

Государственное образовательное учреждение высшего профессионального образования
Санкт-Петербургский национальный исследовательский университет
Информационных Технологий, Механики и Оптики
Факультет инфокоммуникационных технологий

**ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ № 2**

По Мультимедиа Технологиям

На тему: «Точные и локальные методы обработки изображений»

Проверил(а): Хлопотов М.В.

Работу выполнил(а):

Никончук А.П.
Студент(ка) группы К3242
Дневного отделения

Дата: «___» 201___ г.

Оценка: _____

Цель: Изучить существующие методы коррекции изображений

Задачи:

1. выполнить коррекцию контраста
2. выполнить светокоррекцию робастным линейным растяжением
3. выполнить цветокоррекцию операцией серый мир
4. зашумить изображение перцем и солью, затем подавить шум мелианным фильтром
5. выполнить операцию свертки с применением ядер сдвига на 1, усреднения, повышения резкости, размытия гаусса
6. выполнить повышение резкости unsharp mask

Выполнение работы:

При

Задача 1: Коррекция контраста (яркости) изображения

Используя стандартные функции постройте гистограмму яркостей изображения. Примените к изображению операцию «робастное линейное растяжение яркости» (в качестве порога возьмите 5-10% в зависимости от изображения).

Рисунок 1. Функция робастного линейного растяжения

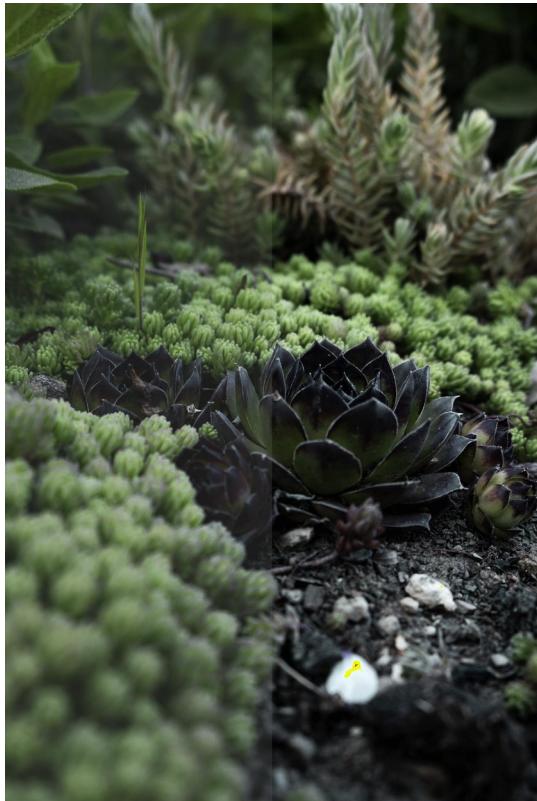
```
In [187]: def linearF(im, percent):
    values, bin_edges, patches = hist(im.ravel(), bins=range(256))
    YnotZeroValuesList = listOfPix(values)
    k = int(len(YnotZeroValuesList)*percent/100)
    new_min = YnotZeroValuesList[k]
    new_max = YnotZeroValuesList[len(YnotZeroValuesList)-k-1]

    # for virtuality dropped elements
    #lght_list = YnotZeroValuesList[0:k]
    #dark_list = YnotZeroValuesList[len(YnotZeroValuesList)-k:len(YnotZeroValuesList)]
    #print(lght_list, dark_list)

    y, u, v = rgbToYUV(im)
    new_y = np.clip(np.uint8((y[:, :] - new_min) *
                           (255 / (new_max - new_min))), 0, 255)
    rd = new_y[:, :] + 1.402*(v[:, :] - 128)
    gr = new_y[:, :] - 0.34414*(u[:, :] - 128) - 0.71414*(v[:, :] - 128)
    bl = new_y[:, :] + 1.772*(u[:, :] - 128)
    return dstack((rd, gr, bl)).astype('uint8')
```

На вход передается исходное изображение для обработки и процент (целое число) отбрасываемых светлых и темных пикселей в линейки распределения яркости. В первую очередь выбираются по гистограмме изображения значения яркостей для дальнейшей выборки. Для установки нового значения минимума и максимума выбирается следующий за последним (в зависимости от стороны обхода) отбрасываемым элементом. Затем происходит выделение канала яркости, его изменение и обратный перевод в пространство GRB. На выходе новое изображение с повышенной контрастностью.

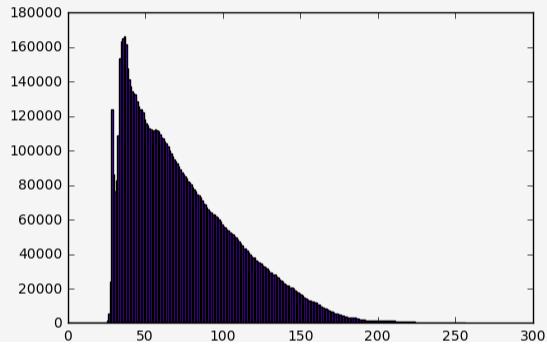
Рисунок 2. Результат работы функции робастного линейного растяжения яркости.
Склейенные входное и выходное изображения, карта разницы.



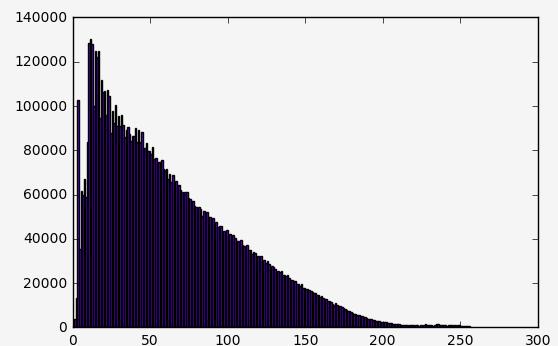
Гистограмма входного изображения,

гистограмма выходного изображения

```
In [83]: values, bin_edges, patches = hist(les2.ravel(), bins=range(257))  
plt.show()
```



```
In [181]: values, bin_edges, patches = hist(new_les2.ravel(), bins=range(257))  
plt.show()
```



Следующий вариант имеет несколько иной подход. исправлены ошибки, среди которых - взятие гистограммы по всему изображению, когда нужно только по яркости (хотя казалось бы это и должно отражаться таким образом). Прежде всего во втором варианте алгоритма для выборки новых границ гистограмма не строится. Вместо этого копия многомерного массива сворачивается в одномерный и сортируется. Затем по нему находятся новые минимум и максимум. Все вычисления производятся в формате float и только перед выводом "обрезаются" методом clip и переводятся в uint8, что позволяет производить более точные вычисления. Гистограмма результата показывает более равномерное распределение гаммы, а изображение зрительно заметную насыщенность (контрастность) цветов.

Значения гистограммы на минимуме или максимуме нарастает, в связи с чем увеличивается масштаб графика при том, что рост нулевого значения может быть незаметен, график визуально становится "приземленным". Явление происходит в связи с тем, что значений становится больше. все больше пикселей принимают значения крайние и после округления, естественно, большее количество пикселей будет иметь граничные значения.

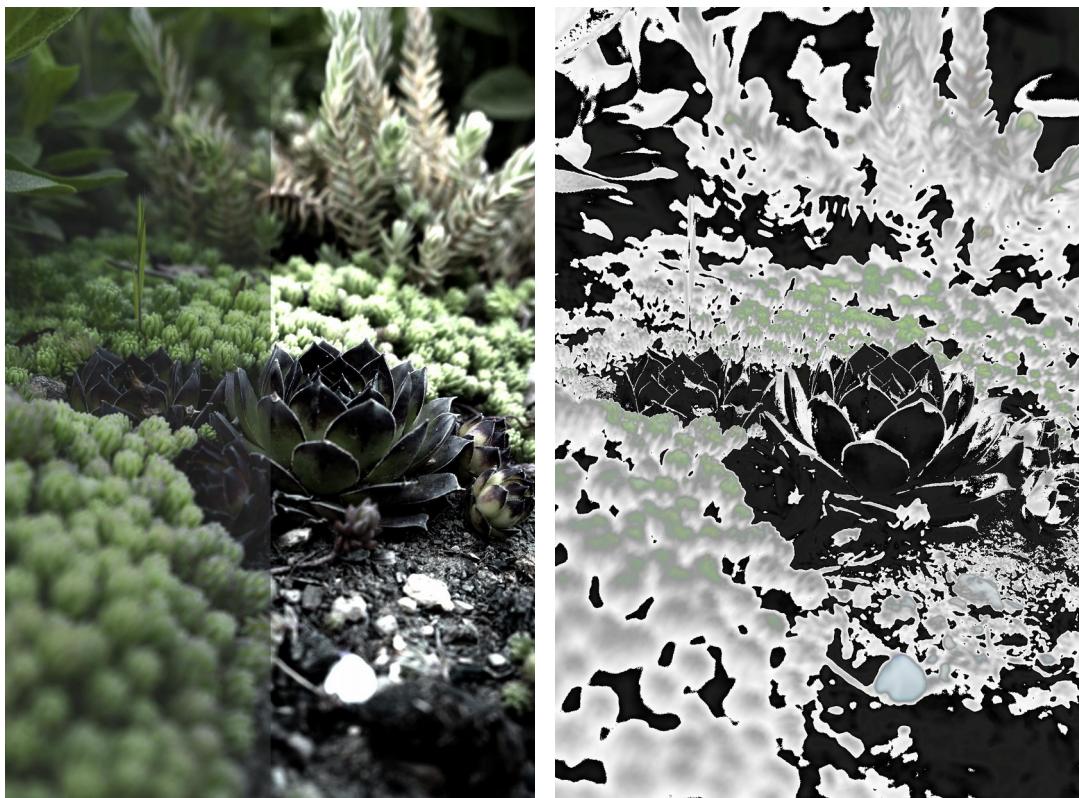
Рисунок 3. Второй вариант вариации робастного линейного растяжения

```
def TakeaList(llist):
    new_list = []
    for i in range(len(llist)):
        for j in range(len(llist[i])):
            new_list.append(llist[i][j])
    return new_list
def linearFinal(im, percent):
    y, u, v = rgbToYUV(im)

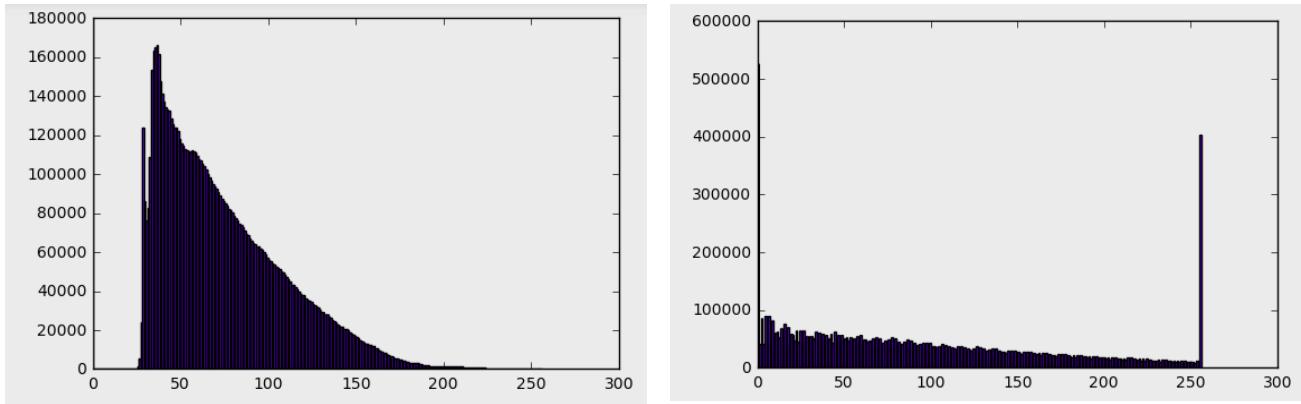
    y_s = y.copy().reshape(-1)
    y_s.sort()
    #print(y_s)
    k = int(len(y_s)*percent/100)
    new_min = y_s[k]
    new_max = y_s[len(y_s) - k - 1]

    new_y = (y[:, :] - new_min) * (255 / (new_max - new_min))
    rd = new_y[:, :] + 1.402*(v[:, :] - 128)
    gr = new_y[:, :] - 0.34414*(u[:, :] - 128) - 0.71414*(v[:, :] - 128)
    bl = new_y[:, :] + 1.772*(u[:, :] - 128)
    return clip(dstack((rd, gr, bl)), 0, 255).astype('uint8')
```

Рисунок 4. Результат работы второй вариации функции робастного линейного растяжения яркости.
Склейенные входное и выходное изображения, карта разницы.



Гистограмма входного изображения,
гистограмма выходного изображения



Задача 2: Коррекция цвета

2.1. Операция «линейное растяжение по каналам»

Рисунок 5. Функция линейного растяжения по каналам

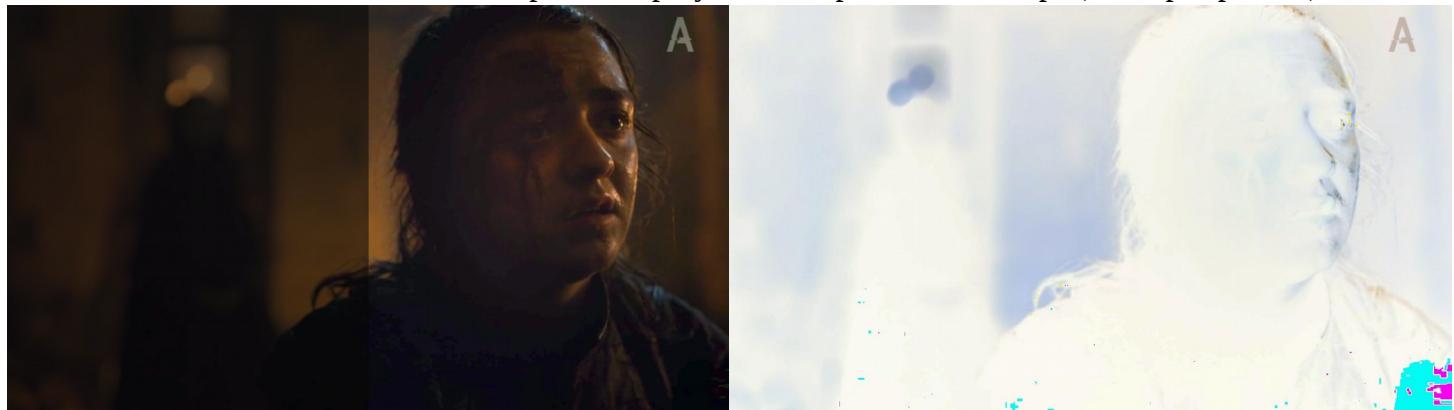
```

def ChannalLinear1(m, percent):
    values, bin_edges, patches = hist(m.ravel(), bins=range(256))
    YnotZeroValuesList = listOfPix(values)
    k = int(len(YnotZeroValuesList)*percent/100)
    new_min = YnotZeroValuesList[k]
    new_max = YnotZeroValuesList[len(YnotZeroValuesList)-k-1]
    # for virtuality dropped elements
    #lght_list = YnotZeroValuesList[0:k]
    #dark_list = YnotZeroValuesList[len(YnotZeroValuesList)-k:len(YnotZeroValuesList)]
    #print(lght_list, dark_list)
    m = (m - new_min) * (255 / (new_max - new_min))
    return m
def Linear1(im, percent):
    r = ChannalLinear1(im[:, :, 0], percent)
    g = ChannalLinear1(im[:, :, 1], percent)
    b = ChannalLinear1(im[:, :, 2], percent)
    return clip(dstack((r, g, b)), 0, 255).astype('uint8')

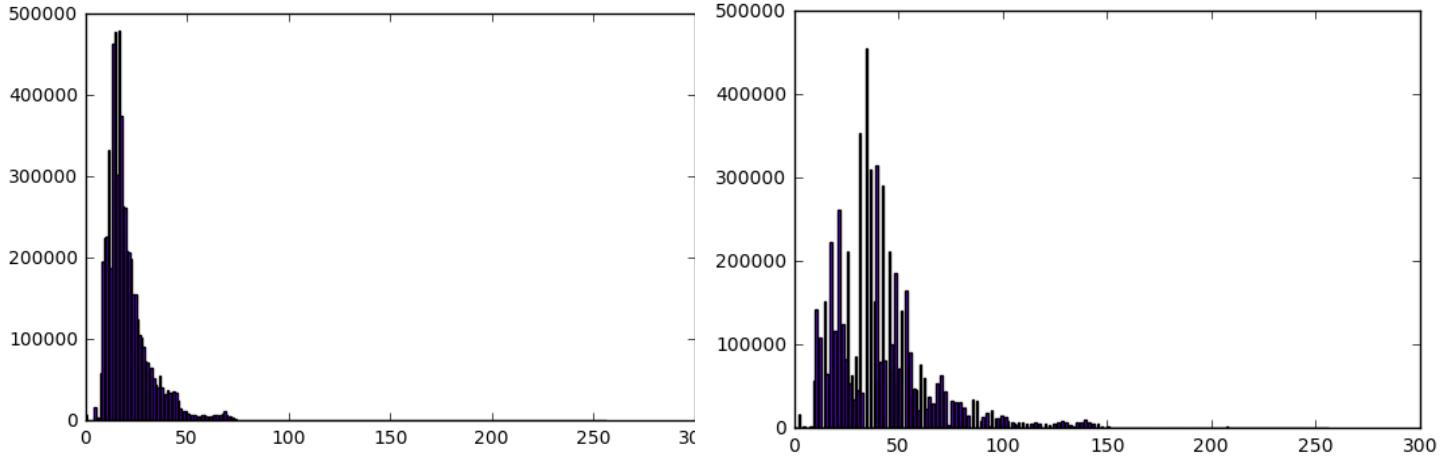
```

В зависимости от переданного процента отбрасываемых пикселей находятся новый минимум и максимум для каждого из 3х выделенных каналов – границы отброшенных значений, по которым применяются изменения при обходе по пикселям изображения.

Рисунок 6. Результат работы функции линейного растяжения по каналам.
Состыкованное изображение результатов применения операций, карта разницы,



Гистограмма входного и выходного изображений.



2.2. Калибровка цвета на основе операции «серый мир»

Рисунок 7. Вариант реализации операции серый мир

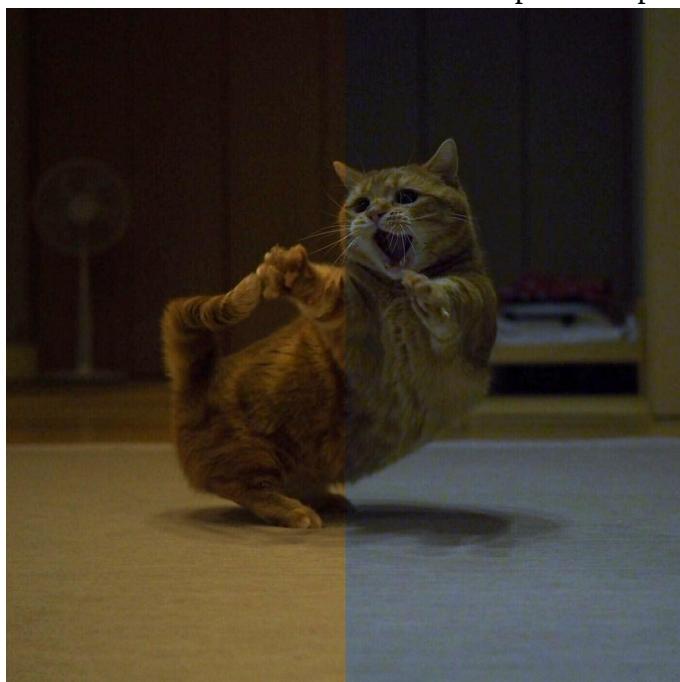
```

def findCannalAvg(m):
    count = 0
    size = m.shape[0]*m.shape[1]
    for row in m:
        for px in row:
            count += px
    return count/size
def greyCorrection(im):
    nR = findCannalAvg(im[:, :, 0])
    nG = findCannalAvg(im[:, :, 1])
    nB = findCannalAvg(im[:, :, 2])
    print(nR, nG, nB)
    avg = ( nR + nG + nB ) / 3 # общее среднее
    # поправочные коэффициенты
    r = clip( (im[:, :, 0] / (nR / avg)), 0 ,256 )
    g = clip( (im[:, :, 1] / (nG / avg)), 0 ,256 )
    b = clip( (im[:, :, 2] / (nB / avg)), 0 ,256 )
    return dstack((r, g, b)).astype('uint8')

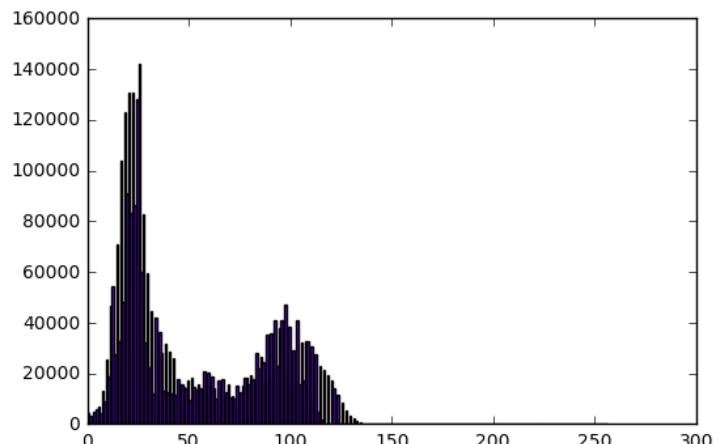
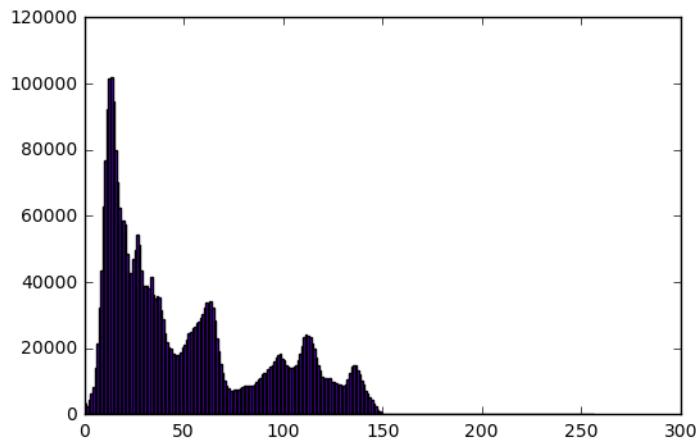
```

Модель предполагает, что средний уровень цвета по каждому из каналов должен быть одинаков. То есть средний цвет – это серый. В операции происходит усреднение значений цвета, сумма которых также усредняется. Считываются поправочные коэффициенты. Каждый из каналов изображения делим на полученный коэффициент. На выходе соранное поканально изображение.

Рисунок 8. Результат выполнения операции серого мира.
Состыкованное изображение результатов применения операций, карта разницы,



Гистограмма входного и выходного изображений



Задача 3: Шум

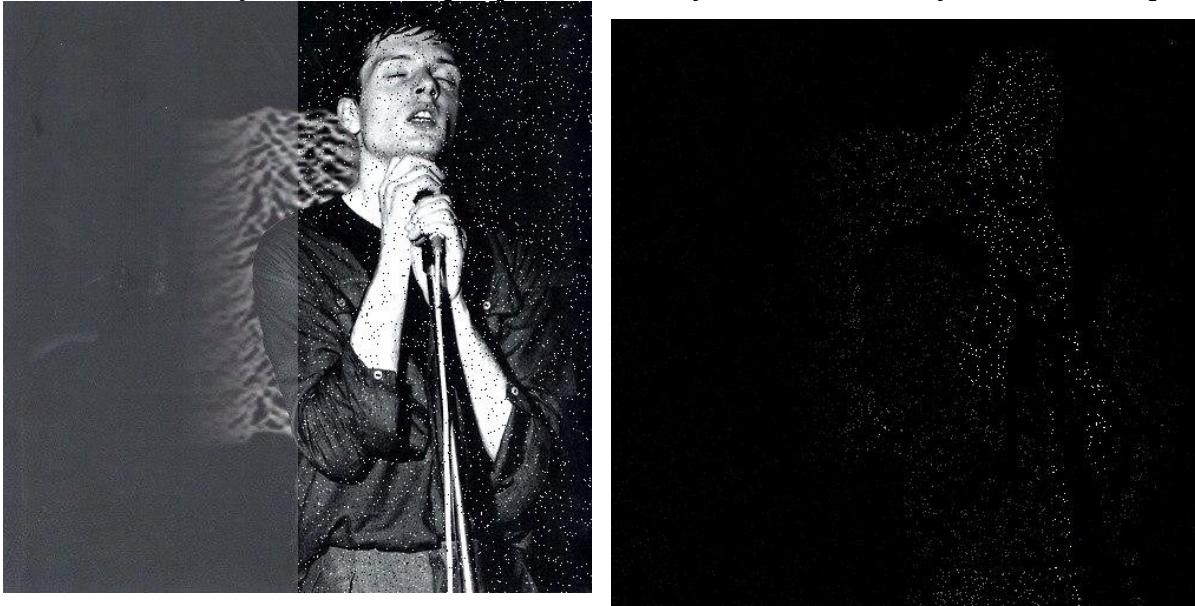
3.1. Зашумление «соль и перец»

Рисунок 9. Вариант реализации алгоритма наложения шума

```
def SaltPepper(im, percent):
    new_im = np.copy(im)
    size = im.shape[0]*im.shape[1]
    one_elem = im[0,0]
    matrix = []
    #matrix = [np.array(([0,0,0])).astype('uint8'),
    #          np.array(([255,255,255])).astype('uint8')]
    for i in [0, 255]:
        new_one = []
        for j in range(len(one_elem)):
            new_one.append(i)
        matrix.append(np.array((new_one)).astype('uint8'))
    for i in range(len(matrix)):
        count_new_elems = int(size* percent/100)
        #print(count_new_elems, size)
        for j in range (count_new_elems):
            x = random.randint(0, im.shape[0]-1)
            y = random.randint(0, im.shape[1]-1)
            new_im[x,y] = matrix[i]
    return new_im
```

Формируется матрица двух вариантов замены пикселя – полностью белый и полностью черный – в зависимости от количества элементов (каналов), хранимых пикселям. На выходе копия изображения с рандомным количеством наложенных крупинок шума.

Рисунок 10. Изображение с применением линейного робастного растяжения с добавлением шума, а также карта разницы между исходным и зашумленным изображениями



3.2. Подавление медианным фильтром (несколько размеров фильтра)

Рисунок 11. Зашумленное солью и перцем изображение с применением подавления медианным фильтром 3x3, а также карта разницы между скорректированным и исходным изображениями

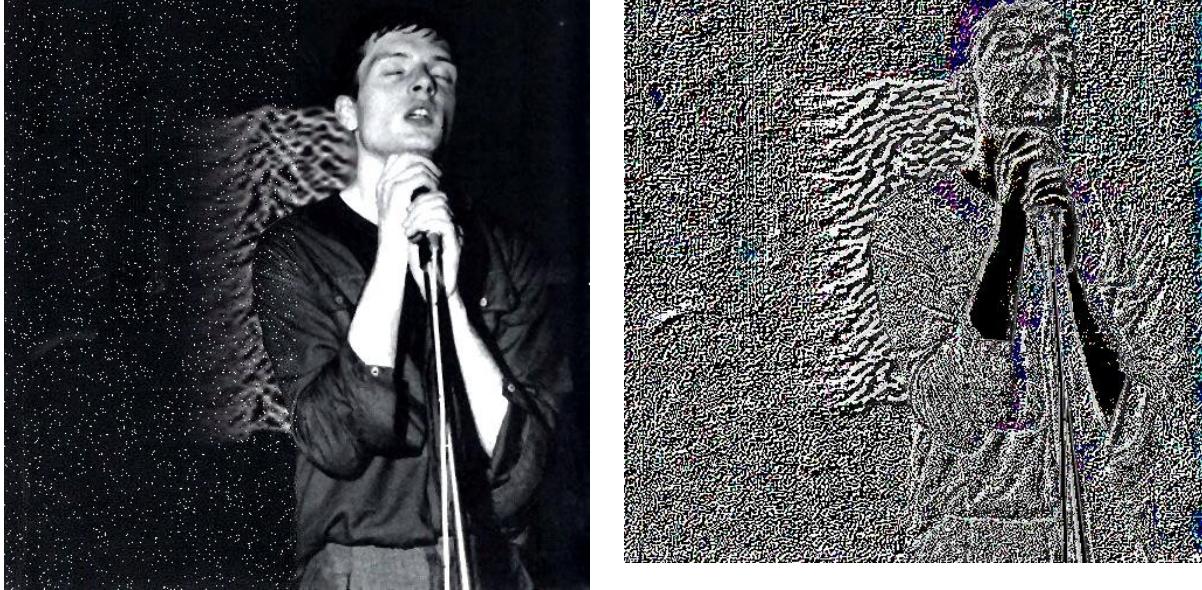


Рисунок 12. Зашумленное солью и перцем изображение с применением подавления медианным фильтром 5x5

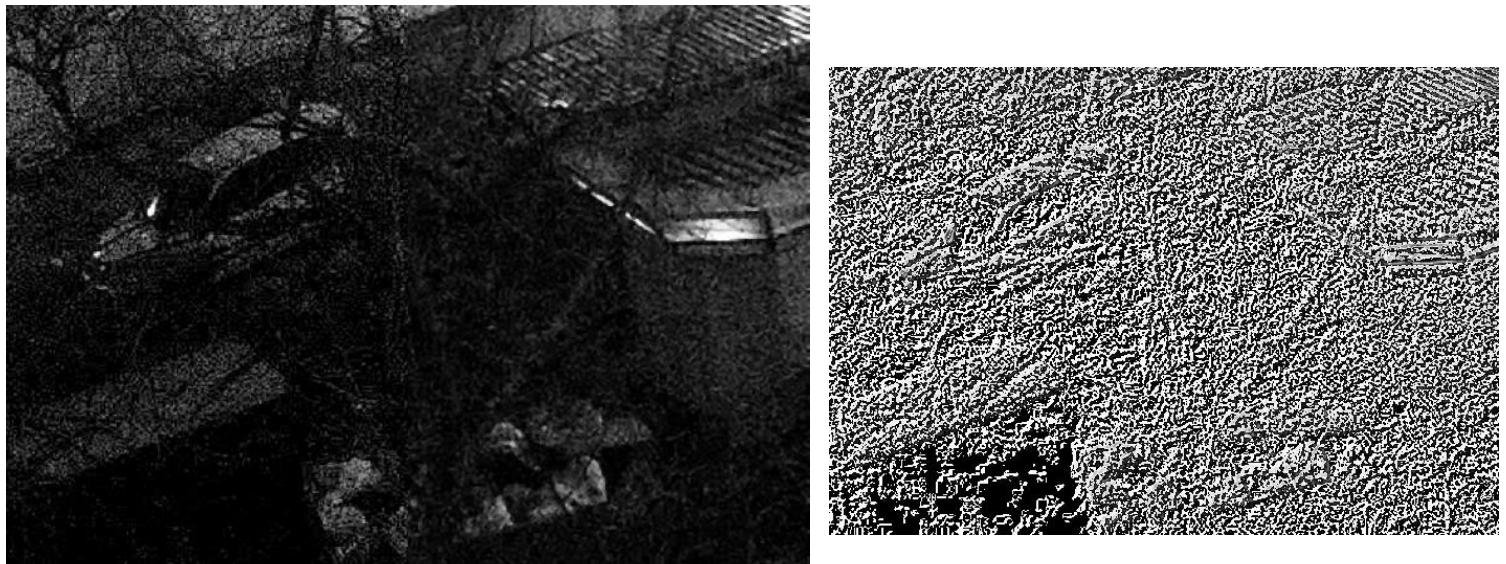


Главная функция MedianFilter1 принимает в качестве параметров исходное изображение и параметр фильтра, подходящим под маску $i \times i$, где i нечетное число, определяющее размер ядра. В алгоритме создается копия изображения, которая будет преобразована в выходное изображение. Функция MedianBox задает матрицу для фильтра, представляющую в виде массива массивов с индексами относительно центра ядра. После формирования фильтра происходит обход каждого пикселя изображения. Для каждого пикселя по матрице выбираются существующие элементы и передаются в функцию avgBoxElem3 единым массивом, где они сортируются и выбирается среднее значение из представленных.

Рисунок 13. Вариант реализации алгоритма подавления медианным фильтром

```
def MedianFilter1(img, parm):
    new_img = img.copy()
    arr = MedianBox(parm)
    for curr_row in range(img.shape[0]):
        for curr_col in range(img.shape[1]):
            args = []
            for i in range(len(arr)):
                if(len(arr[i])>2):
                    for j in range(len(arr[i])):
                        if(curr_row+i >= 0 and curr_row+i < img.shape[0]
                           and curr_col+j >= 0 and curr_col+j < img.shape[1]):
                            args.append(img[curr_row+i, curr_col+j])
            new_img[curr_row, curr_col] = avgBoxElem3(args)
    return new_img
```

Рисунок 14. Естественно зашумленное изображение с применением подавления медианным фильтром 3x3, а также карта разницы между скорректированным и исходным изображениями



Задача 4: Свертка

Задача свертки требует написания попутно алгоритмов-посредников: симметричного отражения по вертикали, горизонтали и диагонали. Однако вынесение в отдельную и воспроизведение рекурсивное при необходимости использование обвенчалось неудачей, связанной с возникновением ошибки “”.

Причина возникновения ошибки заключается в том, что `index` является тензорной символьической переменной (длинный скаляр). Поэтому, когда python пытается создать словарь, который необходим для «заданного» ввода, он пытается нарезать массив с помощью символьической переменной - чего он явно не может сделать, потому что у него еще нет значения (это только установить, когда вы вводите что-то в функцию).

Следовательно во избежание составные части, которые следовало бы вынести, внесены в общее тело функции.

Отражение по диагонали требует изменение координаты с участием величины размера матрицы – `n`. Ориентирование происходит по углам – верхний меняется на нижний, `new_img[i, j] → img[n-i-1, n-j-1]`. Разные углы соответственно добавляют величину по днише и по ширине изначального изображения.

По вертикали и горизонтали используется одноуровневый цикл, а сам алгоритм заключается в последовательном отдалении по заданной оси симметрии средством увеличения индекса. Потому каждая грань рамки, увеличивающей изображение для прохождения по нему матрицы фильтра, увеличивается одним прохождением. Процесс переноса данных на графический процессор медленный, поэтому гораздо лучше минимизировать количество передач, если это возможно. Что также позволяет уменьшить отчасти сложность алгоритма – плюс к появлению ошибки. [x]

Рисунок 15. Код отражений. Сверху вертикальный и горизонтальный, снизу диагональный

```
def addBorder(img, n):
    new_img = np.zeros((img.shape[0]+n*2, img.shape[1]+n*2, 3)).astype('uint8')
    new_img[n:new_img.shape[0]-n, n:new_img.shape[1]-n] = img
    for i in range(n):
        # top down left right
        new_img[i, n:img.shape[1]+n] = img[n-i, 0:img.shape[1]]
        new_img[img.shape[0]+n+i, n:img.shape[1]+n] = img[img.shape[0]-i-1, 0:img.shape[1]]
        new_img[n:img.shape[1]*n, i] = img[0:img.shape[1], n-i]
        new_img[n:img.shape[1]*n, img.shape[0]+n+i] = img[0:img.shape[1], img.shape[0]-i-1]

    for i in range(n):
        for j in range(n):
            # down-right down-left top-left top-right
            new_img[n+img.shape[0]+i, n+img.shape[1]+j] = img[img.shape[0]-i-1, img.shape[1]-j-1]
            new_img[n+img.shape[0]+i, j] = img[img.shape[0]-i-1, n-j-1]
            new_img[i, j] = img[n-i-1, n-j-1]
            new_img[i, n+img.shape[1]+j] = img[n-i-1, img.shape[1]-j-1]
    return new_img
```

Рисунок 16. отражение по горизонтали. Четыре отраженных угла выделены.
Вертикальное и горизонтальное отражение

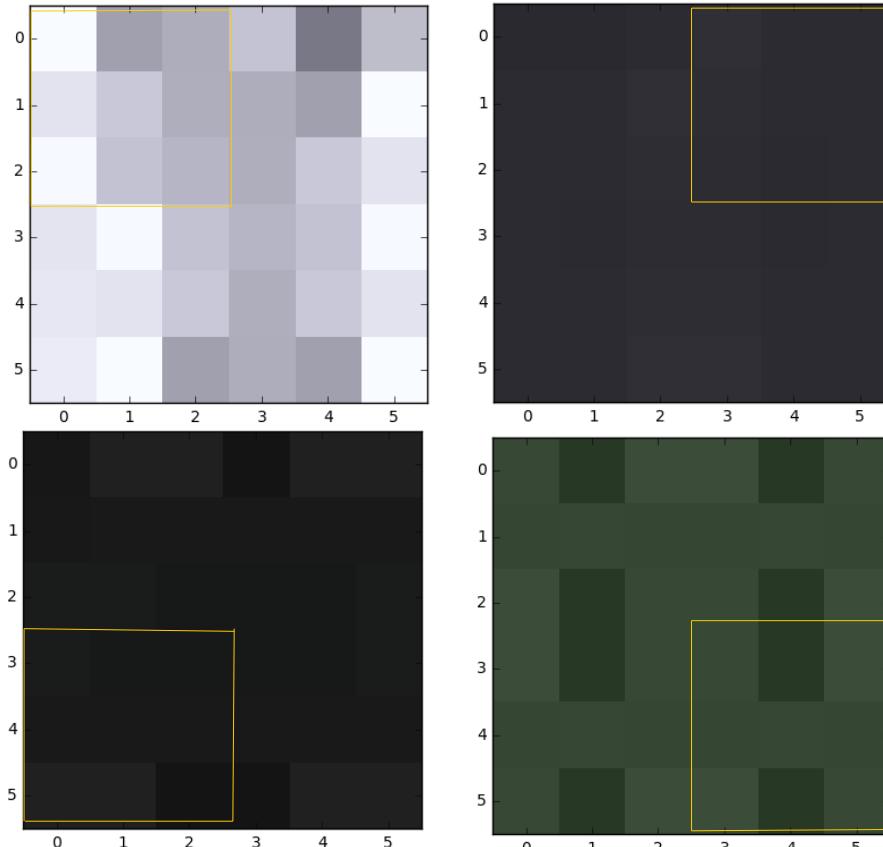
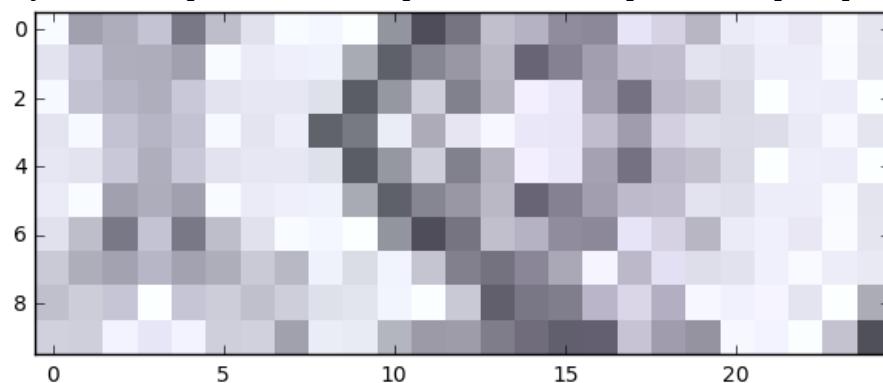


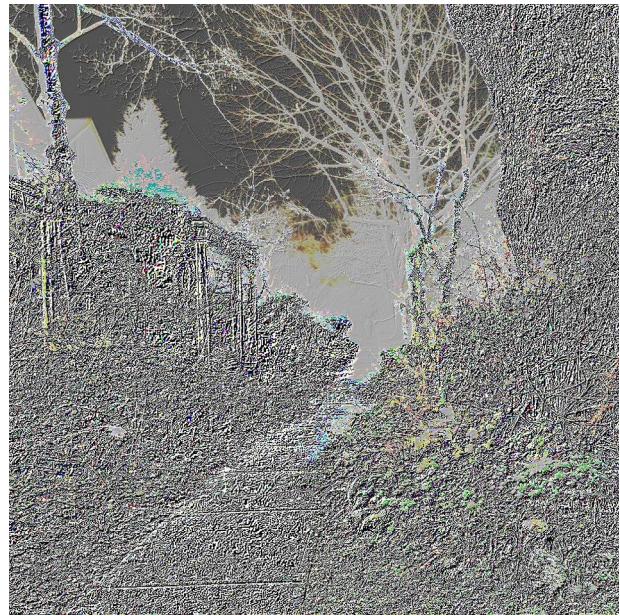
Рисунок 17. Вертикальное и горизонтальное отражения, пример верхнего левого угла



4.1. Усреднение

$n1on16 = np.array([[1/16, 2/16, 1/16], [2/16, 4/16, 2/16], [1/16, 2/16, 1/16]])$

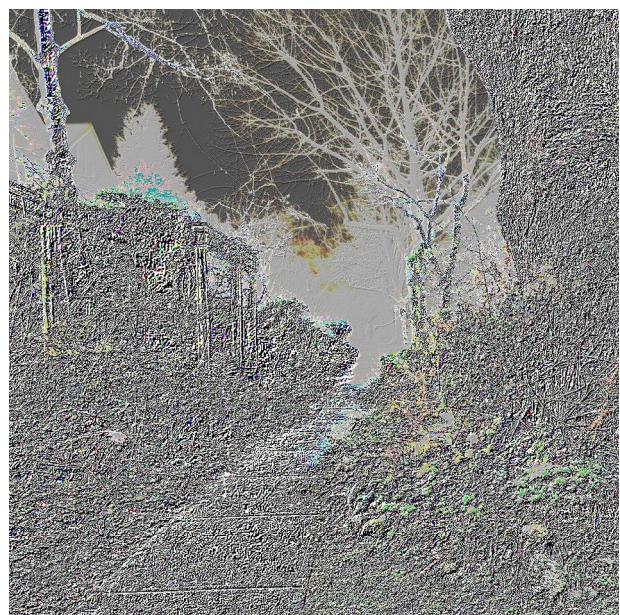
Рисунок 18. Результат свертки фильтра усреднения



4.2. Сдвиг на 1

$n1to1 = np.array([[0, 0, 0], [0, 0, 1], [0, 0, 0]])$

Рисунок 19. Результат свертки фильтра сдвига на единицу



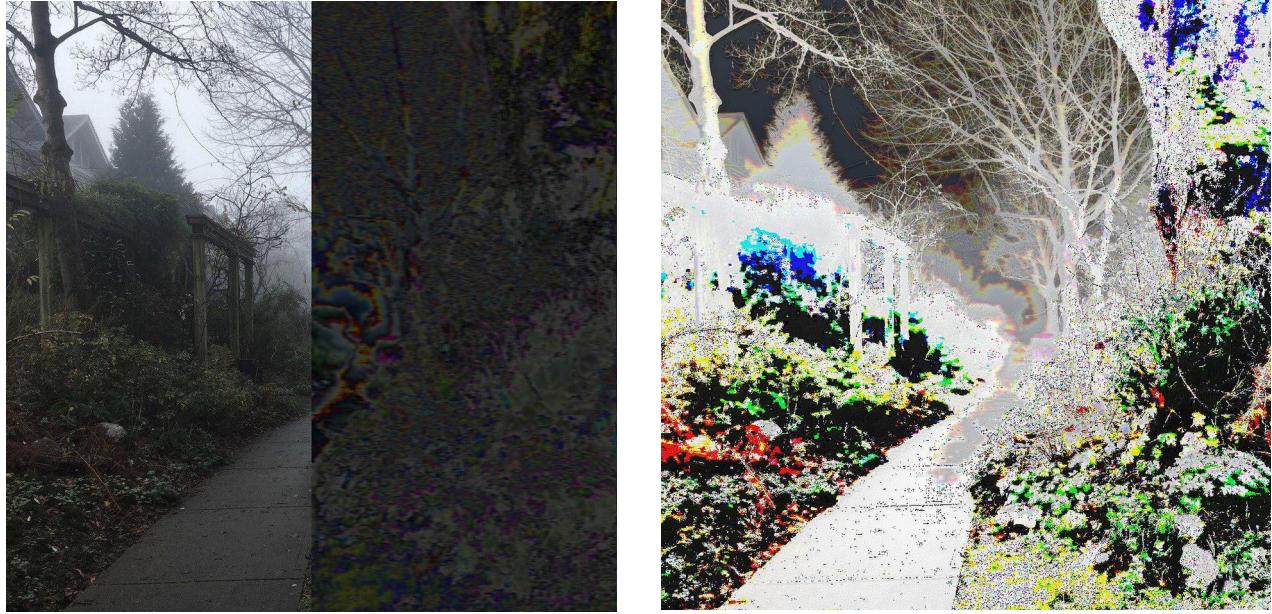
4.3. Гаусс

матрица: $nG = np.array($

```
[[0.003, 0.013, 0.022, 0.013, 0.003],  
 [0.013, 0.059, 0.097, 0.059, 0.013],  
 [0.022, 0.097, 0.159, 0.097, 0.022],
```

[0.013, 0.059, 0.097, 0.059, 0.013],
[0.003, 0.013, 0.022, 0.013, 0.003]])

Рисунок 20. Результат свертки с гауссовым фильтром (разница)



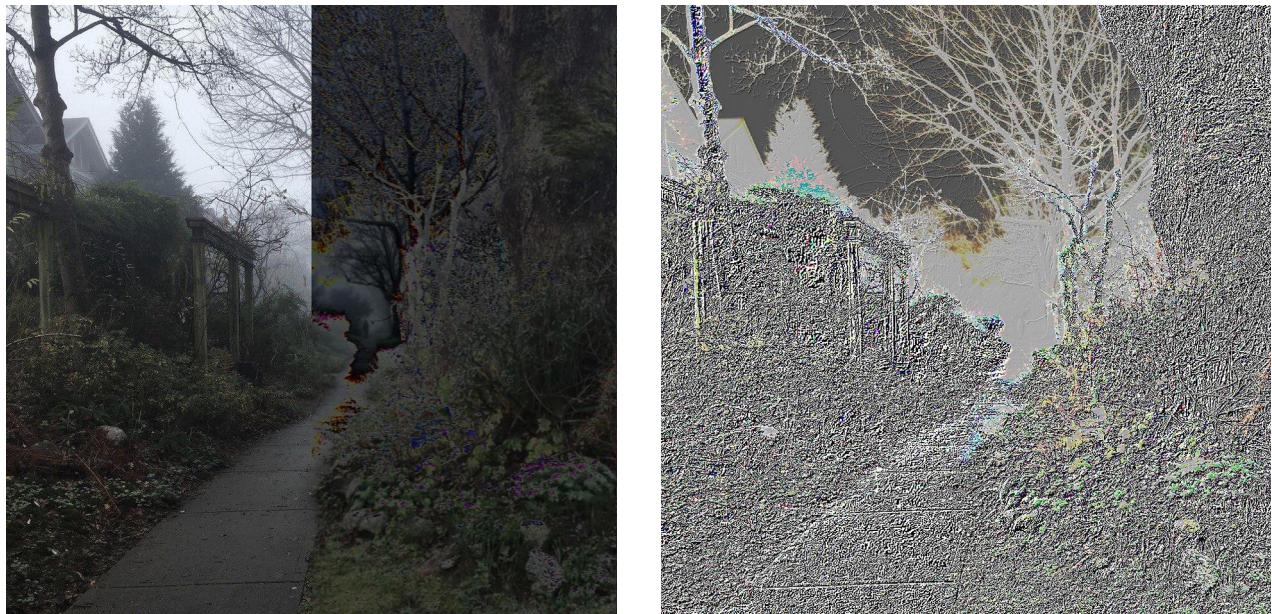
4.4. Повышение резкости

разницы между фильтрами $1/9$ $n1on9$ и 9 окруженного -1 $n9$ нет

$n1on9 = \text{np.array}([[1/9, 1/9, 1/9], [1/9, 1/9, 1/9], [1/9, 1/9, 1/9]])$

$n9 = \text{np.array}([[-1, -1, -1], [-1, 9, -1], [-1, -1, -1]])$

Рисунок 21. Свертка с применением фильтра повышения резкости



сравнение результата работы собственного фильтра со встроенными функциями

Рисунок 22. Код свертки

```
def bundle2(img, nuclear):
    n = len(nuclear) - 1
    print(n)
    mid_img = addBorder(img.copy(), len(nuclear))
    res_img = img.copy()
    for i in range(n, n+img.shape[0]):
        for j in range(n, n+img.shape[1]):
            #print(i-n, ':', i+n-1, ' ', j-n, ':', j+n-1)
            #print(sum(sum(mid_img[i-n:i+n-1, j-n:j+n-1])/ (n+1)), sum(sum(nuclear)/ (n+1)))
            res_img[i-n-1, j-n-1] = sum(sum(mid_img[i-n:i+n-1, j-n:j+n-1])/ (n+1))*sum(sum(nuclear)/ (n+1))
            #mid_img[i-n:i+n-1, j-n:j+n-1] * nuclear[:, :]
    return img_as_ubyte(np.clip(img_as_float(res_img), 0, 1))
#return res_img.astype('uint8')
```

Для работы алгоритма необходимо увеличить размер изображения, дополнив рамкой размером в половину рамки отраженными в противоположную сторону (зеркально) пикселями изображения, прилегающие к границе. Сам алгоритм заключается в том, что происходит прохождение по изображению внутри рамки и каждый пиксель вычисляется заново в зависимости от его окружения – обрезается зона по рамке дважды суммируется и умножается на такую же двойную сумму ядра.

Рисунок 23. Дополнительный пример исходное изображение и результат свертки с ядром фильтра усреднения



Задача 5: Резкость

5.1. Гауссов фильтр с разными ядрами и разными дисперсиями (“unsharp mask”)

В результате работы встроенного фильтра гаусса получено размытое изображение без артефактов. Прибавленная разница изображений начального и полученного применением гауссовского фильтра увеличивает резкость изображения.

Рисунок 23. Изображение с применением гауссовского размытия и разница с исходным

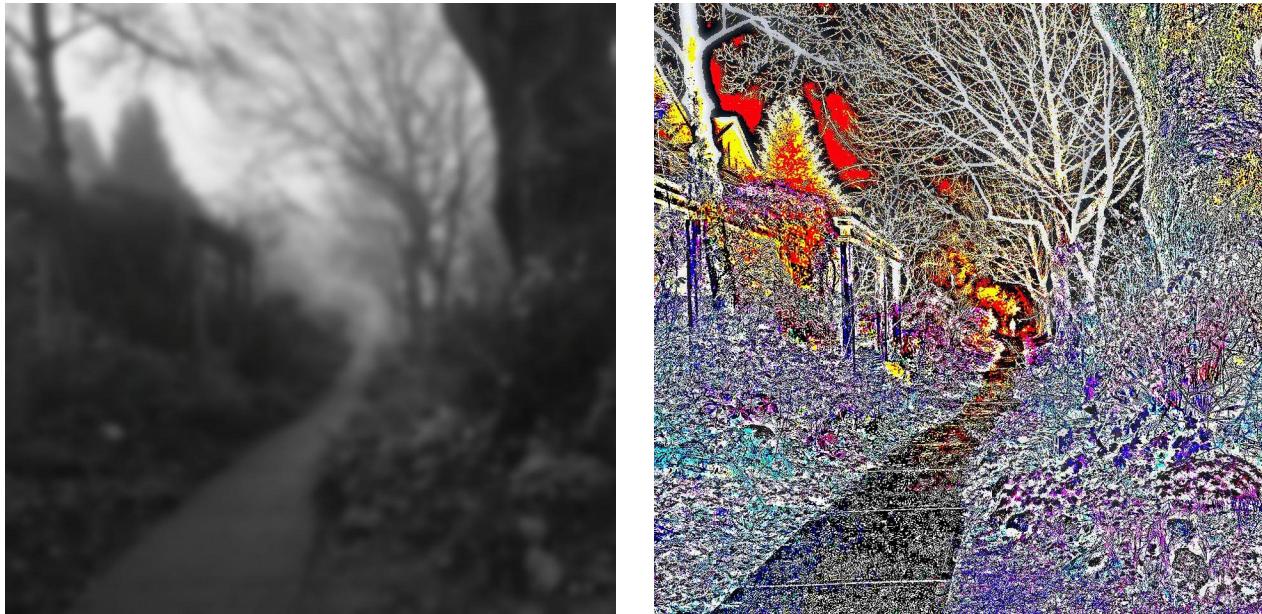


Рисунок 24. Сравнение изображения исходного и с увеличенной резкостью с прибавлением разницы от гауссовского размытия.



Выводы:

В результате выполнения лабораторной работы изучены представленные алгоритмы, отработан навык написания фильтров изображений, получена практика работы с гистограммами, попробован встроенный фильтр гаусса. Частные выводы:

6. В частности фильтр повышения резкости и unsharp mask (прибавление разницы гауссовского размытия с исходным изображением) дают похожий результат.

5. Чем больше ядро, тем заметнее разница скорректированного фильтром изображения. Можно отметить, что применение усреднения дает изображение утемненное, менее контрастное, приглушенное, с эффектом грязного или затонированного стекла. Повышение резкости мало заметно при увеличении ядра, цвета становятся светлее, не теряя свою глубину, контрастнее, при сохранении видимости тумана (изображен туман).

Было выбрано изображение с ветками деревьев чтобы показать искажения – так как происходит перерасчет пикселя в зависимости от соседних “суммирование” далеких оттенков дает усреднение, не вписывающееся в общий контекст. В результате изображении искажения покрывают всю поверхность, что показывает, что на изображении было гораздо больше мелких деталей, чем предполагалось первоначально.

4. (кроме 5 пункта) нужно использовать размер медианного фильтра шумоподавление с учетом “степени” шумозашумленности, так как больший фильтр помимо выполнении функции шумоподавления размазывает изображение

3. Применение линейного растяжения по каналом корректирует соответственно освещение изображения, изображение становится заметно освещеннее и от этого контрастнее

2. Операция серого мира корректирует изображение, убирая цветовой оттенок, цвета изображения становятся естественнее

1. Робастное растяжение растягивает цвета изображения, прореживая яркостный канал. Изображение становится ярче, на глаз недостаток цветов от прореживания не заметен, однако несколько применений засвеченные и затемненные участки – границы яркости – дают искажения.

Почти все алгоритмы после определенного этапа обработки (на какой-то стадии или при единственном исполнении) даже при применении обрезании цветовых границ дают цветовые искажения (яркие пиксели как будто из float палитры), результат которых связан с несовершенством алгоритмов – непродуманностью выхода за границы или отсечения выбора.