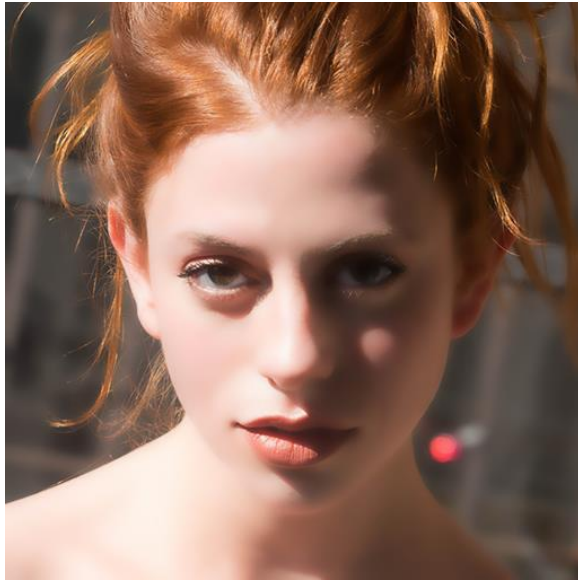
原图



检测图

如上图，绿色框内即为检测的人脸

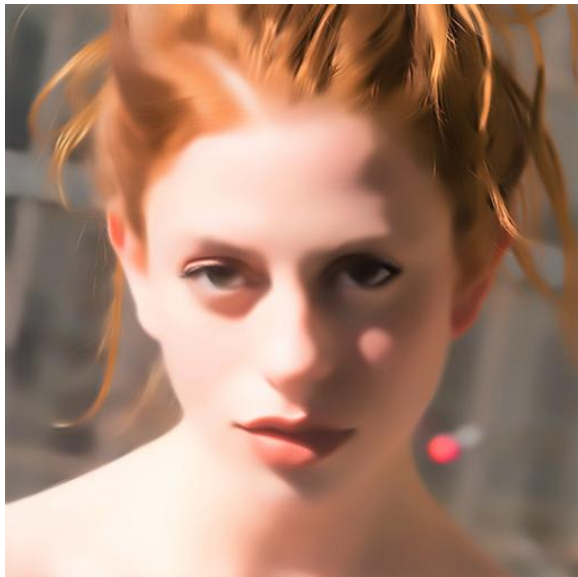
2. 滤波器



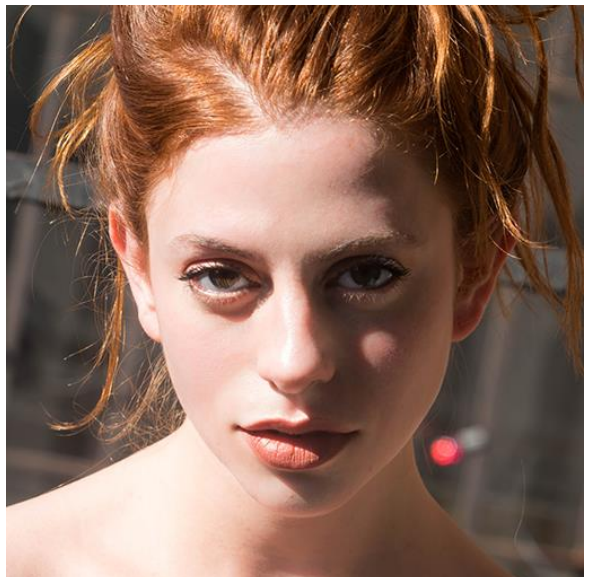
引导滤波图



cv 传统 NLM 滤波（不加噪声）



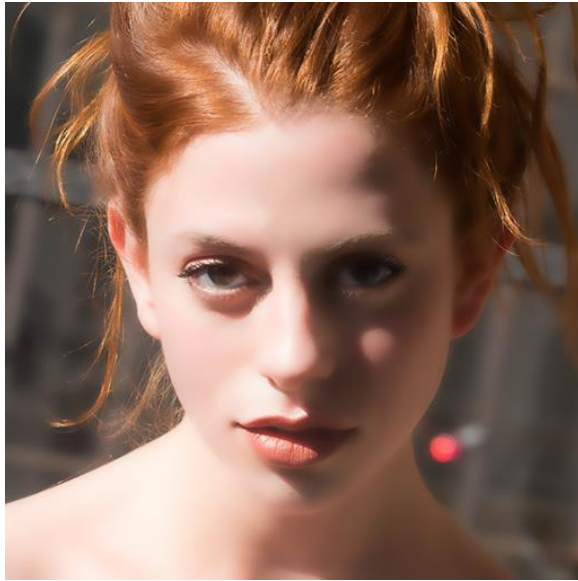
自定义 NLM 滤波（不加噪声）



原图

所以选择引导滤波效果较好，原细节保留更丰富

3. 皮肤检测（选择引导滤波）



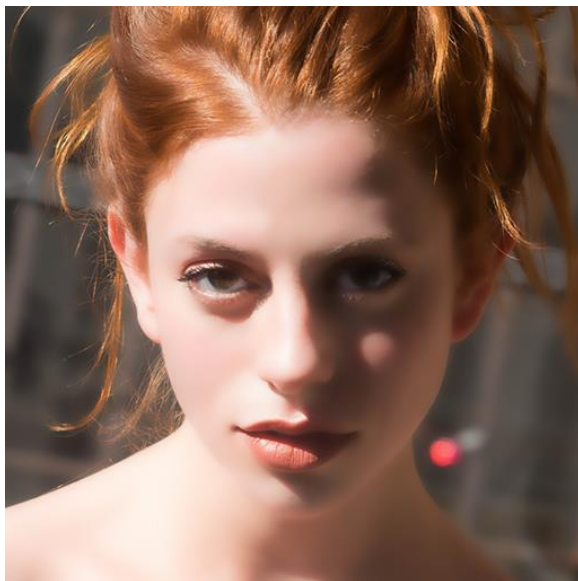
引导滤波图



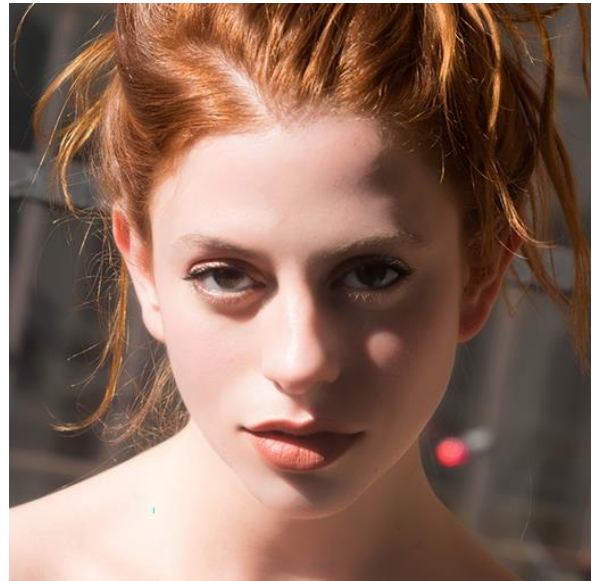
皮肤检测

因为主角脸部存在阴影，所以函数可能将阴影区域给去除了

4. 融合（选择引导滤波）



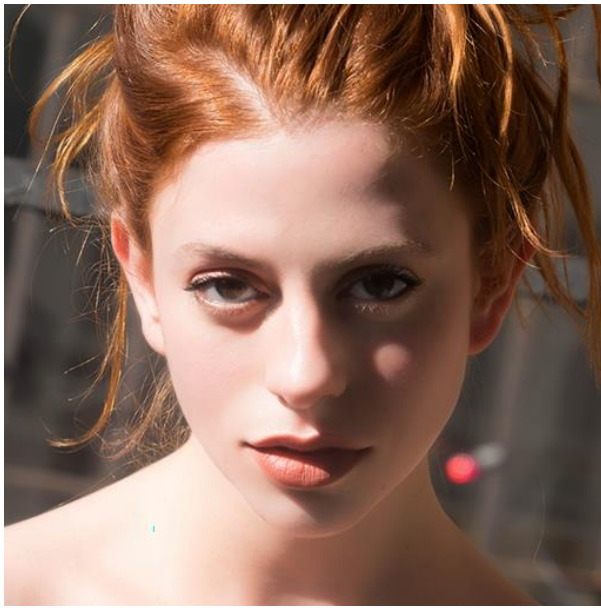
引导滤波图



引导滤波-融合图

滤波图与原图按权重进行小波融合后，既实现了磨皮，又保留了原图的细节

5. 锐化（选择引导滤波）



引导滤波-融合图



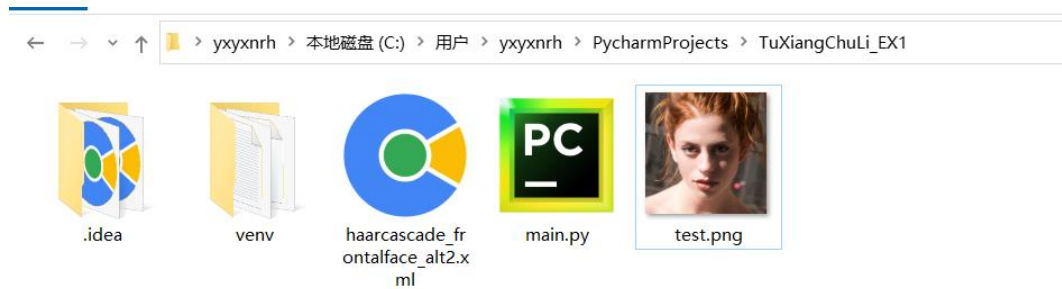
引导滤波-融合-锐化图

锐化后更加突出了五官，边缘细节更丰富了

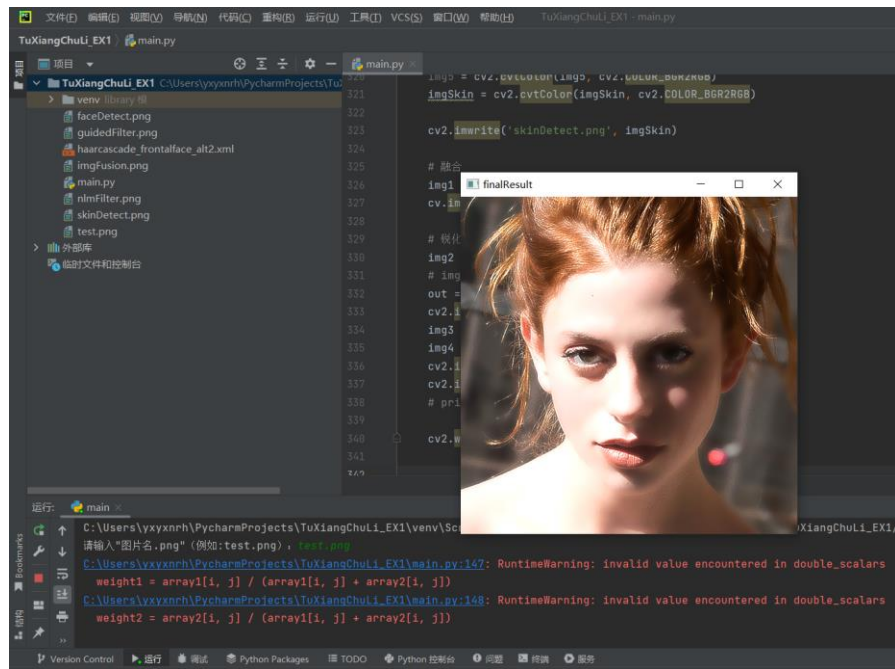
6. 综合测试

对 30 多张照片进行总体测试：

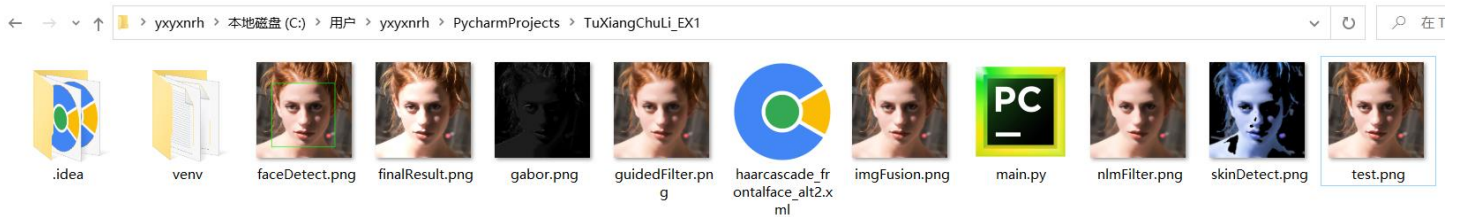
运行 `main.py`，`test.png` 在同一目录下，`.xml` 为人脸检测文件，输入 `test.png`



回车后，一会跳出弹窗 `finalResult`，美颜结果：



目录中多出以下.png 图像文件:



faceDetect.png: 人脸检测结果

guidedFilter.png: 引导滤波结果

nlmFilter.png: nlm 滤波结果

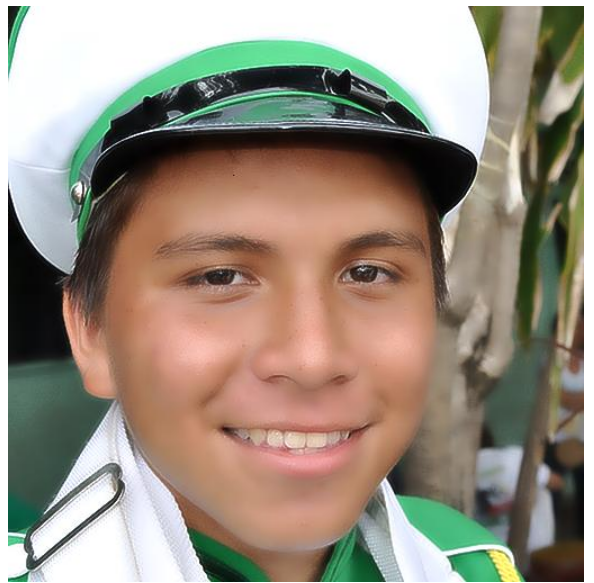
skinDetect.png: 皮肤检测结果

imgFusion.png: 图像融合结果

gabor.png: gabor 滤波边缘提取 (锐化)

finalResult.png: 最终美颜结果

以下是部分原图与美颜图的对比，左为原图，右为美颜图



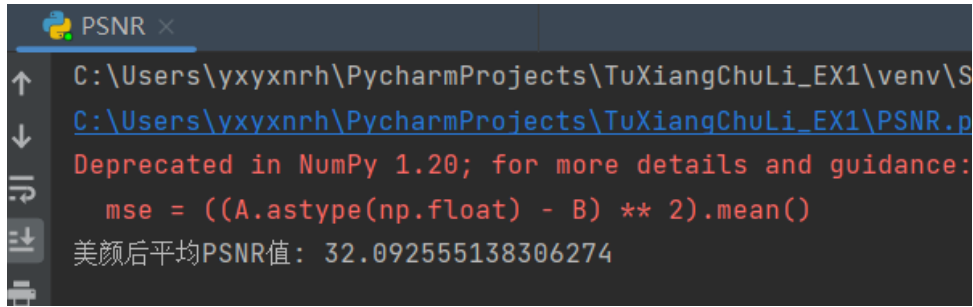
7. 美颜标准评价

使用图像质量评估指标 PSNR 原理对 30 多份图像进行评分，编写 PSNR 函数计算每张图片 PSNR 值，并批量计算所有处理后的美颜图的 PSNR 平均值

```
1. def psnr(A, B):
2.     val = 255
3.     mse = ((A.astype(np.float) - B) ** 2).mean()
4.     return 10 * np.log10((val * val) / mse)

1. def main():
2.     WSI_MASK_PATH1 = 'E:/test/A/'
3.     WSI_MASK_PATH2 = 'E:/test/B/'
4.     path_real = glob(os.path.join(WSI_MASK_PATH1, '*.jpg'))
5.     path_fake = glob(os.path.join(WSI_MASK_PATH2, '*.jpg'))
6.     list_psnr = []
7.     list_ssim = []
8.     list_mse = []
9.
10.    for i in range(len(path_real)):
11.        t1 = read_img(path_real[i])
12.        t2 = read_img(path_fake[i])
13.        result1 = np.zeros(t1.shape, dtype=np.float32)
14.        result2 = np.zeros(t2.shape, dtype=np.float32)
15.        cv2.normalize(t1, result1, alpha=0, beta=1, norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F)
16.        cv2.normalize(t2, result2, alpha=0, beta=1, norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F)
17.        mse_num = mse(result1, result2)
18.        psnr_num = psnr(result1, result2)
19.        ssim_num = ssim(result1, result2)
20.        list_psnr.append(psnr_num)
21.        list_ssim.append(ssim_num)
22.        list_mse.append(mse_num)
23.
24.    #输出平均指标:
25.    print("美颜后平均 PSNR 值:", np.mean(list_psnr)) # , list_ssim)
```

输出结果，30 多张美颜图像的 PSNR 平均值为 32:

A screenshot of a terminal window with a dark background. The title bar shows a Python icon and the text 'PSNR x'. The terminal displays the file path 'C:\Users\yxyxnrh\PycharmProjects\TuXiangChuLi_EX1\venv\Scripts\PSNR.py' in blue. Below it, a red deprecation warning from NumPy 1.20 is shown: 'Deprecated in NumPy 1.20; for more details and guidance:'. The next line shows a code snippet 'mse = ((A.astype(np.float) - B) ** 2).mean()' in red. The final line shows the result '美颜后平均PSNR值: 32.092555138306274' in white.

```
PSNR x
C:\Users\yxyxnrh\PycharmProjects\TuXiangChuLi_EX1\venv\Scripts\PSNR.py
Deprecated in NumPy 1.20; for more details and guidance:
mse = ((A.astype(np.float) - B) ** 2).mean()
美颜后平均PSNR值: 32.092555138306274
```

PSNR 高于 40dB 说明图像质量极好（即非常接近原始图像），在 30—40 dB 通常表示图像质量是好的（即失真可以察觉但可以接受），因此此美颜图片符合 PSNR 评价标准。

（三）结论分析

1. 出现的问题与解决方法。

1) 问题:

在自己编写的 NLM 算法中，给图像添加高斯噪声时，`image` 读取出来的图像，类型是 `uint8` 的（也就是取值范围是 0-255），而计算的随机噪声值，可能是或负或正的小数，直接叠加噪声的图像 `img_with_noise`，其实是有可能是负值，有可能超过了 255 的，而且这个 `np.ndarray` 类型是普通的 `float64` 了，这个时候用 `imshow` 函数来显示，会出现全白全黑的图像 bug。

解决方案:

解决方法是在显示前将图像做归一化 `normalization` 转成 0-255 区间的 `uint` 类型。

2) 问题:

本来图像融合这步是想使用 Brovey 图像融合方法的，通过对多光谱数据进行归一化后，与高分辨率原图进行乘积来增加美颜相片信息。但是此方法需要提前安装 python 的 `gdal` 库，`gdal` 库不是纯净的 python 库，无法像 `pip install flask` 这样安装库文件，需要本地安装 `.whl` 文件，一直没安装好。

解决方案：最后还是使用 `opencv` 的小波融合

2. 尚存在的问题。

1) 人脸检测函数不适合处理检测那些复杂场景中的人脸图片, 精确度不够, 主要是 `opencv` 自带的函数功能还是过于简单了, 分类器文件无法适应所有情况, 还是应该基于深度学习的网络, 这样精度会提高。

2) 程序运行时间略长, 可能需要优化一些。