第一题

一、算法描述

1. 由于要实现动画效果，故生成足够多的图片序列，然后按时间顺序将其做成动画。
2. 提取两张图片的红色通道作为灰度值，并得到灰度图像。
3. 对于每一帧图片，给定 lena 图片在诺贝尔图片上的半径大小，其中半径大小与经过时间 t 为正比。因为要覆盖第一张图片，故最大半径为对角线长度的一半。
4. 针对每帧图像逐像素进行扫描，由于图像被存成二维矩阵，计算当前像素点的圆心距，如果大于 lena 图片设定的半径，则像素值为诺贝尔图片相应位置的值，否则为 lena 图片相应位置的值。

二、程序实现

1. 采用 python3，及 numpy，opencv，scipy 等库实现。
2. scipy.misc 库中的 imread 可以将图片读取成 numpy 数组，imsave 可以将 numpy 数组存成图片。
3. openv 库的 VideoWriter 用于写视频。

```
# read img from the disk
nobel_img = imread('诺贝尔.jpg')
lena_img = imread('lena.jpg')
print(nobel_img.dtype,nobel_img.shape)
print(lena_img.dtype,nobel_img.shape)

# convert to red channel
nobel_img = nobel_img[:,:,0]
lena_img = lena_img[:,:,0]
```

```
for i in range(1,100):
    r_t = r_max*i/99
    for x in range(0,row_size):
        for y in range(0,col_size):
            r = sqrt((x-row_size/2)**2+(y-col_size/2)**2)
            if r< r_t:
                img[x,y] = lena_img[x,y]
            else:
                img[x,y] = nobel_img[x,y]
    images.append(img)
    # Write the tinted image back to disk
    imsave('Assets/img'+str(i)+'.jpg', img)
```

```
# 设定视频每帧的图片格式                                    G: str
fourcc = cv2.VideoWriter_fourcc('M', 'J', 'P', 'G')
size = (row_size,col_size)
# 设定视频每秒帧数和每帧像素大小
vw = cv2.VideoWriter('file.avi', fourcc=fourcc, fps=20, frameSize=size)

# 设定所有帧序列
for i in range(0,100):
    f_read = cv2.imread('Assets/img'+str(i)+'.jpg')
    vw.write(f_read)
vw.release()
```

三、实现效果

具体见 pro1 文件夹

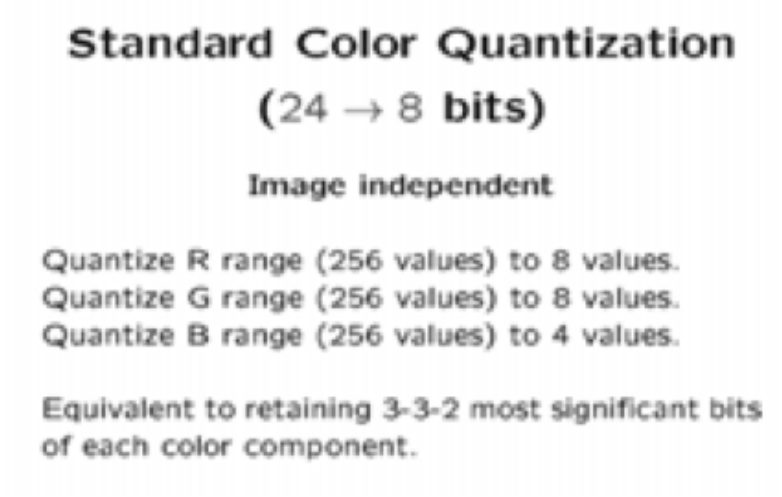
1. 生成的 file.avi 为最终动画
2. pro1.gif 为转换效果动图

第二题

一、算法描述

1. 8 位彩色图片使用颜色查找表存储 256 种颜色信息，需要将 24 位真彩色图划分为 256 个颜色空间然后将将每个区域的平均值作为颜色查找表中的一项，而每个像素点存储的是表的索引。这种减色算法适用于压缩一些不注重细节的图片，如许多 gif 格式中的图片，其中常见的划分方法有几种，分别是流行色算法(popularity algorithm)，中位切分算法(median-cut algorithm)，八叉树颜色量化算法和 KMeans 聚类。

2. 最直接的方法就是将 256*256*256 的颜色空间等分成 256 份，将每个区域中心的颜色作为查找表的表项，然而比起蓝色，人们对红色和绿色更加敏感，所以还可以使用如下方法将每个通道 0-255 的值进行不同映射。



3. 上面这种简单的方案会出现边缘效果，因为 RGB 的轻微变化就会导致移动到新的表项，而中值分割算法即有比较高的识别能力，又具有比较小的运算代价，故采用其生成颜色查找表。根据维基百科的定义，以下是算法伪代码：

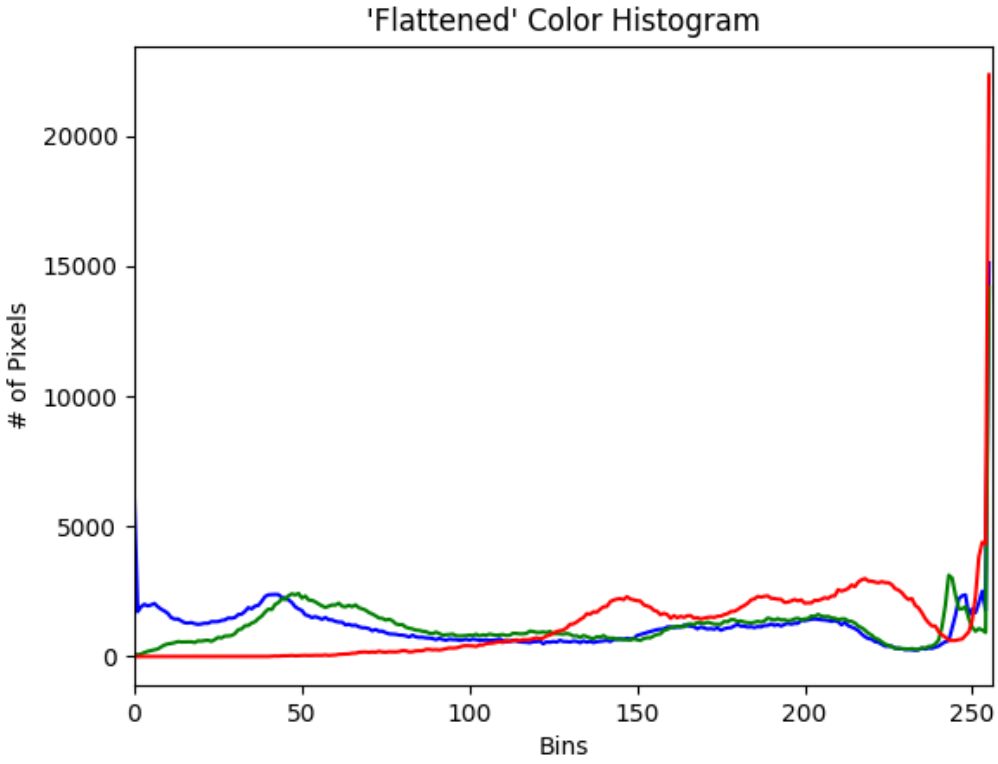
1. 将图片内的所有像素加入到同一个区域
2. 对于所有的区域做以下的事：
    1. 计算此区域内所有像素的 RGB 三元素最大值与最小值的差。
    2. 选出相差最大的那个颜色（R 或 G 或 B）
    3. 根据那个颜色去排序此区域内所有像素
    4. 分割前一半与后一半的像素到二个不同的区域（这里就是“中位切割”名字的由来）
3. 重复第二步直到你有 256 个区域
4. 将每个区域内的像素平均起来，于是你就得到了 256 色

The median cut algorithm

```
Color_quantization(Image, n){

        For each pixel in Image with color C, map C in RGB space;

        B = {RGB space};
        While (n-- > 0) {
                L = Heaviest (B);
                Split L into L1 and L2;
                Remove L from B, and add L1 and L2 instead;
                }

        For all boxes in B do
                assign a representative (color centroid);

        For each pixel in Image do
                map to one of the representatives;
}
```

二、程序实现

首先用 opencv 的 calcHist 在一张图中画出 rgb 的直方图。



建立 ColorSpace 数据结构，每个颜色空间存储 RGB 三元组，其长宽高分别为该空间中所有像素点 RGB 的极差，用 size 成员函数返回，average 成员函数返回每个空间所有像素 RGB 的平均值。

```python
# 每个颜色通道的极差
@property
def size(self):
    return self.rmax - self.rmin, self.gmax - self.gmin, self.bmax - self.bmin

# 每个cube的RGB平均值
@property
def average(self):
    # 所有像素点的数量
    length = len(self.colors)
    # 提取每个通道的值
    r = [c[0] for c in self.colors]
    g = [c[1] for c in self.colors]
    b = [c[2] for c in self.colors]
    # 取每个通道的平均值
    r = sum(r) / length
    g = sum(g) / length
    b = sum(b) / length
    # print(r,g,b)
    return r, g, b
```

```
# 按中值一分为二，其中axis代表要排序的颜色，0为r，1为g，2为b
def split(self, axis):
    self.color_size()
    self._colors = sorted(self.colors, key=itemgetter(axis))

    # Find median
    med_idx = round(len(self.colors) / 2)

    # Create splits
    return (
        ColorSpace(*self.colors[:med_idx]),
        ColorSpace(*self.colors[med_idx:]
    ))
```

根据之前的算法，每次保存一个全局最大极差和局部最大极差，在每次循环时，首先在每个颜色空间保存局部最大极差，然后不断更新全局最大极差。然后根据全局最大极差的那一维排序 rgb 三元组，不断切分初始颜色空间，而另一种方式是书上所说，每次按照 RGB 的顺序排序，然后指数分裂，分别最后相当于 9 层二叉树，每层按照 RGB 的顺序排序而不是最大极差。

```
def median_cut(image, num_colors):
    colors = Colors(image)
    print(colors[:10])
    # 建立RGB立方体，每个像素点的RGB值为一个三维坐标
    cubes = [ColorSpace(*colors)]
    count = 1
    while len(cubes) < num_colors:
        # 所有子颜色空间最大的边长
        global_max_size = 0

        for index, cube in enumerate(cubes):
            # 每个通道的极差
            size = cube.size
            # 此颜色空间RGB的最大极差
            max_size = max(size)

            if max_size > global_max_size:
                global_max_size = max_size
                max_cube = index
```

```
        # 第1，4，7层按照R划分
        if (len(cubes)>=1 and len(cubes)<=2) or
          (len(cubes)>=8 and len(cubes)<=16) or
          (len(cubes)>=64 and len(cubes)<=128):
            max_color = 0
        # 第2，5，8层按照G划分
        elif (len(cubes)>=2 and len(cubes)<=4) or
           (len(cubes)>=16 and len(cubes)<=32) or
           (len(cubes)>=128 and len(cubes)<=256):
            max_color = 1
        # 第3，6层按照B划分
        else:
            max_color = 2
        split_cube = cubes[max_cube]
        cube_low, cube_high = split_cube.split(max_color)
        cubes = cubes[:max_cube] + [cube_low, cube_high] + cubes[max_cube + 1:]

    return [c.average for c in cubes]
```

最后得到 256 色颜色查找表 palette，与原图片每个像素点算欧式距离，以最小的距离的 24 位颜色作为分配。

```
count = 0
for x in range(height):
    for y in range(width):
        distances = []
        for c in palette:
            distances.append(np.sqrt(np.sum((apple_im[x,y]-c)**2)))
        dest1[x, y] = palette[np.argmin(distances)]
        count += 1
        if(count%1000==0):
            print(count)
```

三、实现效果

由于 pillow 库中有接口 quantize 接口，将生成的图片进行比较

Image.quantize(*colors=256, method=None, kmeans=0, palette=None*) [source]

Convert the image to 'P' mode with the specified number of colors.

Parameters:
- **colors** – The desired number of colors, <= 256
- **method** – 0 = median cut 1 = maximum coverage 2 = fast octree 3 = libimagequant
- **kmeans** – Integer
- **palette** – Quantize to the palette of given `PIL.Image.Image`.

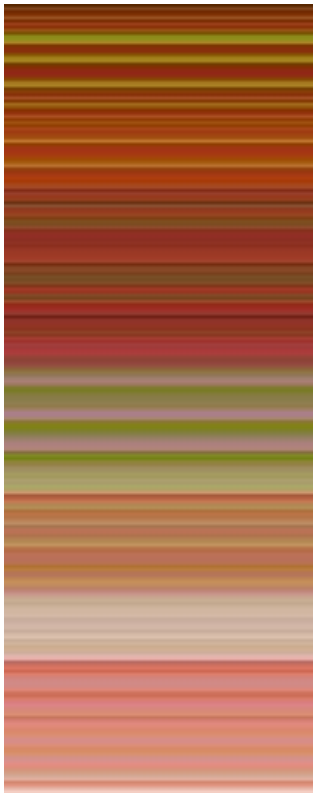
Returns: A new image

原图

按 RGB 顺序指数分裂的 8 位减色图

按所有区域中的最大 RGB 极差分裂的 8 位减色图

使用 Quantize 接口生成的图



按 RGB 顺序指数分裂 256 色调色板

按最大 RGB 极差分裂 256 色调色板