

哈爾濱工業大學

实验报告

题目	图片基本处理
专业	计算机科学与技术
学号	1160300329
班级	1603107
学生	黄海
指导教师	郑铁然
实验地点	G709
实验日期	12.10

计算机科学与技术学院

一、图片信息的获取

1.1 图片像素数据的获取

对于不同格式的图片 我统一交给了opencv来进行基本的打开和像素数据的获取

通过opencv打开后 我们能够通过对对象获取到图片的基本信息 例如图片的长和高 还有像素数据的深度 并且可以简单的获取到图片的像素数据

```
# 亮度调整
def brightness(self, filepath, change):
    # try:
        # 图片的打开
        img = cv.imread(filepath)
        if img == None:
            print("IOError")
            exit(0)
        rows, cols = img.shape[:2]
        ...
        # 像素调整
        for i in range(rows):
            for j in range(cols):
                for k in range(3):
                    img[i, j][k] += change
        ...
    # except IOError as e:
    #     print("ERROR: " + str(e))
```

1.2 数据结构的变换

通过opencv打开的图片 需要对**数据结构**格外注意

这里的数据结构是标准的类二维数组表示 在调用像素点的数据时需要通过类似于`data[i, j]`的方式来获取 并且拿到的数据为标准的list 内部存有RGB数据 在调用时需要不断的观察数据结构的变化

这里需要注意以下几个问题

- **二维数组的调用**

在调用时除了通过`i, j`循环外 使用`[:, :, :]`的结构来获得特定窗口的数据 在进行滤波时效率较高 但是可读性较低

- **图片异常处理**

在判断打开时不能通过`try IOError`来获得异常 必须通过判`None` 在官方的文档中有以

下的注意事项：

WARNING:

Even if the image path is wrong, it won't throw any error, bu

- **维度变化**

数据结构的维度变化频繁 需要使用`reshape()`来进行调整

二、基本的处理

2.1 亮度 对比度

对于图片的亮度和对比度 我们由以下公式给出

$$G(i, j) = \alpha F(i, j) + \beta$$

其中 G 为结果 F 为原数据 α 、 β 分别对应该对比度和亮度

所以我们在进行调整时 直接对像素点的值进行调整 同时注意超过上下限的处理

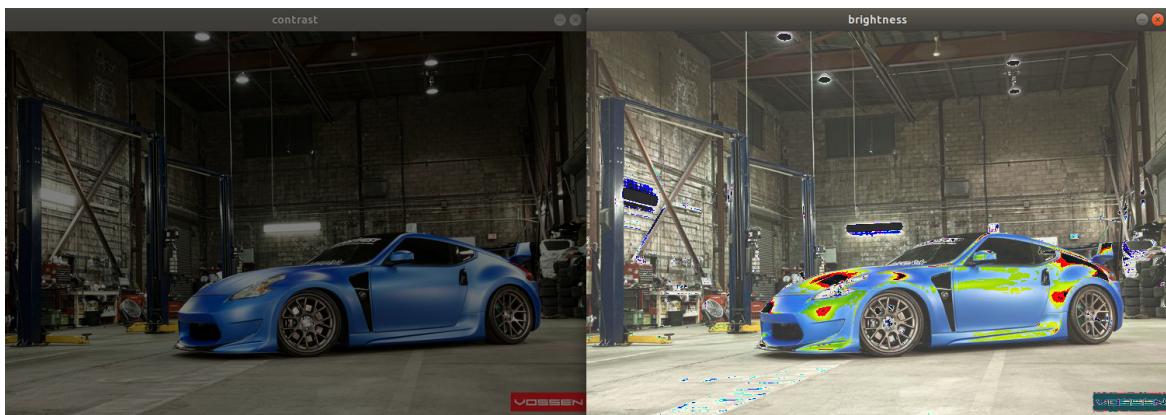
```
def brightness(self, filepath, change):
    try:
        img = cv.imread(filepath)
        rows, cols = img.shape[:2]
        if change >= 255 or change < -255:
            print("illegal!")
            exit(0)
        for i in range(rows):
            for j in range(cols):
                for k in range(3):
                    img[i, j][k] += change
                    if img[i, j][k] > 255:
                        img[i, j][k] = 255
                    elif img[i, j][k] < 0:
                        img[i, j][k] = 0
        cv.imshow("brightness", img)
        cv.waitKey(0)
    except Exception as e:
        print("ERROR: " + str(e))

def contrast(self, filepath, change):
    try:
        img = cv.imread(filepath)
        rows, cols = img.shape[:2]
```

```

if change < 0:
    print("illegal!")
    exit(0)
for i in range(rows):
    for j in range(cols):
        for k in range(3):
            img[i, j][k] = int(change * img[i, j][k])
            if img[i, j][k] > 255:
                img[i, j][k] = 255
            elif img[i, j][k] < 0:
                img[i, j][k] = 0
cv.imshow("contrast", img)
cv.waitKey(0)
except Exception as e:
    print("ERROR: " + str(e))

```



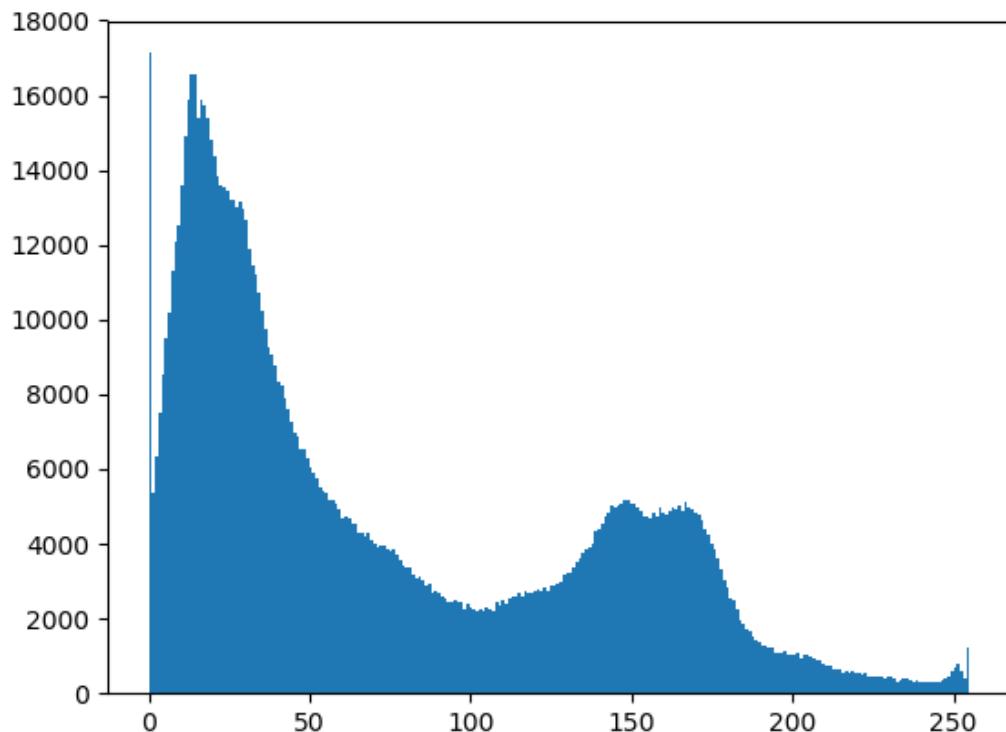
2.2 直方图

对于图片 取直方图并没有什么难点 主要是要将图片数据更改为一维数组 然后操作

```

def histogram(self, filepath):
    try:
        img = cv.imread(filepath)
        data = np.asarray(img)
        data = data.flatten()
        plt.hist(data, bins=256)
        plt.show()
    except Exception as e:
        print("ERROR: " + str(e))

```



2.3 中值滤波

中值滤波采取的是在一个窗口中 将核心像素的数据更改为窗口的中值 这里我采用的是3*3窗口
然后就是遍历像素 获取结果

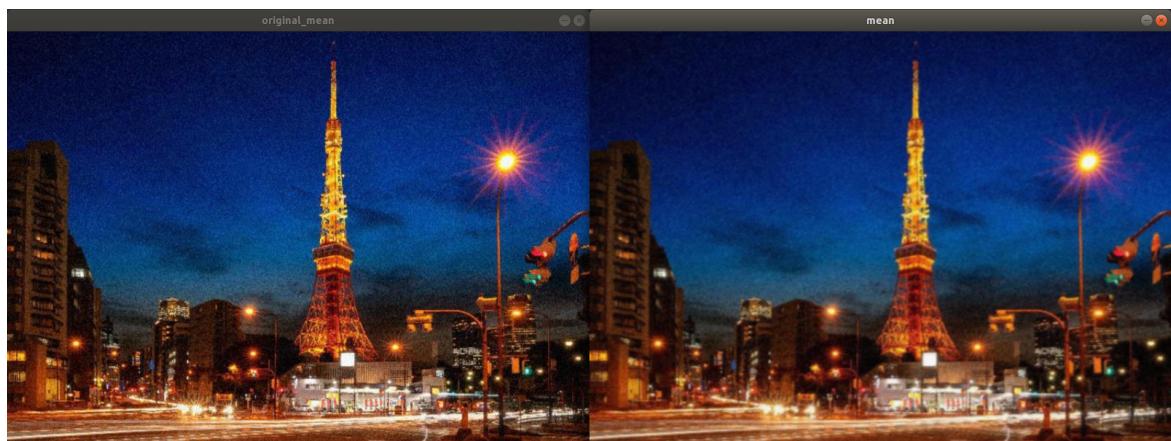
```
def median(self, filepath):
    try:
        img = cv.imread(filepath)
        cv.imshow("original_median", img)
        rows, cols = img.shape[:2]
        d = np.asarray(img)
        data = []
        for i in range(1, rows - 1):
            for j in range(1, cols - 1):
                data = d[i - 1:i + 2:1, j - 1:j + 2:1].reshape(9, 3)
                img[i, j] = np.median(data, axis=0)
        cv.imshow("median", img)
        cv.waitKey(0)
    except Exception as e:
        print("ERROR: " + str(e))
```



2.4 均值滤波

均值滤波采取的是在一个窗口中 将核心像素的数据更改为窗口的均值 这里我采用的是3*3窗口
然后就是遍历像素 获取结果 与中值差别不大

```
def mean(self, filepath):
    try:
        img = cv.imread(filepath)
        cv.imshow("original_mean", img)
        rows, cols = img.shape[:2]
        d = np.asarray(img)
        for i in range(1, rows - 1):
            for j in range(1, cols - 1):
                data = d[i - 1:i + 2:1, j - 1:j + 2:1].reshape(9, 3)
                img[i, j] = np.mean(data, axis=0)
        cv.imshow("mean", img)
        cv.waitKey(0)
    except Exception as e:
        print("ERROR: " + str(e))
```



2.5 边缘检测

reborts算子

reborts算子采用了四像素的窗口来进行梯度计算

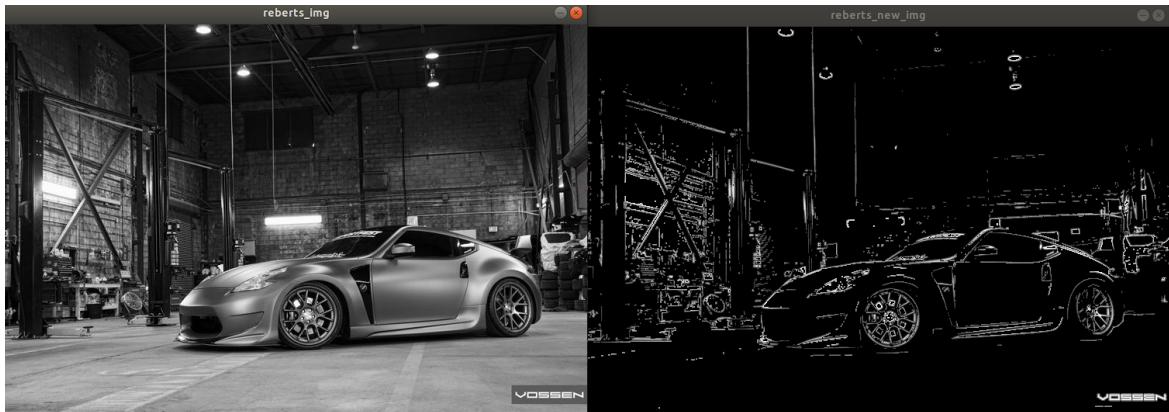
z_1	z_2	z_3
z_4	z_5	z_6
z_7	z_8	z_9

我们定义两个方向的梯度

$$\begin{aligned}g_x &= z_9 - z_5 \\g_y &= z_8 - z_6 \\M &= |g_x| + |g_y|\end{aligned}$$

M就是我们判断的依据 通过梯度的大小来判断边缘

```
def edge_reborts(self, filepath):
    try:
        img = cv.imread(filepath, 0)
        rows, cols = img.shape[:2]
        new_img = np.zeros((rows, cols), np.uint8)
        d = np.asarray(img)
        for i in range(rows - 1):
            for j in range(cols - 1):
                data = d[i:i + 2:1, j:j + 2:1]
                g_x = abs(int(data[1, 1]) - int(data[0, 0]))
                g_y = abs(int(data[1, 0]) - int(data[0, 1]))
                M = g_x + g_y
                if M >= 100:
                    new_img[i:i + 2:1, j:j + 2:1] = d[i:i + 2:1, j:j + 2:1]
        cv.imshow("reborts_img", img)
        cv.imshow("reborts_new_img", new_img)
        cv.waitKey(0)
    except Exception as e:
        print("ERROR: " + str(e))
```

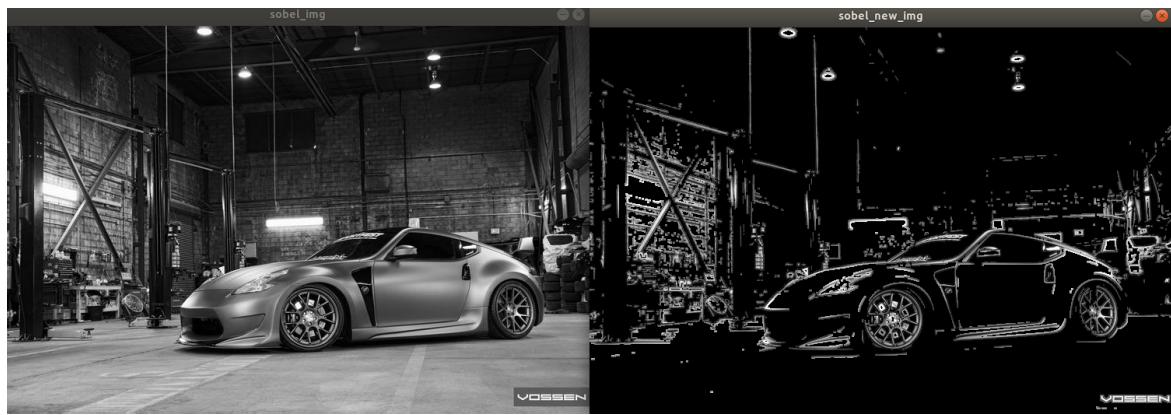


sobel算子

sobel算子取用了九像素窗口 同样的是上边的示意图 只不过梯度发生了变化

$$\begin{aligned} g_x &= (z_9 + 2z_8 + z_7) - (z_3 + 2z_2 + z_1) \\ g_y &= (z_3 + 2z_6 + z_9) - (z_1 + 2z_4 + z_7) \\ M &= |g_x| + |g_y| \end{aligned}$$

```
def edge_sobel(self, filepath):
    try:
        img = cv.imread(filepath, 0)
        rows, cols = img.shape[:2]
        new_img = np.zeros((rows, cols), np.uint8)
        d = np.asarray(img)
        for i in range(rows - 2):
            for j in range(cols - 2):
                data = d[i:i + 3:1, j:j + 3:1]
                g_x = abs((int(data[2, 0]) + 2 * int(data[2, 1]) + int(data[2, 2])) - (int(data[0, 0]) + 2 * int(data[0, 1]) + int(data[0, 2])))
                g_y = abs((int(data[0, 2]) + 2 * int(data[1, 2]) + int(data[2, 2])) - (int(data[0, 0]) + 2 * int(data[1, 0]) + int(data[2, 0])))
                M = g_x + g_y
                if M >= 300:
                    new_img[i:i + 3:1, j:j + 3:1] = d[i:i + 3:1, j:j + 3:1]
    cv.imshow("reborts_img", img)
    cv.imshow("reborts_new_img", new_img)
    cv.waitKey(0)
except Exception as e:
    print("ERROR: " + str(e))
```



2.6 快速中值

中值滤波的问题主要是性能 由于每个像素基本都需要遍历 造成了时间上的浪费 而快速中值的表现更加稳定

显然最初的算法的时间复杂度为 $O(n^2)$ 下边给出的算法 时间复杂度提高到 $O(1)$

提高效率 集中在如何更加高效率的使用加载的数据 首先是初等的优化

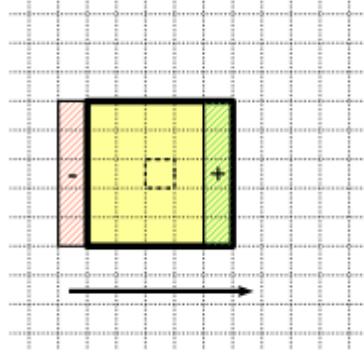


Fig. 1: In Huang's $O(n)$ algorithm, $2r + 1$ pixels must be added to and subtracted from the kernel's histogram when moving from one pixel to the next. In this figure, $r = 2$.

Algorithm 1 Huang's $O(n)$ median filtering algorithm.

Input: Image X of size $m \times n$, kernel radius r

Output: Image Y of the same size as X

```

Initialize kernel histogram  $H$ 
for  $i = 1$  to  $m$  do
    for  $j = 1$  to  $n$  do
        for  $k = -r$  to  $r$  do
            Remove  $X_{i+k,j-r-1}$  from  $H$ 
            Add  $X_{i+k,j+r}$  to  $H$ 
        end for
         $Y_{i,j} \leftarrow \text{median}(H)$ 
    end for
end for

```

可见 是对每次处理的列进行了保存优化 但是 这只是在行上的移动 对于列方向的移动 也有优化的空间

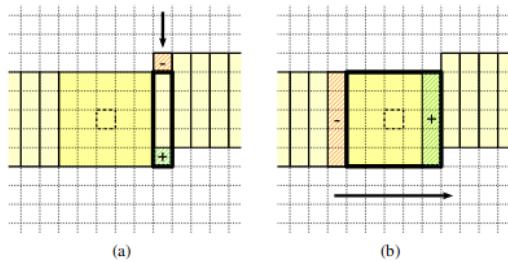


Fig. 2: The two steps of the proposed algorithm. (a) The column histogram to the right is moved down one row by adding one pixel and subtracting another. (b) The kernel histogram is updated by adding the modified column histogram and subtracting the leftmost one.

Algorithm 2 The proposed $O(1)$ median filtering algorithm.

Input: Image X of size $m \times n$, kernel radius r

Output: Image Y of the same size

```

Initialize kernel histogram  $H$  and column histograms  $h_{1\dots n}$ 
for  $i = 1$  to  $m$  do
    for  $j = 1$  to  $n$  do
        Remove  $X_{i-r-1,j+r}$  from  $h_{j+r}$ 
        Add  $X_{i+r,j+r}$  to  $h_{j+r}$ 
         $H \leftarrow H + h_{j+r} - h_{j-r-1}$ 
         $Y_{i,j} \leftarrow \text{median}(H)$ 
    end for
end for

```

of the image, the redundancy of information outside the image (e.g. solid color, or repeated edge pixels) correspondingly simplifies the

这里将操作分成了两部 首先是将变化的列中右下角的像素进行了更改 最大程度保存原来的数据 然后便是标准的列操作 这里的复杂度 由于只进行了一次遍历 所以达到了 $O(1)$

```
def fast_median(self, filepath):
    try:
        img = cv.imread(filepath)
        rows, cols = img.shape[:2]
        d = np.asarray(img)
        data = []
        for i in range(1, rows - 1):
            for j in range(1, cols - 1):
                if j == 1:
                    data = d[i - 1:i + 2:1, j - 1:j + 2:1]
                elif i == 1 and j != cols - 2:
                    data[:, 0] = data[:, 1]
                    data[:, 1] = data[:, 2]
                    data[:, 2] = d[i - 1:i + 2:1, j + 1]
                elif i != 1 and j == cols - 2:
                    data[0, 2] = data[1, 2]
                    data[1, 2] = data[2, 2]
                    data[2, 2] = d[i + 1, j + 1]
                    data[:, 0] = data[:, 1]
                    data[:, 1] = data[:, 2]
                    data[:, 2] = d[i - 1:i + 2:1, j + 1]
                data0 = data.reshape(9, 3)
                img[i, j] = np.mean(data0, axis=0)
        cv.imshow("mean", img)
        cv.waitKey(0)
    except Exception as e:
        print("ERROR: " + str(e))
```

三、总结

3.1 请总结本次实验的收获

这次实验难度不大 都是在已有的基础上进行操作 更多的是对算法实现细节的理解 以及商业用包中对相关算法的替代以及优化 更过的了解了图像处理的瓶颈和优化步骤

4.2 请给出对本次实验内容的建议

希望能够将CUDA内容去除 然后将快速中值变为必做内容