1 Реализация методов шумоподавления

1.1 Реализовать модель аддитивного шума, используя Гауссовскую случайную величину с распределением $N(0, \sigma^2)$

Модель формирования аддитивного шума для 8-битных значений интенсивностей пикселей изображения можно представить следующим образом:

$$I'_{y,x} = Clip(I_{y,x} + N_{y,x}, 0, 255)$$

где $I'_{y,x} -$ значение интенсивности компоненты I', полученной в результате обработки исходной компоненты I, $I_{y,x} -$ значение интенсивности, а $N_{y,x} -$ значение шума на позиции пикселя с координатами (y, x).

Для реализации модели аддитивного шума требуется сгенерировать случайную величину N, распределенную по нормальному закону. Генерация случайной величины, распределенной по нормальному закону, осуществляется с помощью преобразования Бокса-Мюллера.

В результате такого преобразования значения пикселей яркостной компоненты с глубиной 8 бит могут выйти за границы интервала [0;255]. В этом случае результат подвергается дополнительной операции клиппирования $Clip()$, которая описывается формулой:

$$Clip(I_{y,x}, min, \max) = \begin{cases} min, \text{если } I_{y,x} < min \\ max, \text{если } I_{y,x} > ma[\\ I_{y,x}, \text{иначе} \end{cases}$$

Исходное изображение имеет вид:

Рисунок 1 - Исходное изображение

В результате формирования аддитивного шума были получены следующие изображения:

Рисунок 2 – Использование гауссовской случайной величины с распределением N(0,100), $\sigma = 10$

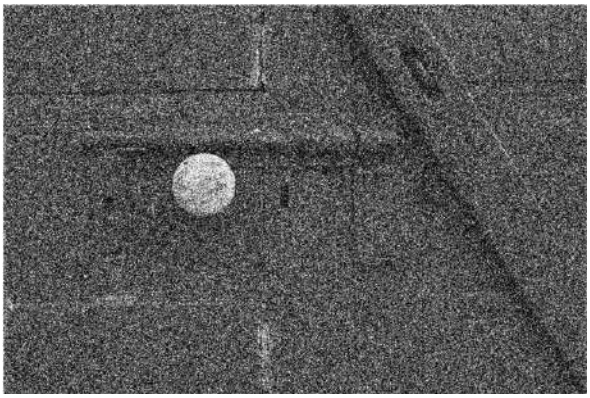


Рисунок 3 - Использование гауссовской случайной величины с распределением N(0,6400), $\sigma = 80$

Из полученных изображений можно заметить, что с увеличением параметра σ усиливается и искажение изображения.

1.2 Реализовать модель импульсного шума

Модель формирования импульсного шума для 8-битных значений интенсивностей описывается следующей формулой:

$$I'_{y,x} = \begin{cases} 0, \text{с вероятностью } p_a \\ 255, \text{с вероятностью } p_b \\ I_{y,x}, \text{с вероятностью } 1 - p_a - p_b \end{cases}$$

где p_a, p_b — параметры модели импульсного шума.

В результате формирования импульсного шума были получены изображения:

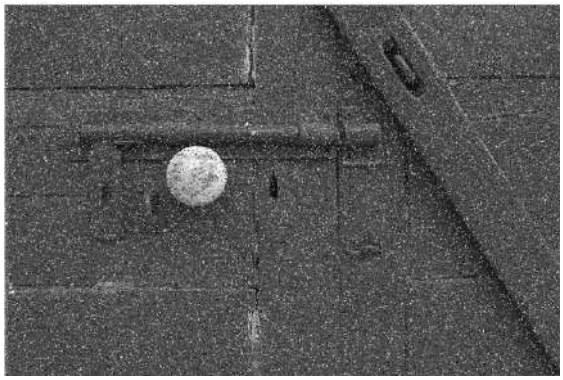


Рисунок 4 - Модель импульсного шума с параметрами p_a= 0.05 и p_b= 0.05

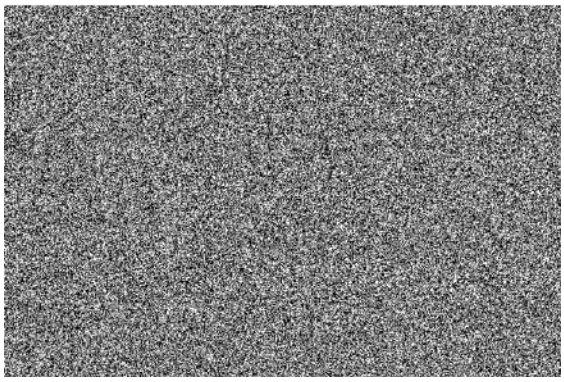


Рисунок 5 – Модель импульсного шума с параметрами p_a = 0.5 и p_b = 0.5

Из полученных изображений становится видно, что импульсный шум более выражен чем аддитивный даже при небольшом значение вероятности, т.к. зашумлённые пиксели принимают крайние значения 0 или 255.

1.3 Построить графики PSNR(σ) для модели аддитивного шума и PSNR(p_a, p_b) для модели импульсного шума.

Критерием оценки искажений, вносимых в изображение, является пиковое отношение сигнал/шум PSNR. В используемой разрядной сетке L значениче PSNR вычисляется по формуле:

$$PSNR = 10lg\frac{WH(2^L - 1)^2}{\sum_{i=1}^{H}\sum_{j=1}^{W}(I_{i,j} - I'_{i,j})}$$

где W – ширина изображения, H – высота изображения, $I_{i,j}$ – значение яркостной компоненты с координатами (i, j), $I'_{i,j}$ – значение яркостной компоненты в искаженному изображении с координатами (i, j).

Были получены следующие значения $PSNR(\sigma)$ для модели аддитивного шума:

```
PSNR = 47.8385
PSNR = 28.187
PSNR = 18.73
PSNR = 14.7127
PSNR = 11.5519
```

Рисунок 6 - Значения PSNR для $\sigma = 1, 10, 30, 50, 80$

График имеет вид:

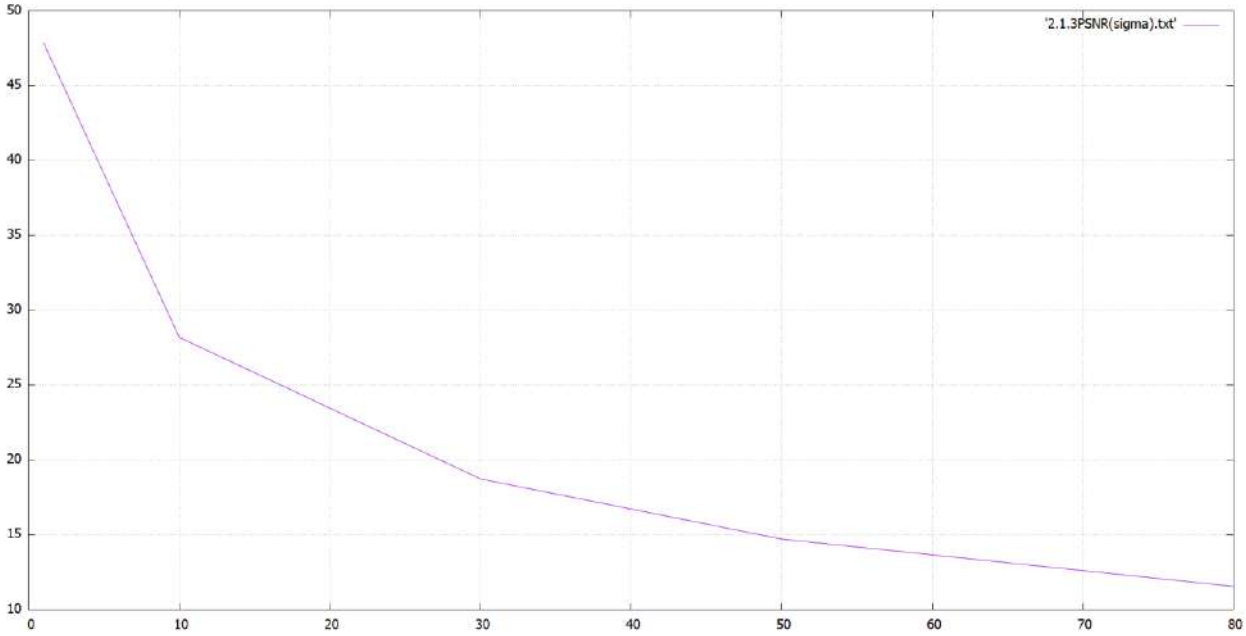


Рисунок 7 – График $PSNR(\sigma)$ *для модели аддитивного шума*

Были получены следующие значения $PSNR(p_a, p_b)$ для модели импульсного шума:

```
PSNR = 17.5569
PSNR = 15.1585
PSNR = 11.1766
PSNR = 8.3228
```

Рисунок 8 - Значения PSNR для $p_a = p_b = 0{,}025, 0{,}05, 0{,}125, 0{,}25$

График имеет вид:

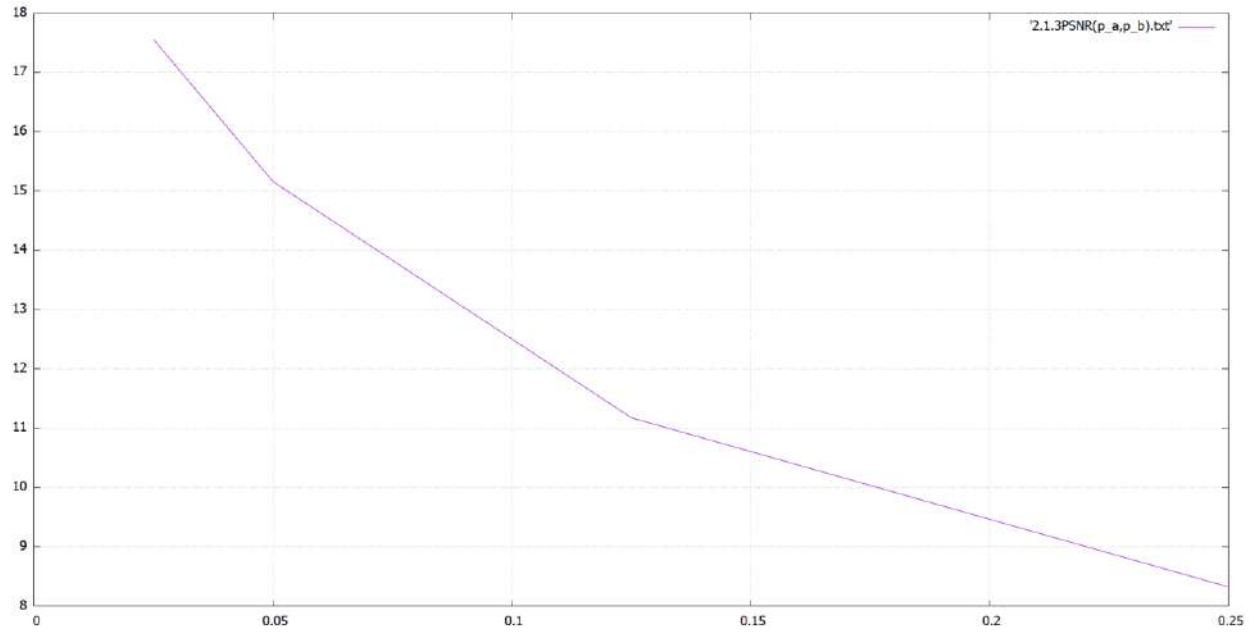


Рисунок 9 – График $PSNR(p_a, p_b)$ для модели импульсного шума

По результатам графиков видно, что импульсный шум вносит большие потери в качество картинки даже при малых параметрах.

2 Обработка изображений с Гауссовским шумом

2.1 Реализовать метод скользящего среднего.

В методе скользящего среднего значения весовых коэффициентов постоянны и не зависят от расстояния до центрального пикселя. Все весовые коэффициенты равны единице, в связи с этим выход фильтра рассчитывается по следующей формуле:

$$I'_{y,x} = \frac{1}{(2R+1)^2} \sum_{k=-R}^{R} \sum_{m=-R}^{R} I_{i+k,j+m}$$

где R – радиус фильтра.

Фильтр скользящего среднего вносит специфическое размытие в области резких перепадов интенсивностей на изображении.

В результате наложения аддитивного шума с параметром $\sigma = 30$ и применения фильтра, скользящего среднего с параметром $R = 2$ были получены изображения:

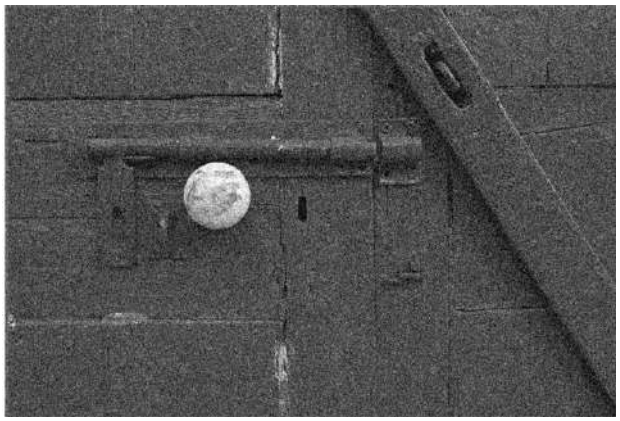


Рисунок 10 - Изображение с аддитивным шумом с параметром $\sigma = 30$

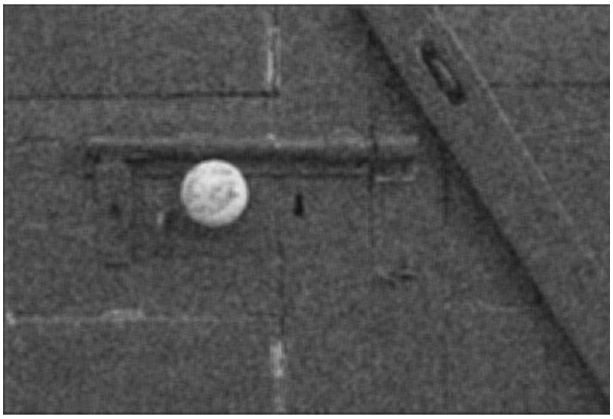


Рисунок 11 - Изображение, полученное после применения фильтра методом, скользящего среднего при R = 2

В результате фильтрации изображение получилось немного размытым, но шум все же удалось сгладить, значение PSNR тоже подросло.

2.2 Подобрать размер окна R для метода, скользящего среднего, который бы позволил максимизировать значение PSNR, вычисляемое по оригинальному изображению и зашумленному изображению после фильтрации

Рассмотрим модель шума с параметрами $\sigma = 1, 10, 30, 50, 80$. Для каждой из них будет рассчитано значение PSNR после применения фильтрации методом скользящего среднего с радиусом R = 1, 2, 3, 4 и 5. В результате работы программы были получены следующие значения:

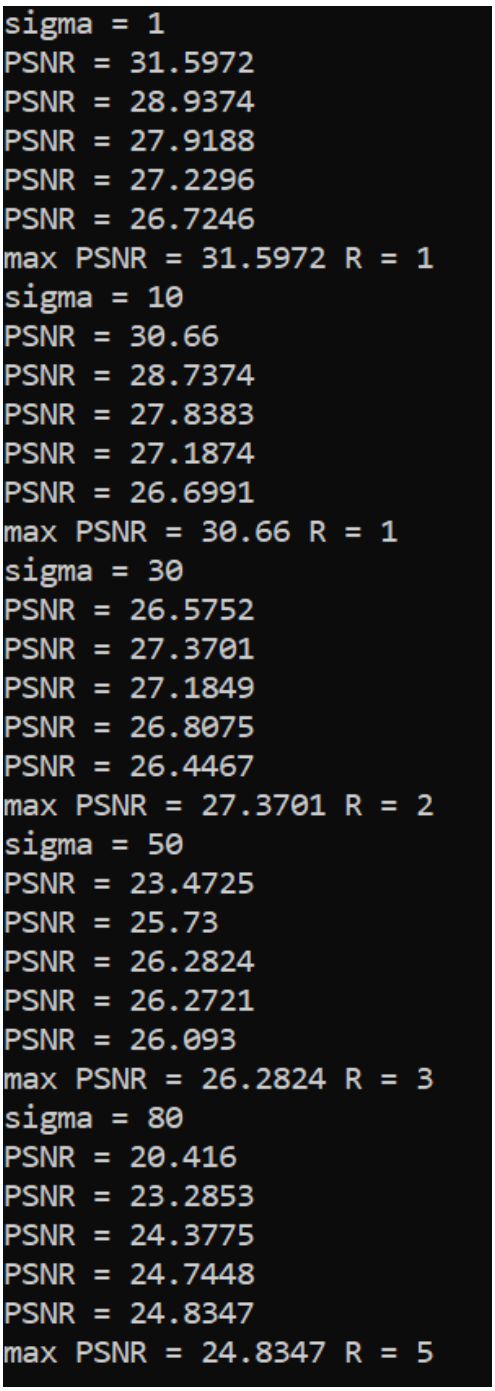


Рисунок 12 - Рассчитанные значения PSNR для различных R и σ

Сравнивая результаты, можно заметить, что максимальное значение PSNR достигается при маленьких значениях σ для малых R, однако с увеличением шума максимальное значение достигается R тоже должно увеличиваться.

2.3 Реализовать метод Гауссовской фильтрации.

Функции расстояния для расчета значений весовых коэффициентов формируется на базе функции Гаусса от двух переменных (отсюда и название фильтра):

$$w_{k,m} = \exp\left(\frac{-(k^2 + m^2)}{2\sigma^2}\right)$$

где $\sigma -$ параметр фильтра, определяющий скорость убывания коэффициентов $w_{k,m}$ по мере удаления от позиции центрального пикселя.

В процессе выполнения фильтрации входного (зашумленного) изображения $I[H \times W]$ формируется новое изображение с теми же размерами $I'[H \times W]$. Каждый пиксель $I'_{y,x}$ формируется в результате применения некоторого оператора к пикселю $I_{y,x}$ и подмножеству соседних с ним пикселей, образующих так называемую (апертуру) фильтра:

$$I'_{y,x} = \frac{1}{Z} \sum_{k=-R}^{R} \sum_{m=-R}^{R} w_{k,m} I_{y+k,x+m}$$

где $R -$ радиус фильтра, определяющий апертуру, $w_{k,m} -$ весовые коэффициенты фильтра, Z — коэффициент нормировки, рассчитывающийся как

$$Z = \sum_{k=-R}^{R} \sum_{m=-R}^{R} w_{k,m}$$

В окрестности контуров (резких перепадов) применение линейных фильтров приводит к “размытию” контура. Поэтому данный класс фильтров также называют фильтрами размытия (в англоязычной литературе Blur).

В результате наложения аддитивного шума с параметром $\sigma = 30$ и после применения фильтра Гаусса были получены изображения:

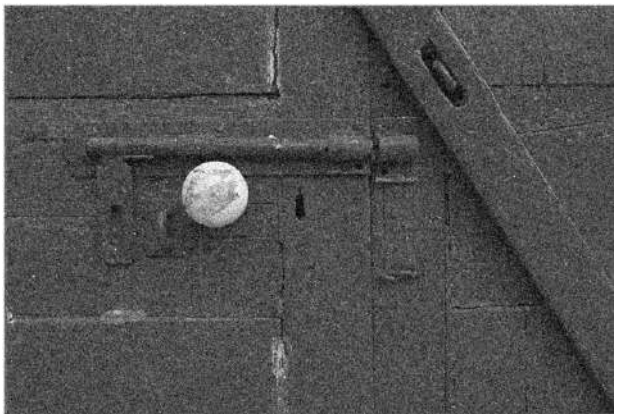


Рисунок 13 - Изображение с аддитивным шумом с параметром σ=30

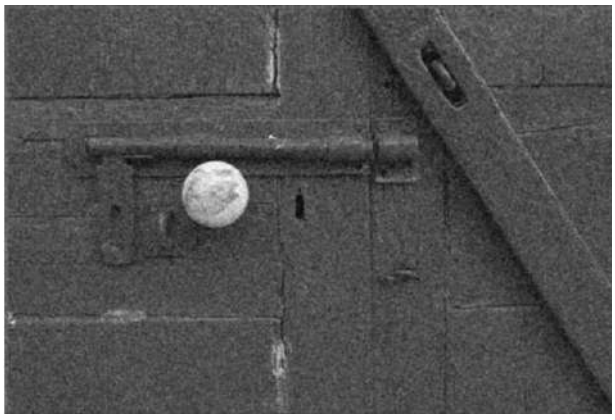


Рисунок 14 - Изображение, полученное после применения фильтра Гаусса

После фильтрации опять же получается некоторое размытие картинки, однако шум сглаживается.

2.4 Построить графики PSNR(σ) для нескольких фиксированных значений R, определяемых преподавателем.

Для построения графиков будут зафиксированы 3 значения для R (параметра фильтра) 1, 3, 5 и по пять зависимостей на каждом (для каждой сигмы (параметра шума)) PSNR(сигма(параметра фильтра)). Сигму (параметр фильтра) взяты: 0.1, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 2, а сигма (параметр шума) принимает значения 1, 10, 30, 50 и 80. Таким образом были получены графики:

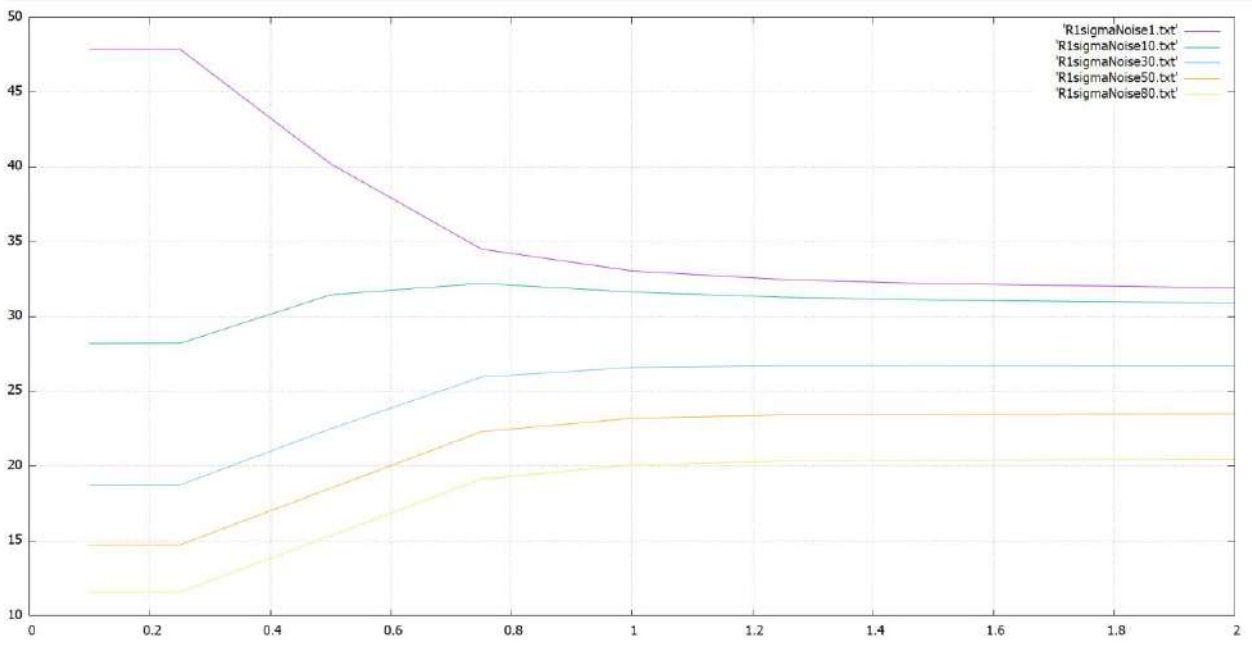


Рисунок 15 – График $PSNR(\sigma_{\text{фильтр}})$ для R = 1

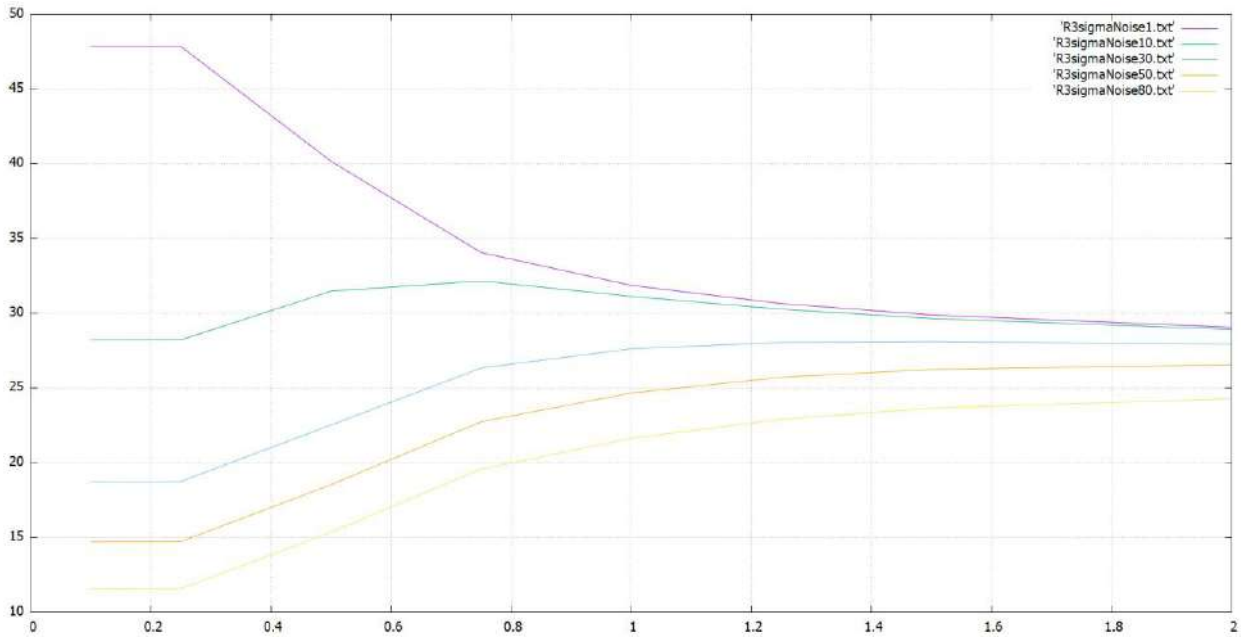


Рисунок 16 - График $PSNR(\sigma_{\text{фильтр}})$ для R = 3

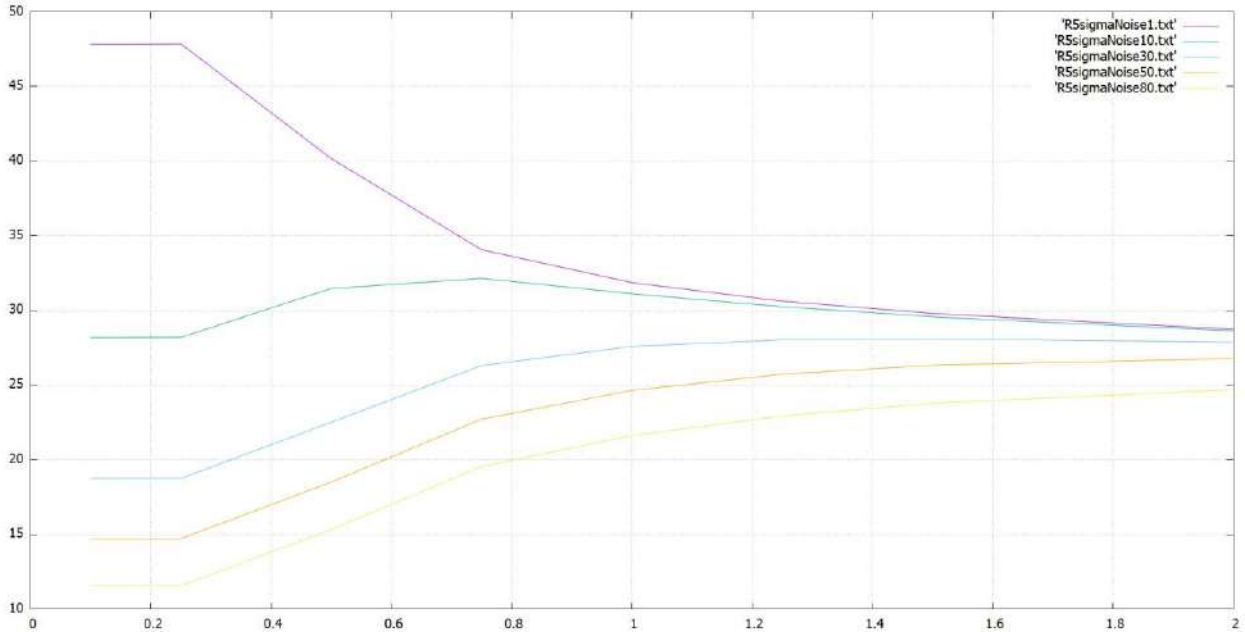


Рисунок 17 - График $PSNR(\sigma_{\text{фильтр}})$ для R = 5

Из полученных при значениях $\sigma_{\text{фильтр}} > 1$ значения PSNR при всех фильтра рассматриваемых шумах и значениях R перестают увеличиваться, а при малых шумах даже начинает уже уменьшаться.

2.5 Подобрать параметр σ для метода Гауссовской фильтрации, который бы позволил максимизировать значение PSNR, вычисляемое по оригинальному изображению и зашумленному изображению после фильтрации

Необходимо определить лучшие параметры фильтра (R и сигма) для каждой сигмы (параметра шума). Таким образом, при анализе полученных графиков, были получены следующие результаты:

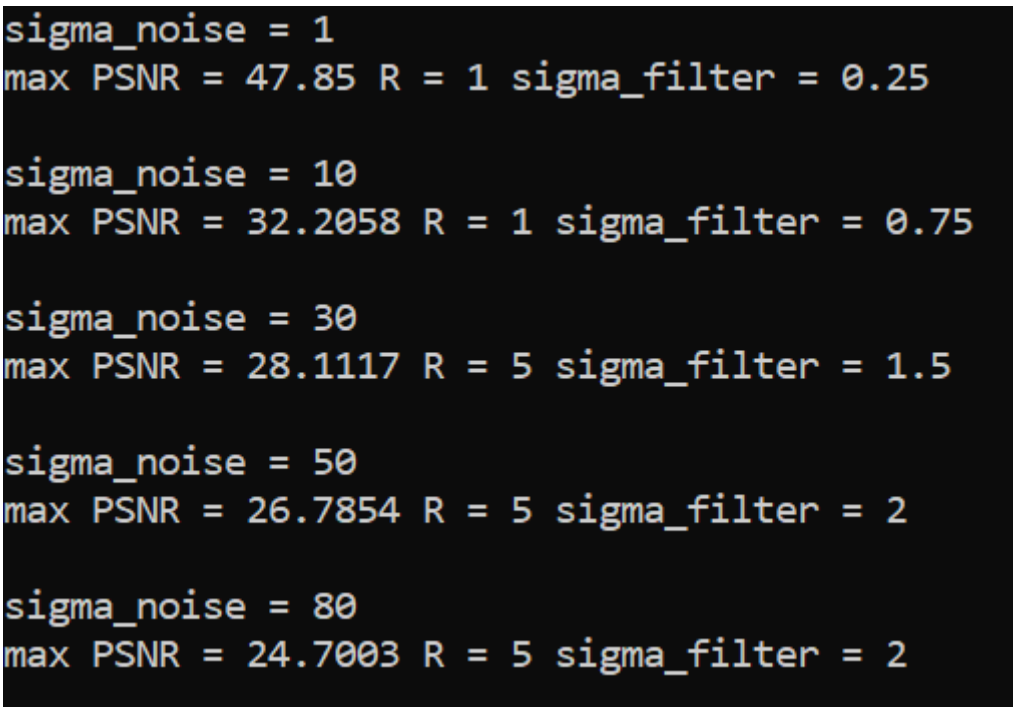


Рисунок 18 - Лучшие значения PSNR при фиксированной $\sigma_{\text{шум}}$

Как видно из полученных значений – чем больше значение шума, тем больше значение σ, при котором PSNR максимально.

2.6 Реализовать метод медианной фильтрации с размерами окон (2R + 1) × (2R + 1), где R определяет радиус фильтра

Алгоритм медианной фильтрации для пикселя $I_{i,j}$ можно представить в виде следующих шагов:

– формирование одномерного массива A из апертуры пикселя $I_{i,j}$;

– взятие медианы одномерного массива A.

Описанный алгоритм применяется для всех пикселей исходного изображения $I[H \times W]$.

Рассмотрим принципы нахождения медианы на примере произвольного одномерного массива $A = a_1, a_2, \dots, a_n$ $a_i \in$ R. Следует выполнить следующие шаги:

– произвести сортировку элементов одномерного массива A (неважно по возрастанию или по убыванию);

– сохранить в $I'_{i,j}$ элемент, находящийся в середине отсортированного массива A. Для апертуры радиуса R необходимо взять из отсортированного массива элемент с индексом $\left\lfloor \frac{(2R+1)^2}{2} \right\rfloor + 1$.

Изображения с аддитивным шумом и после наложения фильтра:

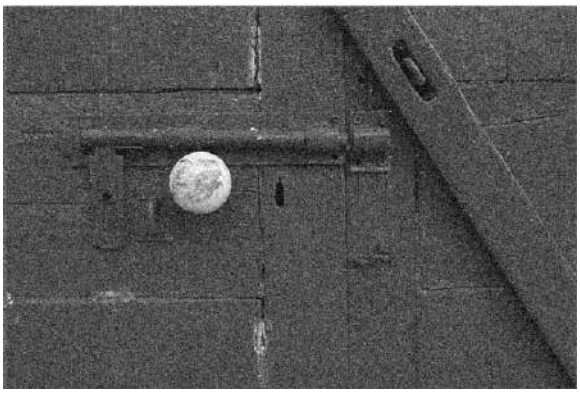


Рисунок 19 - Изображение с аддитивным шумом с параметром σ=30

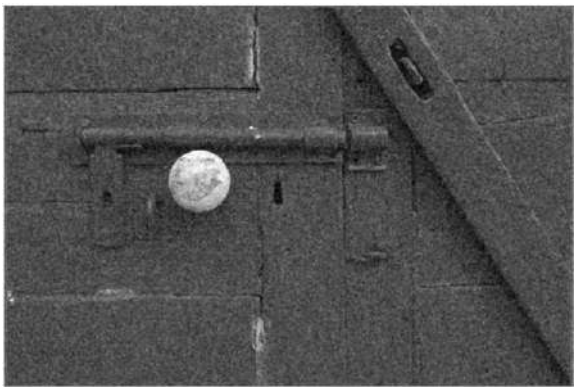


Рисунок 20 - Изображение, полученное после медианной фильтрации с R = 3

Результатом работы фильтра является очень нечеткое изображение, шум удалось сгладить, тем не менее контуры размыты.

2.7 Подобрать размер окна R для метода медианной фильтрации, который бы позволил максимизировать значение PSNR, вычисляемое по оригинальному изображению и зашумленному изображению после фильтрации

Рассмотрим модель шума с параметрами $\sigma = 1, 10, 30, 50, 80$. Для каждой из них будет рассчитано значение PSNR после применения медианной фильтрации с радиусом R = 1, 2, 3, 4. В результате работы программы были получены следующие значения:

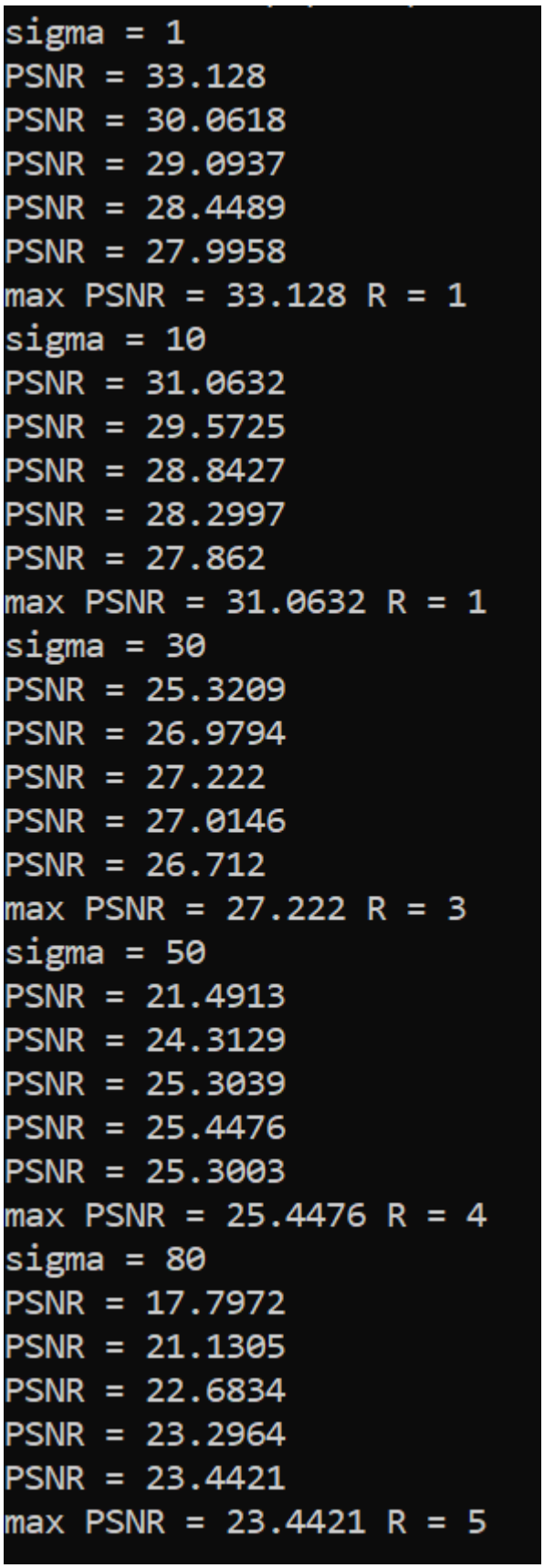


Рисунок 21 - Рассчитанные значения PSNR для различных R и σ

При зашумленности σ < 30 PSNR принимает максимальное значение при R = 1. В случае большой зашумленности, когда σ = 80, наибольшее значение достигается при R = 5. Отсюда

можно сделать вывод, что чем сильнее шум на изображении, тем больший размер окна следует брать.

2.8 + 2.6 Проанализировать значения PSNR, рассчитанные для зашумленных изображений после фильтрации

2.8.1 Параметр шума $\sigma = 1$

Рисунок 22 - Изображение с шумом PSNR = 47,8328

Рисунок 23 - Изображение с применением скользящего среднего PSNR = 31,5972

Рисунок 24 - Изображение с применением фильтра Гаусса PSNR = 47,85

Рисунок 25 - Изображение с применением медианного фильтра PSNR = 33,128

2.8.2 Параметр шума $\sigma = 10$

Рисунок 26 - Изображение с шумом PSNR = 28,1884

Рисунок 27 - Изображение с применением скользящего среднего PSNR = 30,66

Рисунок 28 - Изображение с применением фильтра Гаусса PSNR = 32.205

Рисунок 29 - Изображение с применением медианного фильтра PSNR = 31.063

2.8.3 Параметр шума $\sigma = 30$

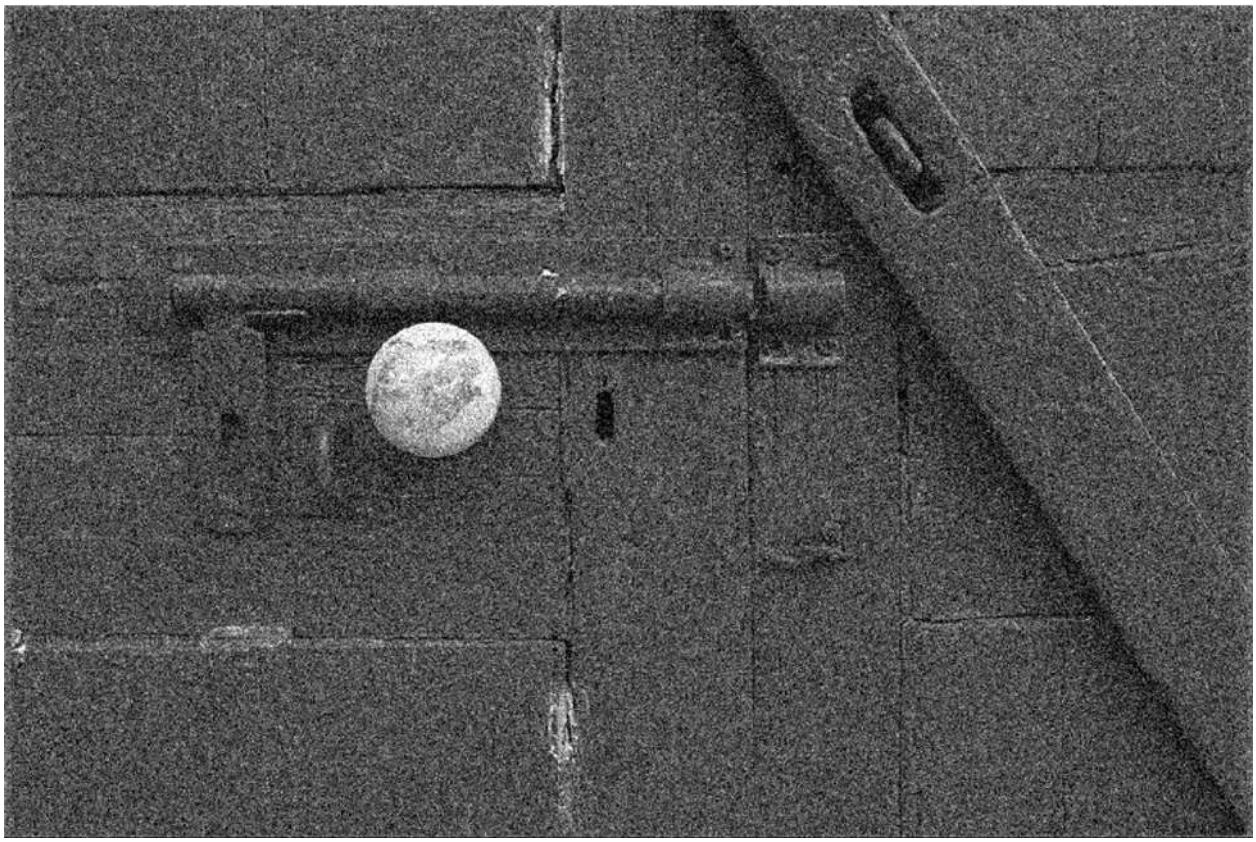


Рисунок 30 - Изображение с шумом PSNR = 18,7307

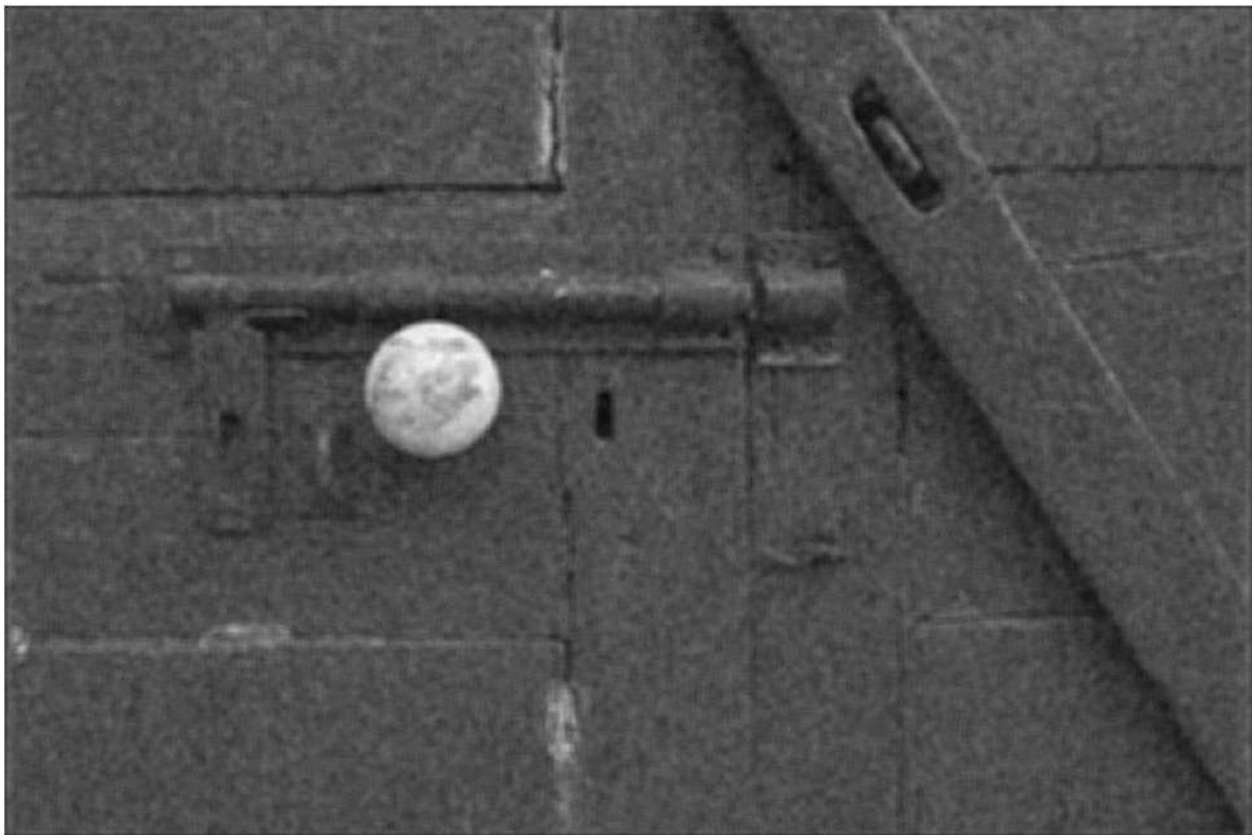


Рисунок 31 - Изображение с применением скользящего среднего PSNR = 27,3701

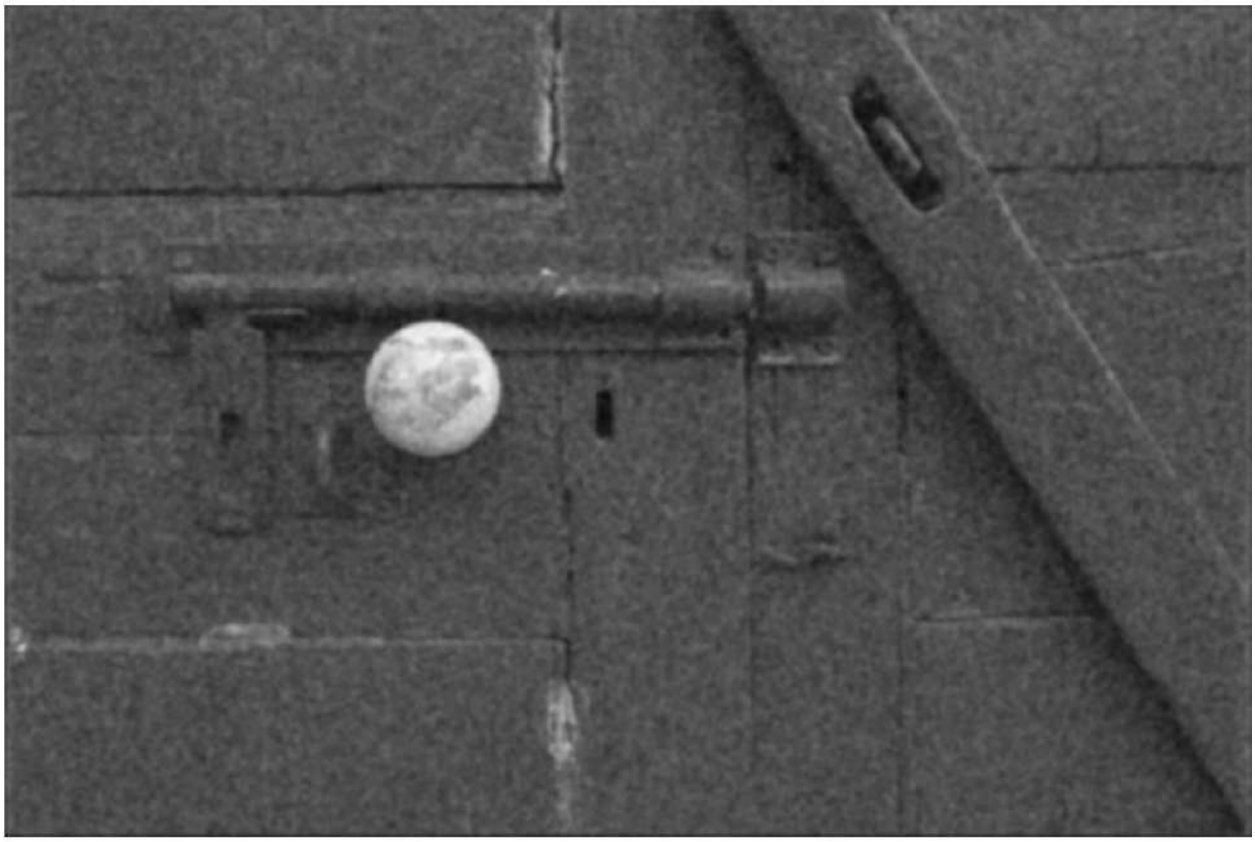


Рисунок 32 - Изображение с применением фильтра Гаусса PSNR = 28,117

Рисунок 33 - Изображение с применением медианного фильтра PSNR = 27,222

2.8.4 Параметр шума $\sigma = 50$

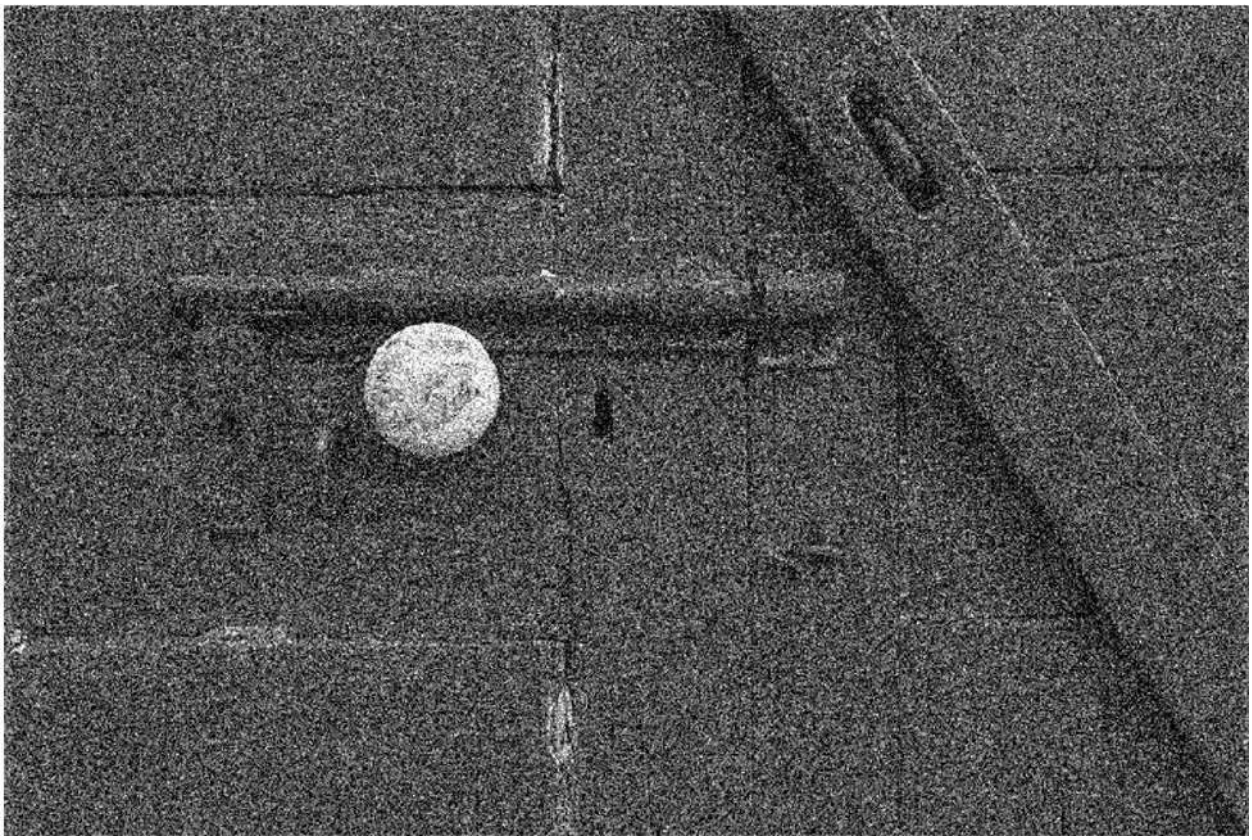


Рисунок 34 - Изображение с шумом PSNR = 14,6995

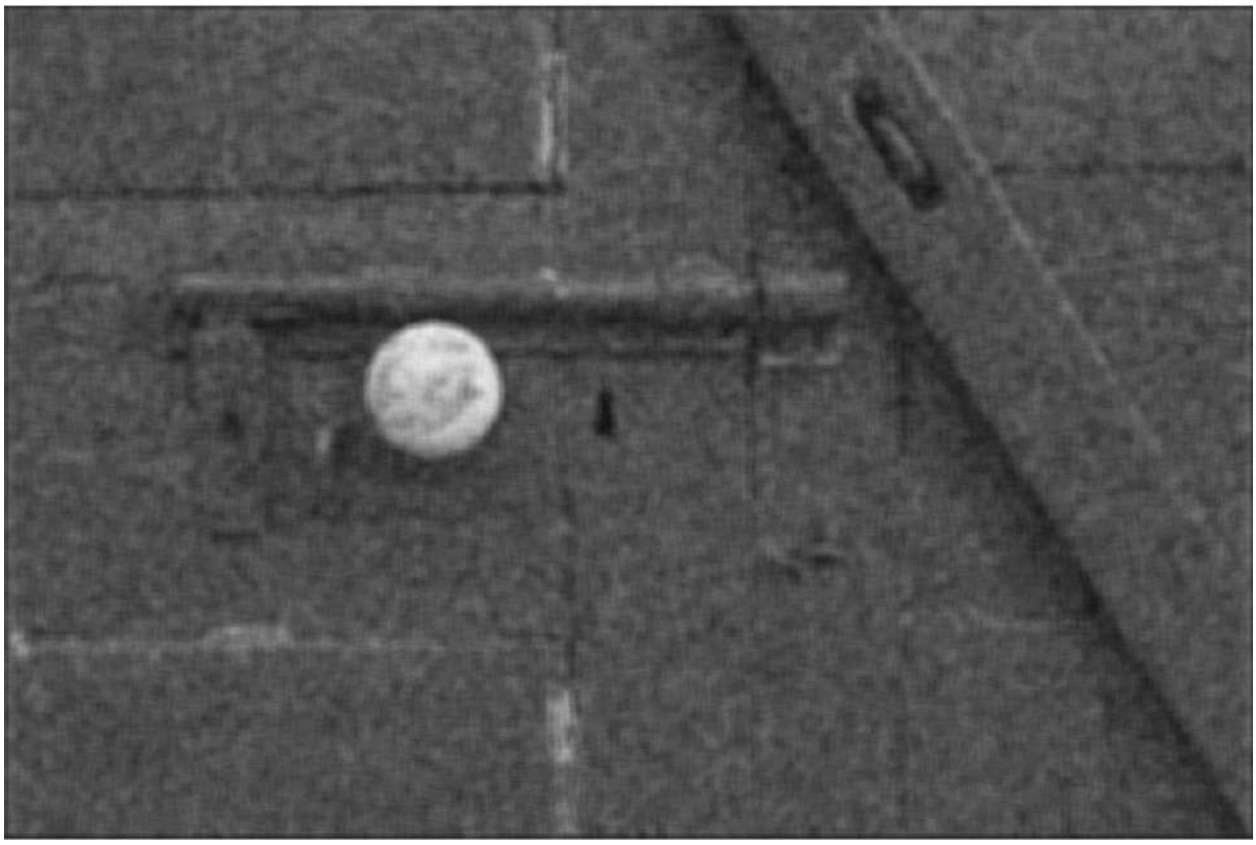


Рисунок 35 - Изображение с применением скользящего среднего PSNR = 26,2824

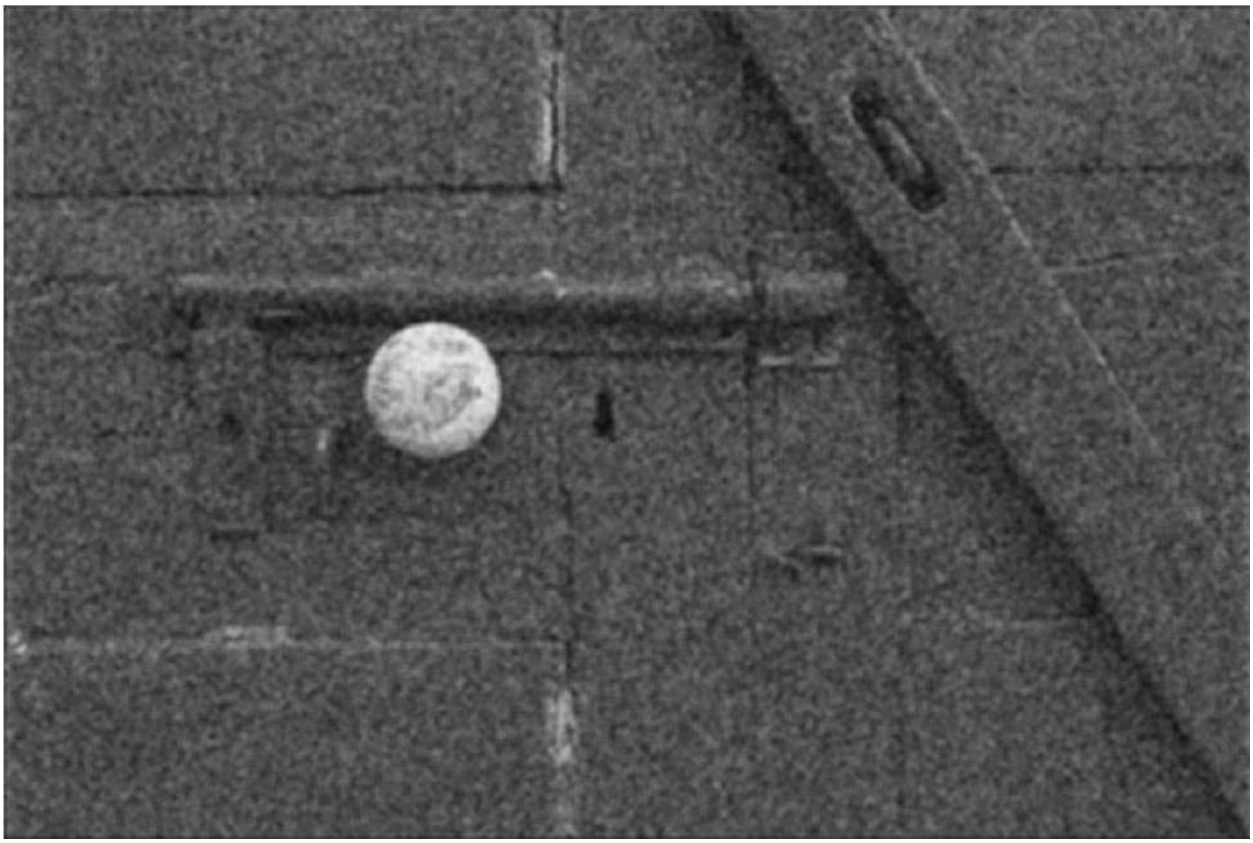


Рисунок 36 - Изображение с применением фильтра Гаусса PSNR = 26,78

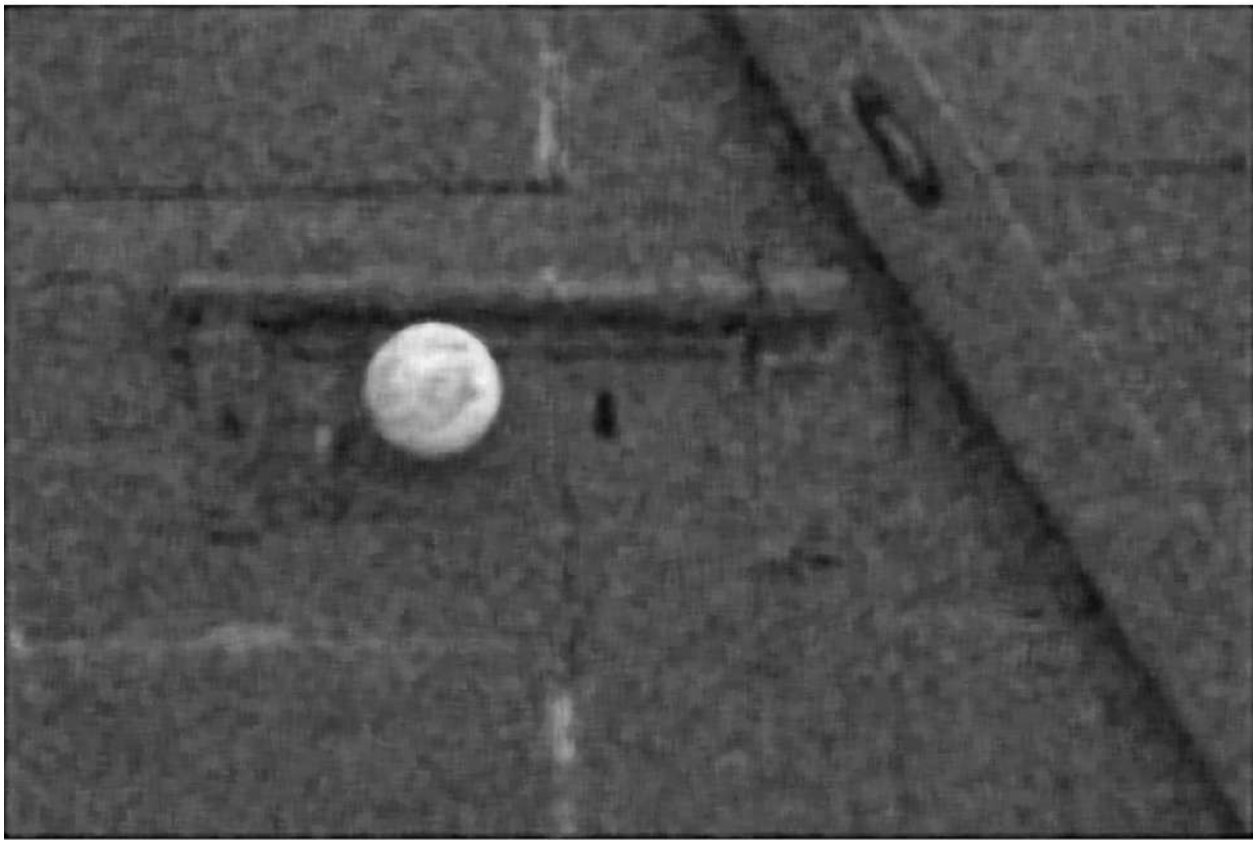


Рисунок 37 - Изображение с применением медианного фильтра PSNR =25,44

2.8.5 Параметр шума $\sigma = 80$

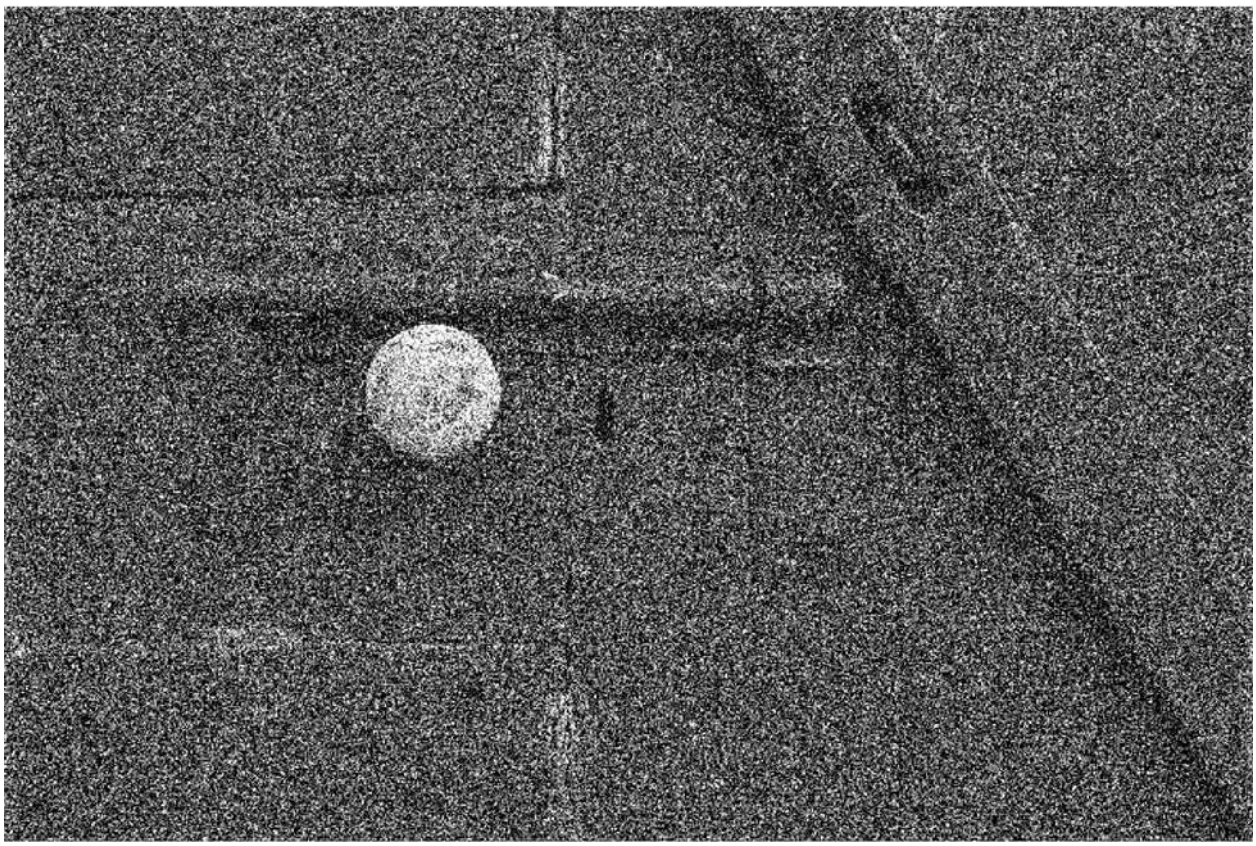


Рисунок 38 - Изображение с шумом PSNR = 11,5537

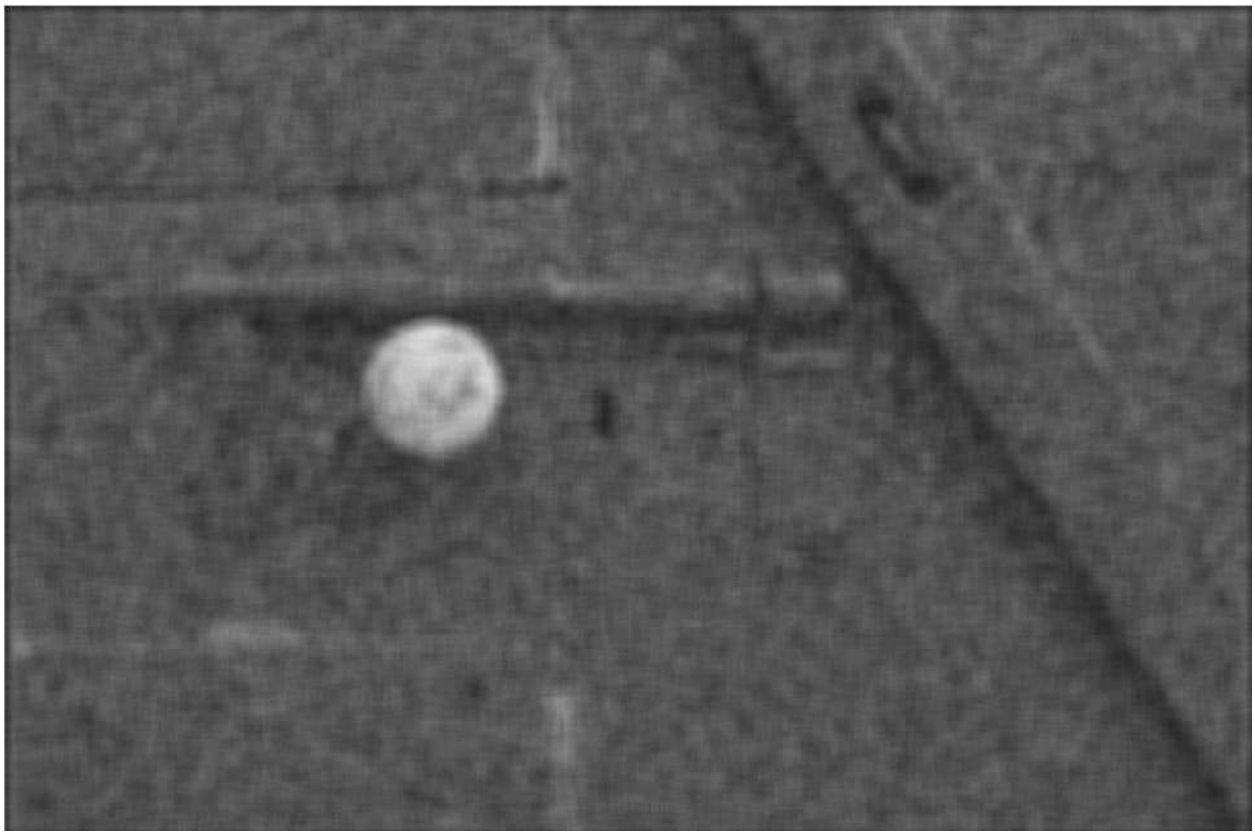


Рисунок 39 - Изображение с применением скользящего среднего PSNR = 24,8347

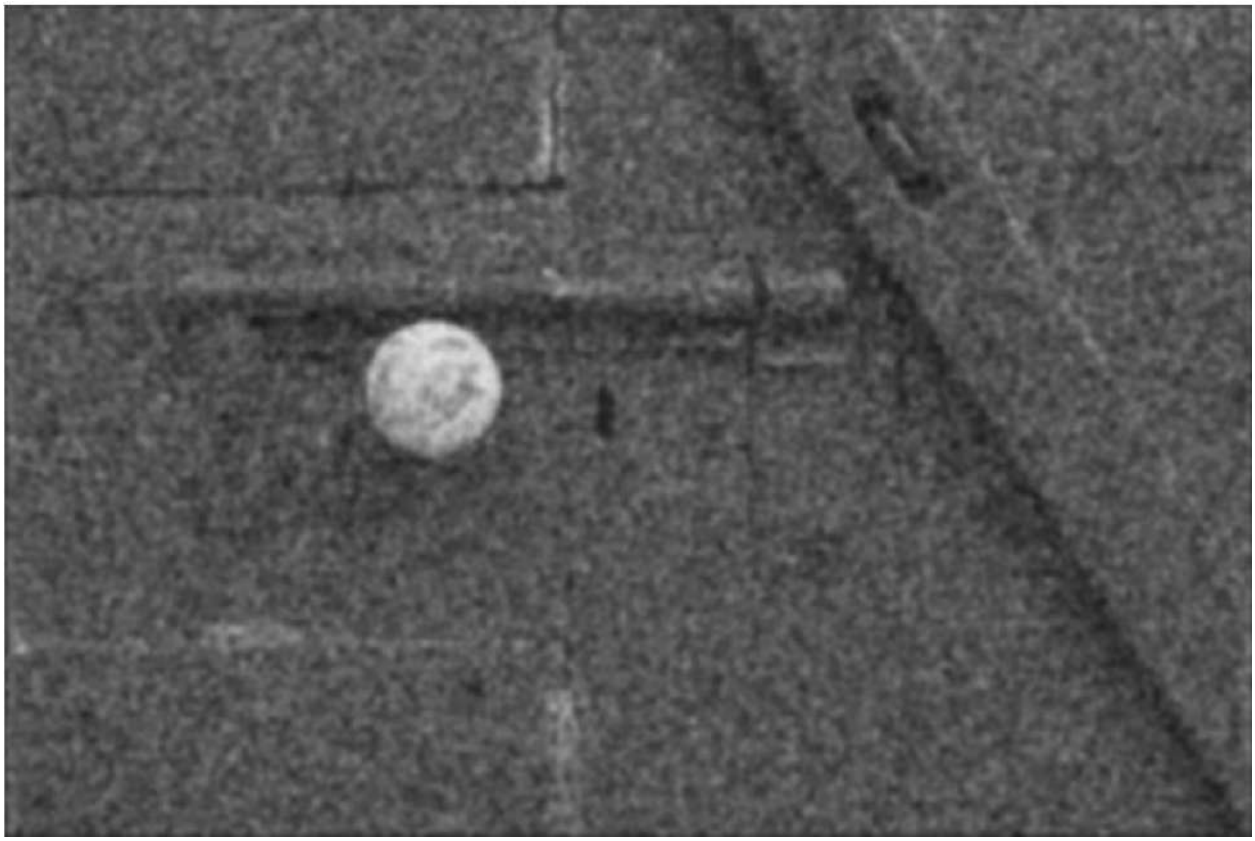


Рисунок 40 - Изображение с применением фильтра Гаусса PSNR = 24,7003

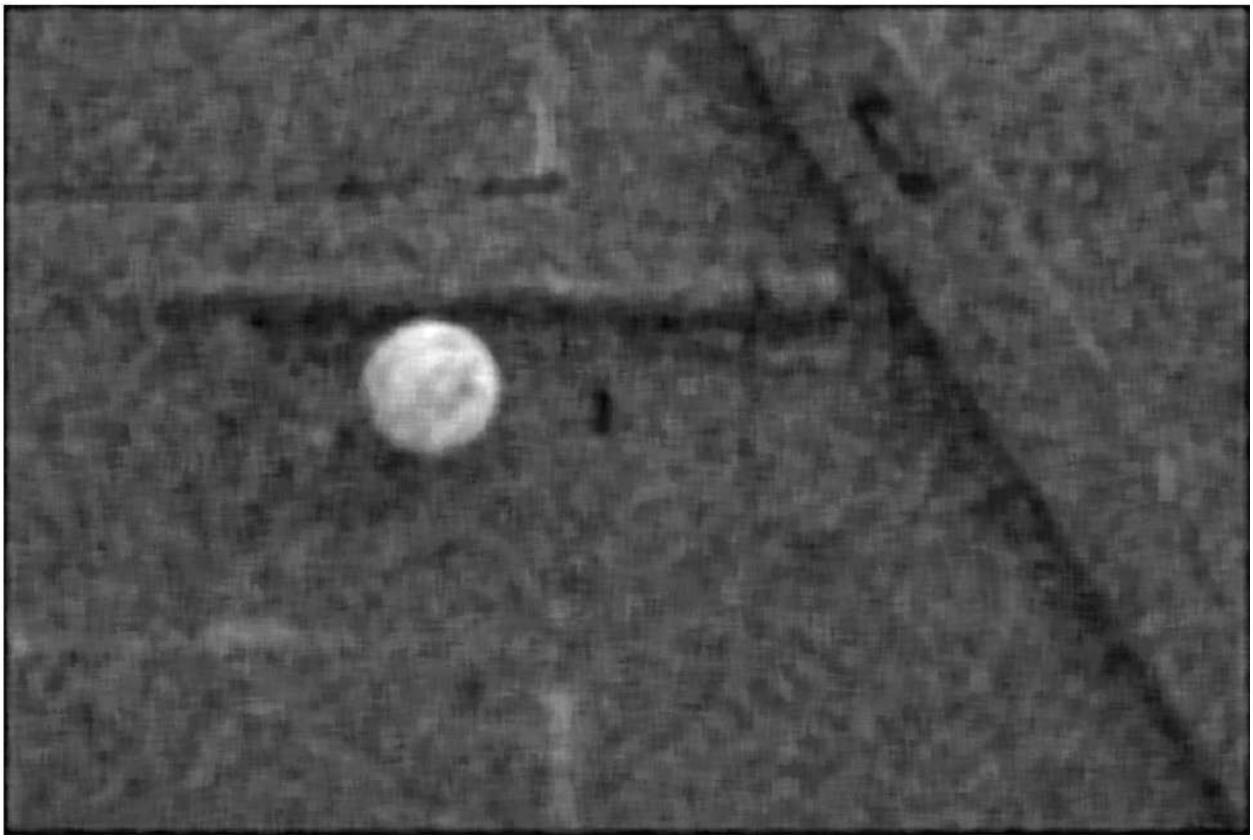


Рисунок 41 - Изображение с применением медианного фильтра PSNR = 23,447

Визуально можно заметить, что метод Гаусса и скользящего среднего эффективнее улучшает изображение. Метод медианной фильтрации даже при слабом шуме заметно ухудшает изображение. Так же у метода медианной фильтрации отсутствует зернистость, как у фильтра Гаусса и скользящего среднего.

2.9 Произвести анализ полученных значений PSNR после применения различных фильтров

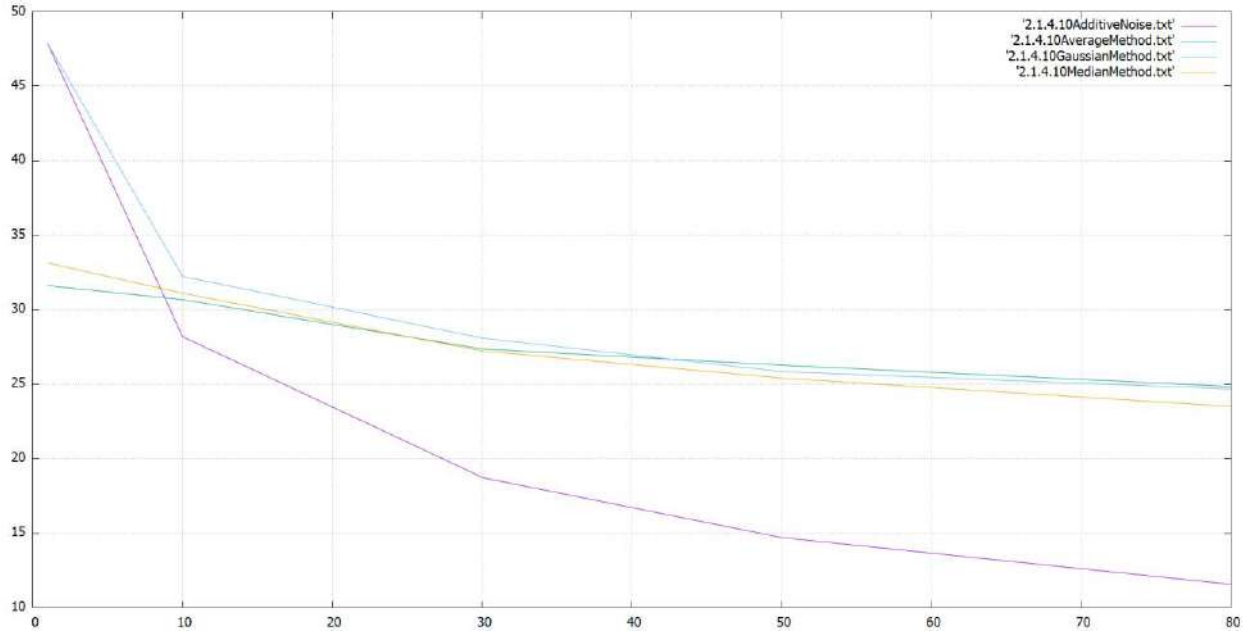


Рисунок 42 - Графики PSNR

По графикам видно, что для слабозашумлённых изображений (σ = 1, σ = 10 лучшей фильтрацией является фильтр Гаусса. Для σ ≈ 30 для данной картинки лучшей фильтрацией является медианный метод, однако визуально для медианного фильтра изображение воспринимается гораздо хуже. Примерно для σ ≈ 80 значения всех фильтров примерно равны.

3 Обработка изображений с импульсным шумом

3.1 Наложить на исходное изображение импульсный шум в соответствии с моделью. Подготовить несколько вариантов зашумленного изображения, для которых доля искаженных пикселей составит 5%, 10%, 25%, 50%.

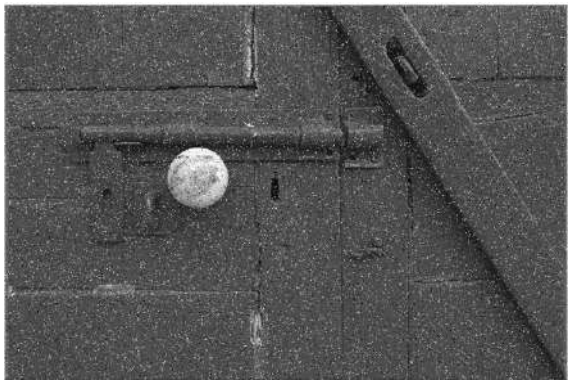


Рисунок 43 - Изображение с импульсным шумом с параметрами $p_a = p_b = 0.025$

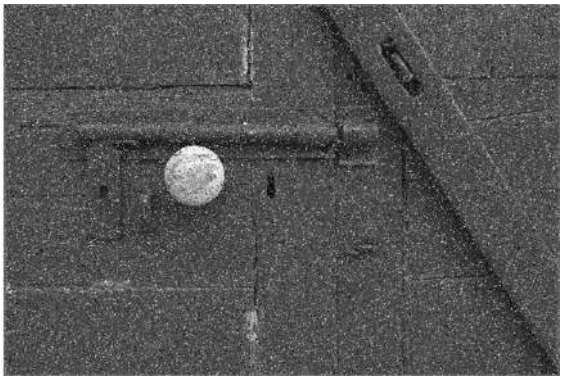


Рисунок 44 - Изображение с импульсным шумом с параметрами $p_a = p_b = 0.05$

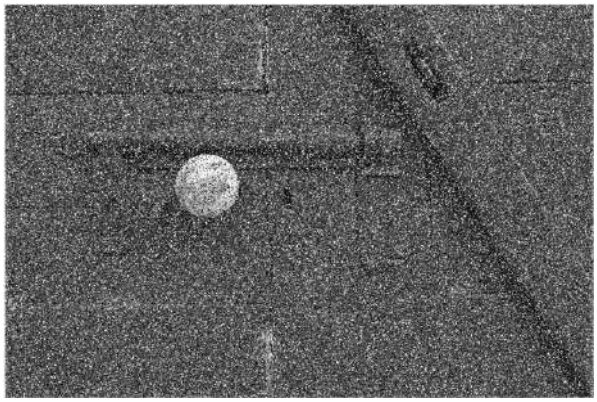


Рисунок 45 - Изображение с импульсным шумом с параметрами $p_a = p_b = 0.125$

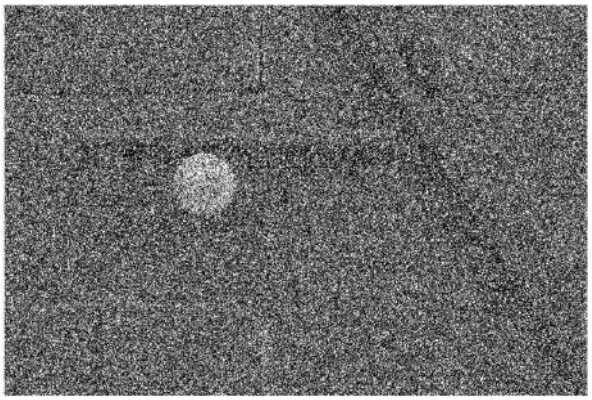


Рисунок 46 - Изображение с импульсным шумом с параметрами $p_a = p_b = 0.25$

3.2 Вычислить значения PSNR по исходному и зашумленным изображениям

Значения PSNR имеют вид:

```
PSNR = 17.5657
PSNR = 15.1988
PSNR = 11.2022
PSNR = 8.31733
```

Рисунок 47 - PSNR импульсного шума

При зашумленности в 50% значение PSNR уменьшается в 2 раза по сравнению с зашумленностью в 5%.

3.3 Реализовать метод медианной фильтрации с размерами окон (2R + 1) × (2R + 1), где R определяет радиус фильтра

Изображения с применением медианного фильтра имеют вид:

Рисунок 48 - Изображение с применением медианного фильтра R =1 для импульсного изображения с параметрами $p_a = p_b = 0.025$

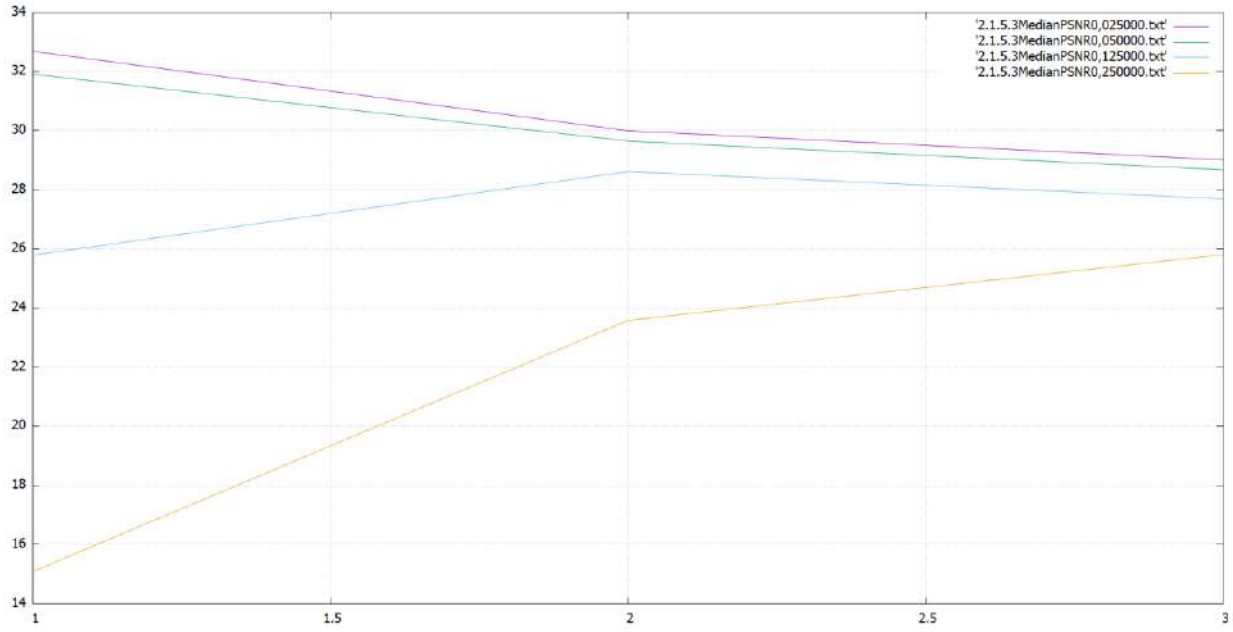
Рисунок 49 - Изображение с применением медианного фильтра R =2 для импульсного изображения с параметрами $p_a = p_b = 0.05$

Рисунок 50 - Изображение с применением медианного фильтра R = 2 для импульсного изображения с параметрами $p_a = p_b = 0.125$

Рисунок 51 - Изображение с применением медианного фильтра R = 3 для импульсного изображения с параметрами $p_a = p_b = 0.25$

По полученным изображением можно сделать вывод, что изображения получились смазанными, медианная фильтрация позволила полностью избавиться от шума. При большом шуме изображение стало визуально восприимчивым по сравнению с зашумленным. Это связано с тем, что данный метод игнорирует крайние значения интенсивности, которые и принимает пиксель при импульсном шуме.

3.4 Построить зависимость PSNR(R) для всех вариантов зашумленных изображений. Провести визуальное сравнение. Указать искажения, специфические для рассмотренных методов. Выбрать наилучший параметр фильтра для каждого варианта зашумленного изображения



По результатам графика видно, что при низкой зашумленности (5% и 10%) оптимальными значениями для достижения максимального PSNR являются R = 1, в случае с зашумленностью в 25% оптимальным является R = 2, при шуме в 50% наилучший результат дает R = 3.

4 Реализация методов выделения контуров

4.1 Применение оператора Лапласа и формирование изображения по отклику

Рисунок 52 – Отклик после первой маски

Рисунок 53 - Отклик после второй маски

В результате были сформированы изображения по отклику:

Рисунок 54 - Изображение по отклику после первой маски

Рисунок 55 - Изображения по отклику после второй маски

По полученным изображениям можно увидеть разницу в применении масок: например, после использования первой маски пазы на шляпках саморезов белого цвета, а после использования второй — пазы на шляпках саморезов черного цвета.

4.2 Синтез изображения с усилением высоких частот

Необходимо синтезировать изображение с усилением высоких частот, используя разность исходного изображения I и отклика оператора Лапласа I', применив маску из следующего рисунка:

0	-1	0
-1	α+4	-1
0	-1	0

Рисунок 56 Маска фильтра Лапласа

Значение пикселя на позиции *(x, y)* синтезированного изображения вычисляется по следующей формуле:

$$Clip\left(I'_{x,y} + I_{x,y}, 0{,}255\right)$$

В результаты было синтезировано изображение с усилением высоких частот:

Рисунок 57 - Синтезированное изображение с усилением высоких частот

4.3 Необходимо синтезировать изображения, применив маску, для разных значений параметра α

Параметр α изменяется в интервале от 1 до 1.5 с шагом 0.1. Результаты синтеза:

Рисунок 58 - Синтезированное изображение при α = 1, 0

Рисунок 59 - Синтезированное изображение при α = 1, 1

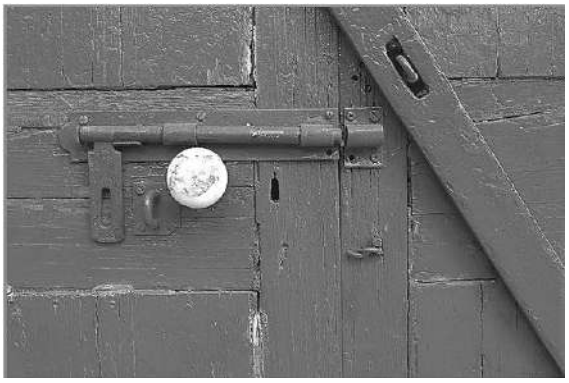


Рисунок 60 - Синтезированное изображение при α = 1, 2

Рисунок 61 - Синтезированное изображение при α = 1, 3

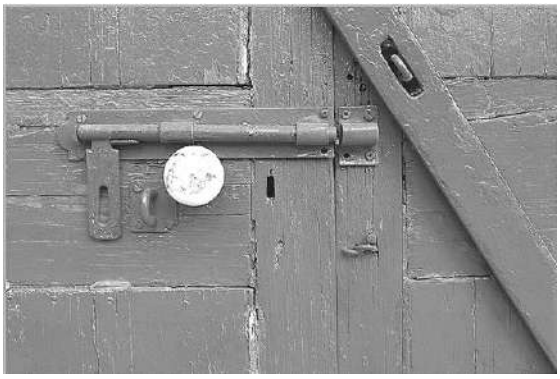


Рисунок 62 - Синтезированное изображение при α = 1, 4

Рисунок 63 - Синтезированное изображение при α = 1, 5

Синтезированное изображение совпадает с I'' при α = 1, так как это соответствует маске Лапласа при α =1, именно при этом значении достигается максимальный эффект увеличения резкости. При увеличении значения растет яркость изображения.

$$I''_{x,y} = I'_{x,y} + I_{x,y} = \left(4I_{x,y} - I_{x-1,y} - I_{x+1,y} - I_{x,y-1} - I_{x,y+1}\right) + I_{x,y} = 5I_{x,y} - I_{x-1,y} - I_{x+1,y}$$

$$-I_{x,y-1} - I_{x,y+1}$$

4.4 Средние значения яркости

Необходимо рассчитать среднее значение яркости для каждого из изображений, полученных в предыдущем пункте:

```
alpha = 1 Средняя яркость = 79.6938
alpha = 1.1 Средняя яркость = 95.2095
alpha = 1.2 Средняя яркость = 110.588
alpha = 1.3 Средняя яркость = 125.774
alpha = 1.4 Средняя яркость = 140.815
alpha = 1.5 Средняя яркость = 155.671
```

Рисунок 64 - Средняя яркость

4.5 Построение и анализ гистограмм

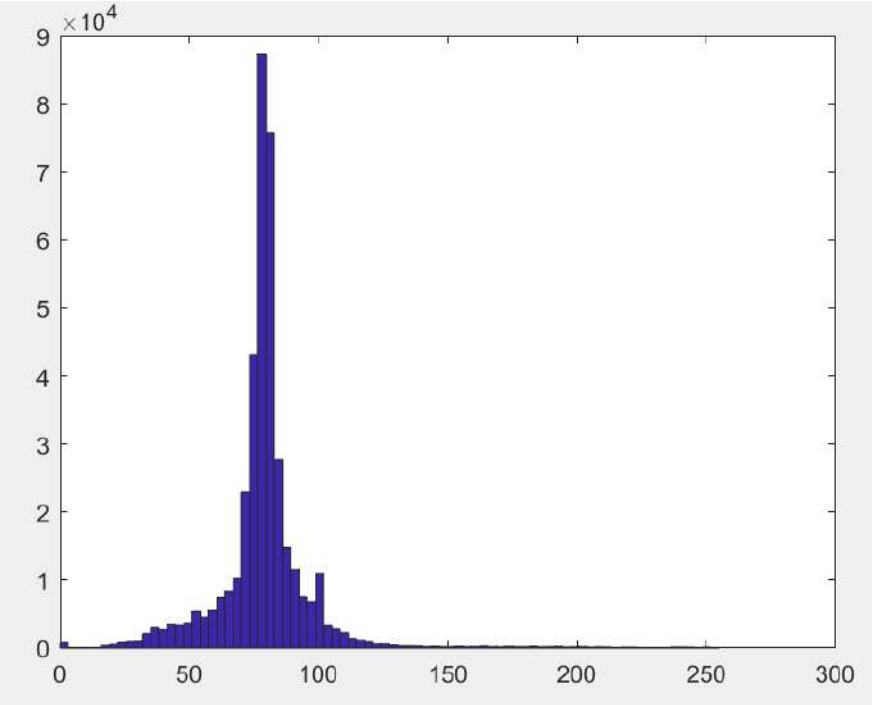


Рисунок 65 - Гистограмма исходного изображения

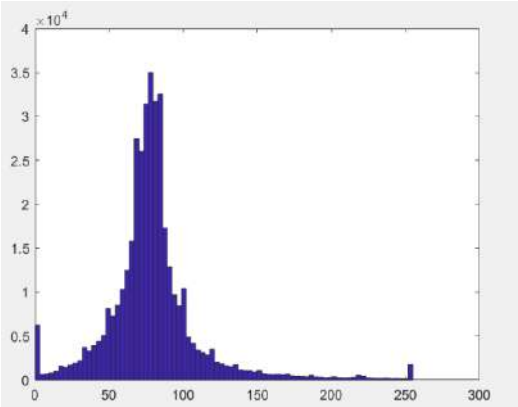


Рисунок 66 - Гистограмма изображения для α = 1.0

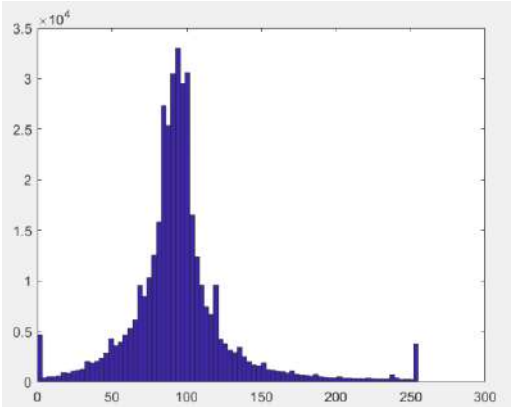


Рисунок 67 - Гистограмма изображения для α = 1.1

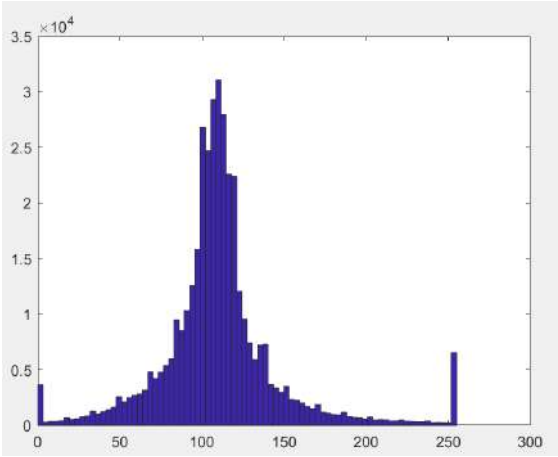


Рисунок 68 - Гистограмма изображения для α = 1.2

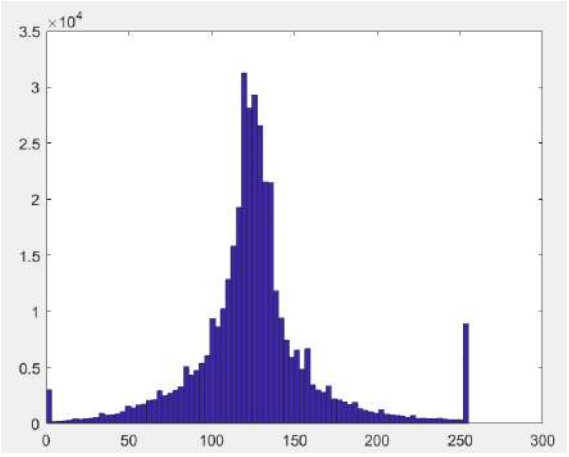


Рисунок 69 - Гистограмма изображения для α = 1.3

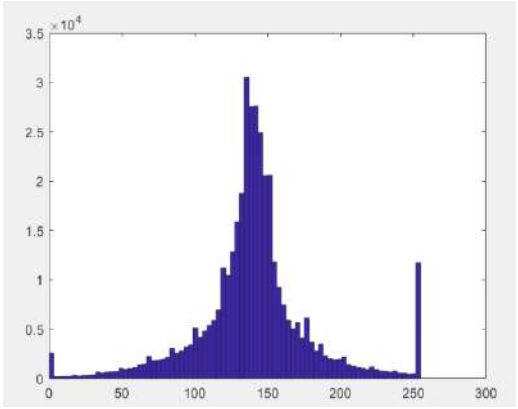


Рисунок 70 - Гистограмма изображения для α = 1.4

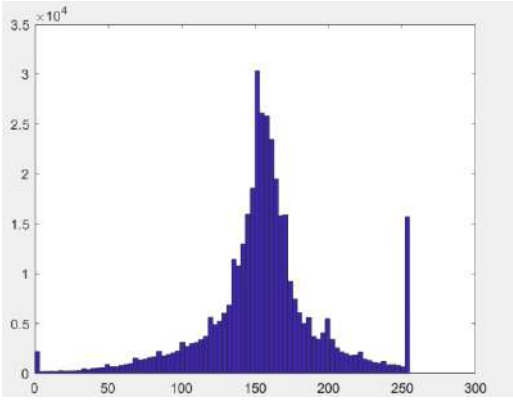


Рисунок 71 - Гистограмма изображения для α = 1.5

По полученным гистограммам отчетливо можно видеть, что, при увеличении α, уменьшается общее количество пикселей с интенсивностью компоненты < 255 и

увеличивается крайний правый пик – количество пикселей с интенсивностью компоненты равной 255. Это связано с постепенным осветлением изображения – все больше компонент пикселей принимают свое максимальное значение.

4.6 Применение оператора Собеля

Необходимо применить оператор Собеля к исходному изображению I.

Оператор Собеля является ключевым во многих алгоритмах анализа контуров изображения. В основе оператора Собеля лежит расчет двух производных: по вертикали и по горизонтали. Соответствующие этим операциям маски фильтров:

а)

-1	0	1
-2	0	2
-1	0	1

б)

1	2	1
0	0	0
-1	-2	-1

а) – по горизонтали, б) – по вертикали

Алгоритм применения оператора:

- Рассчитать значения откликов $G_{x,y}^h$ и $G_{x,y}^v$, используя фильтры с масками
- Рассчитать величину силы (длины) контура $|\nabla I_{x,y}|$ по формуле

$$\left|\nabla I_{x,y}\right| = \sqrt{{G_{x,y}^h}^2 + {G_{x,y}^v}^2}$$

- Рассчитать направление градиента $\theta_{x,y}$ по формуле

$$\theta_{x,y} = arctg\frac{G_{x,y}^v}{G_{x,y}^h}$$

Результат:

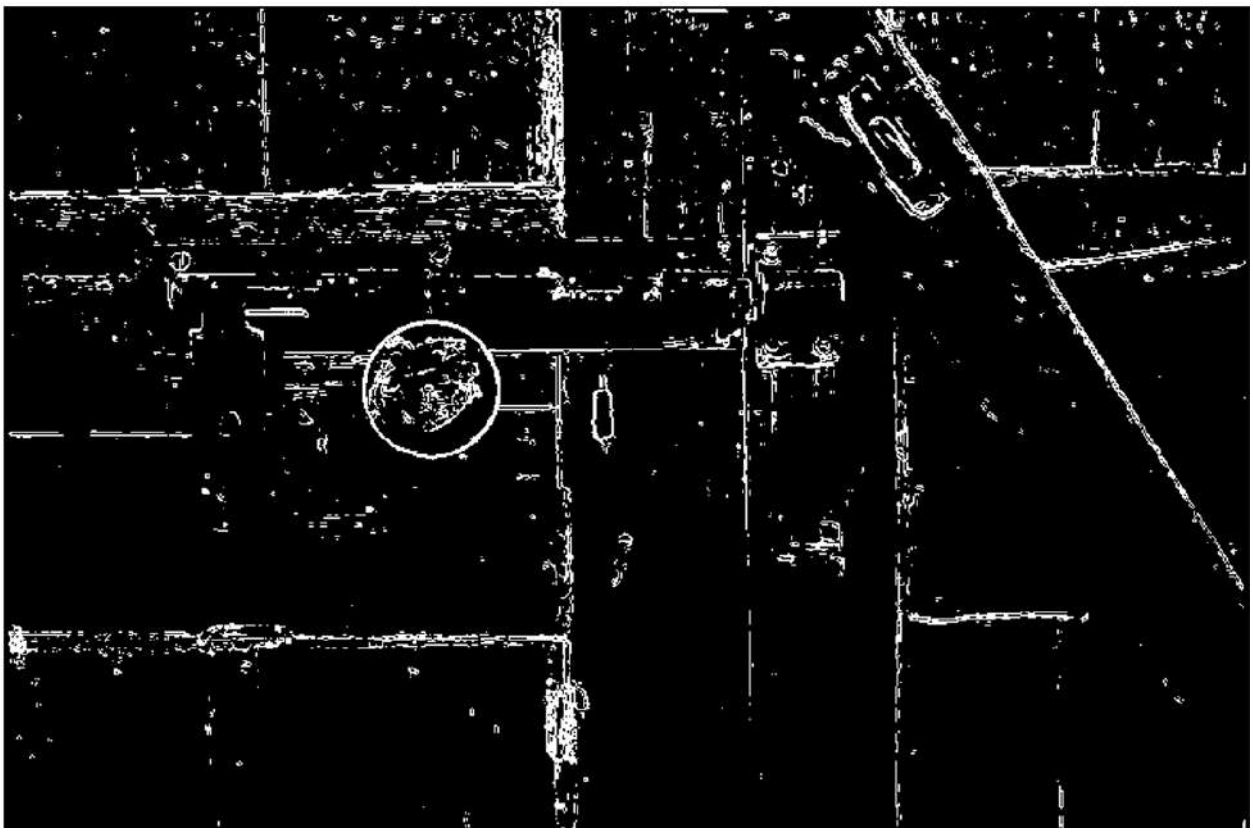


Рисунок 72 - Изображение в результате применения оператора Собеля

4.7 Формирование изображений для рассмотренных порогов thr и выбор наилучшего.

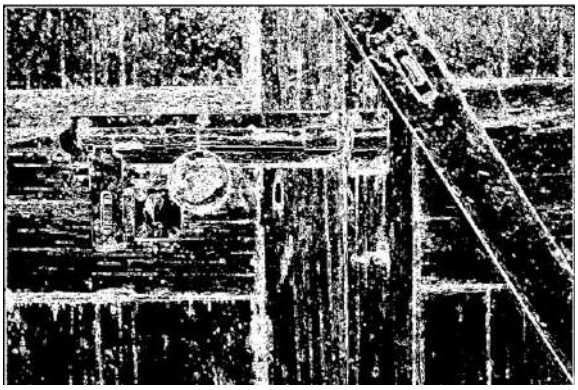


Рисунок 73 - Результат применения оператора Собеля для thr = 30

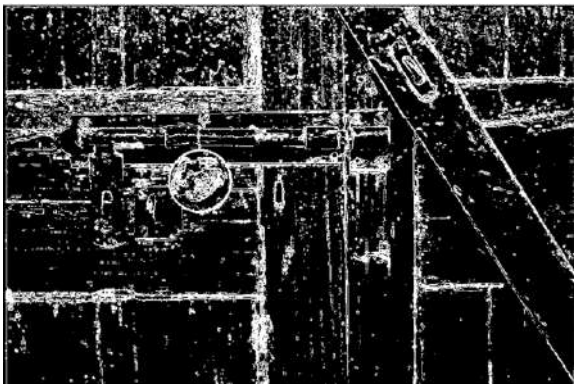


Рисунок 74 - Результат применения оператора Собеля для thr = 60

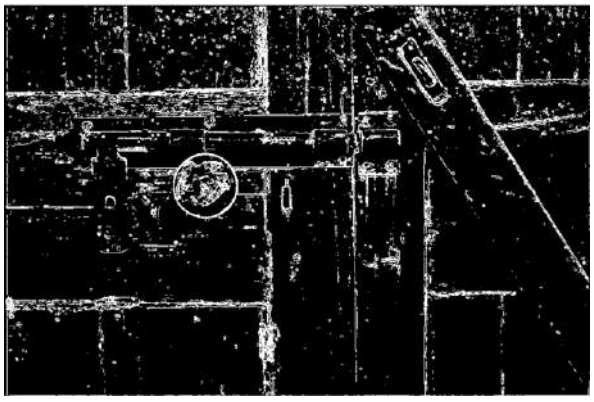


Рисунок 75 - Результат применения оператора Собеля для thr = 90

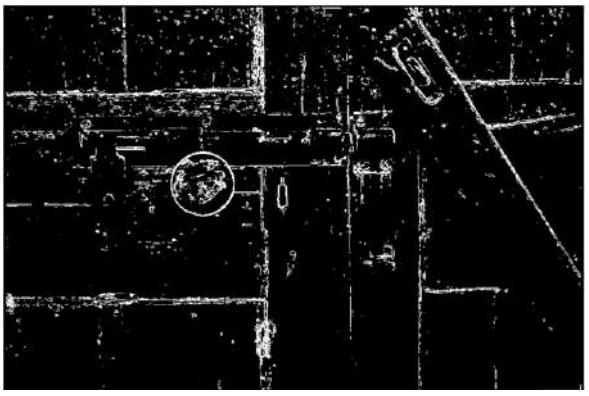


Рисунок 76 - Результат применения оператора Собеля для thr = 120

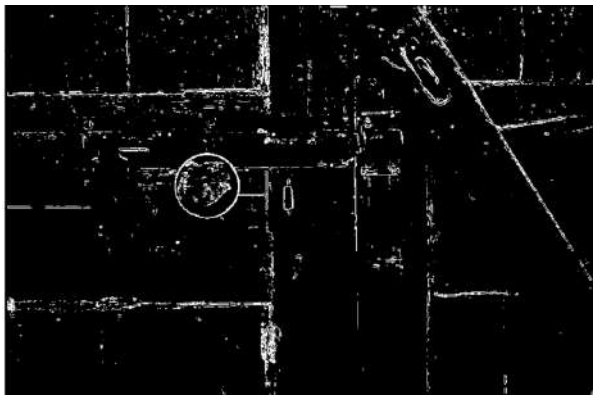


Рисунок 77 - Результат применения оператора Собеля для thr = 150

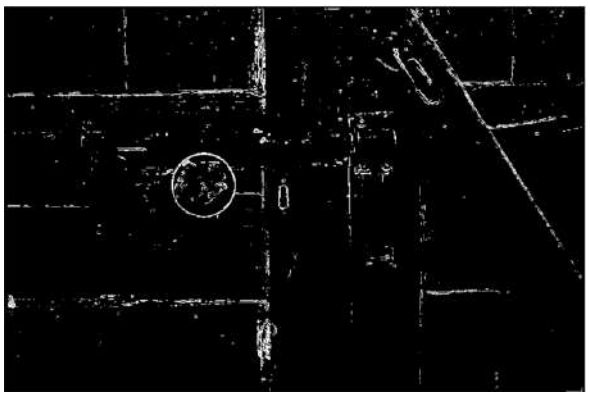


Рисунок 78 - Результат применения оператора Собеля для thr = 180

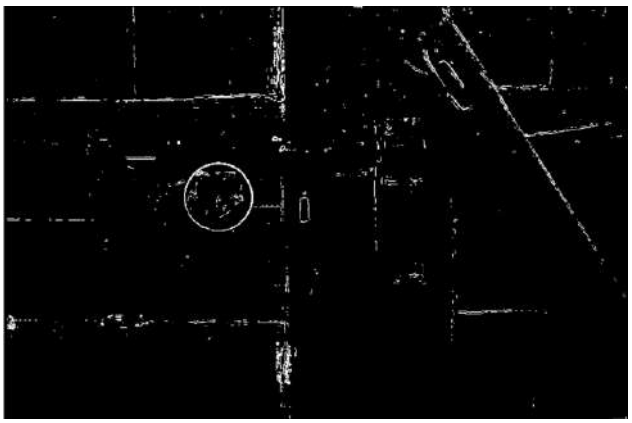


Рисунок 79 - Результат применения оператора Собеля для thr = 210

По сформированным изображениям можно сделать вывод, что наиболее точной бинарной картой является карта с порогом thr = 120, или же при thr=90. В случае, когда thr = 30 изображения в некоторых местах перегружены белыми пикселями. При порогах 150 и 180 некоторые контуры становятся чётче различимы, а некоторые исчезают вовсе.

4.8 Карта направлений градиентов

Необходимо построить 4-цветную карту направлений градиентов на полученных изображениях по такому правилу:

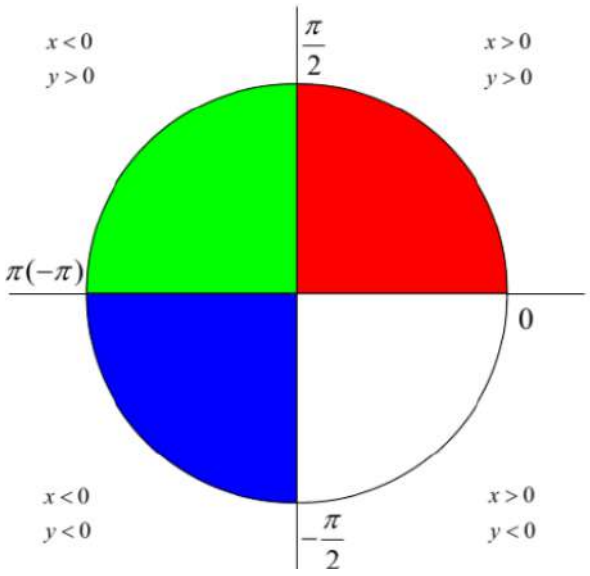


Рисунок 80 Правило соответствия цветов с квадрантами расположения

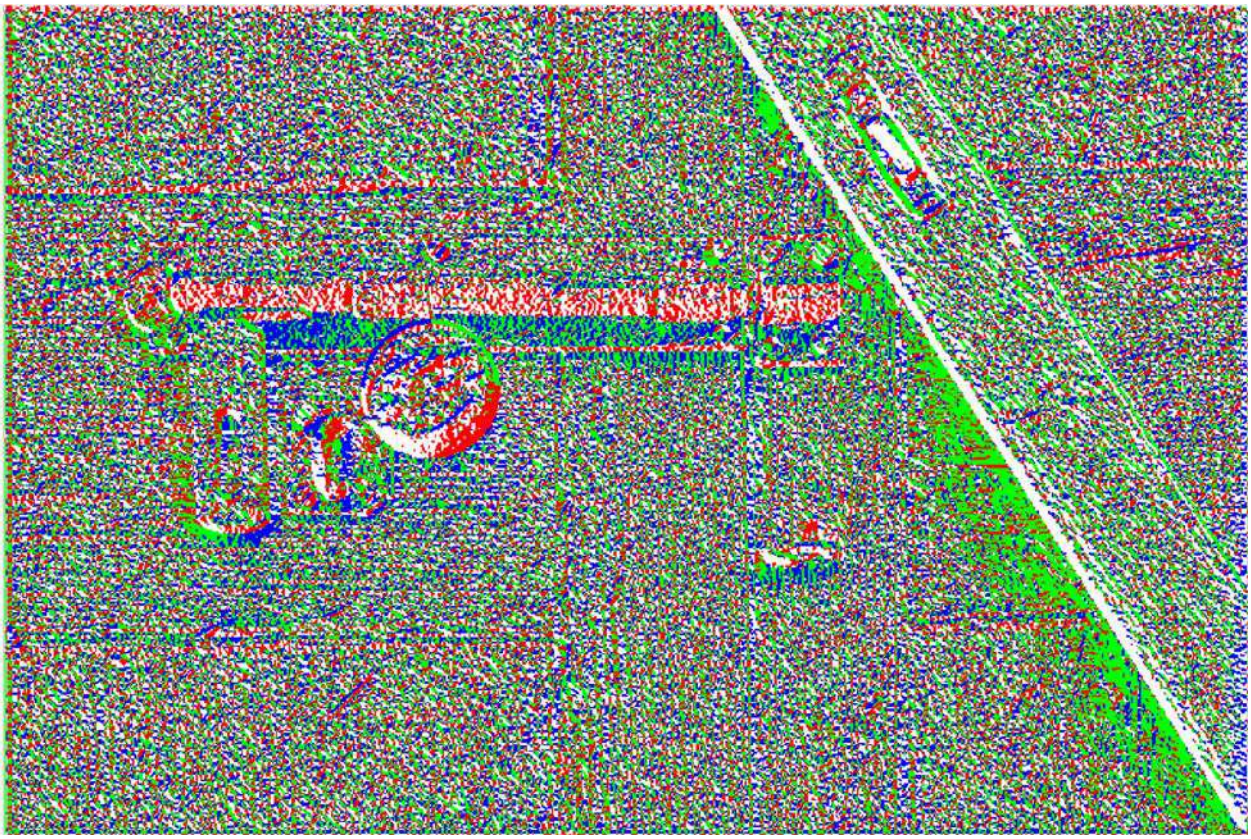


Рисунок 81 - Карта направлений градиентов

5 Градационные преобразования на базе опорных точек

Синтезировать засвеченное, затемненное, а также сбалансированное изображения и применить к ним градационное преобразование на базе двух опорных точек для повышения качества исходных изображений:

Рисунок 82 - Сбалансированное изображение

Рисунок 83 - Преобразование на основе двух опорных точек (65,40) и (180,200) сбалансированного изображения

Рисунок 84 - Затемнённое изображение

Рисунок 85 - Преобразование на основе двух опорных точек (20,70) и (140,220) затемненного изображения

Рисунок 86 - Засвеченное изображение

Рисунок 87 - Преобразование на основе двух опорных точек (120,20) и (210,180) засвеченного изображения

5.1 Формирование гистограммы для исходных и полученных изображений

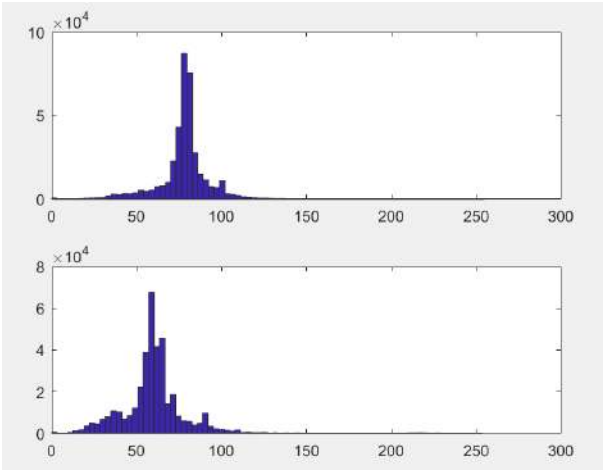


Рисунок 88 - Гистограмма для сбалансированного изображения до и после преобразования

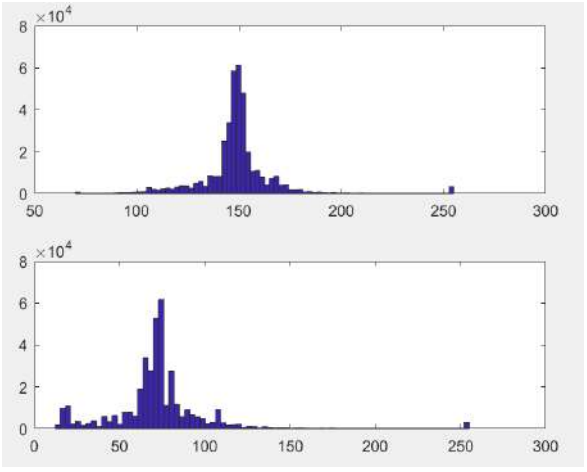


Рисунок 89 - Гистограмма для засвеченного изображения до и после преобразования

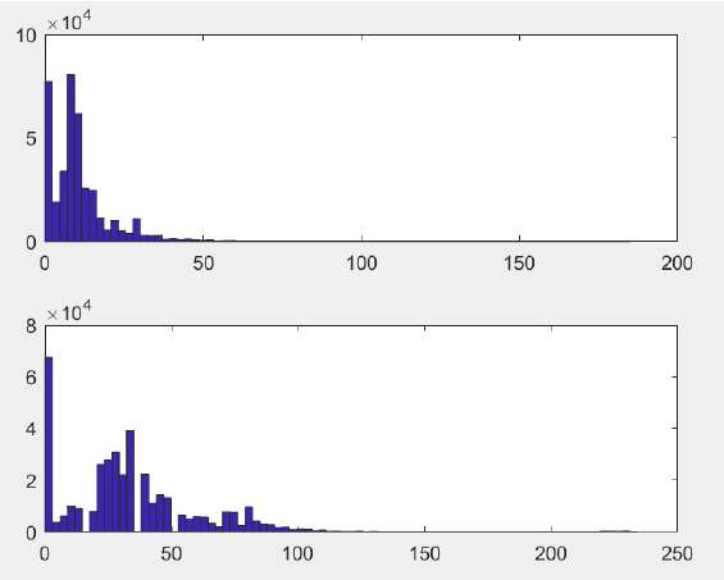


Рисунок 90 - Гистограмма для затемненного изображения до и после преобразования

По полученным гистограммам можно увидеть, что преобразование не влияет на количество пикселей с крайними значениями компонент относительно других пикселей, но поднимает количество отдаленных от крайних пикселей.

5.2 Гамма преобразование

Рисунок 91 - Сбалансированное изображение

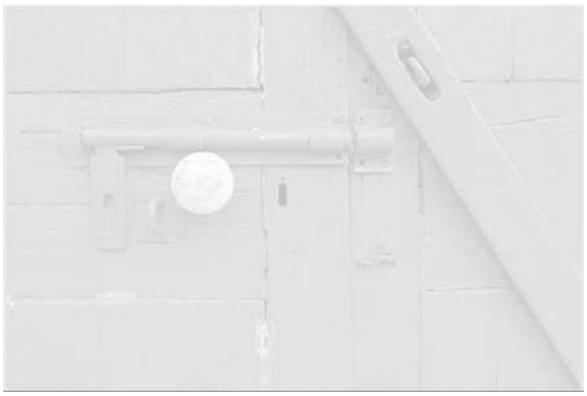


Рисунок 92 - Преобразованное сбалансированного изображения при γ = 0, 1

Рисунок 93 - Преобразованное сбалансированного изображения при γ = 0, 5

Рисунок 94 - Преобразованное сбалансированного изображения при γ = 1

Рисунок 95 - Преобразованное сбалансированного изображения при γ = 2

Рисунок 96 - Преобразованное сбалансированного изображения при γ = 8

Рисунок 97 - Исходное засвеченное изображение

Рисунок 98 - Преобразованное засвеченного изображения при γ = 0.1

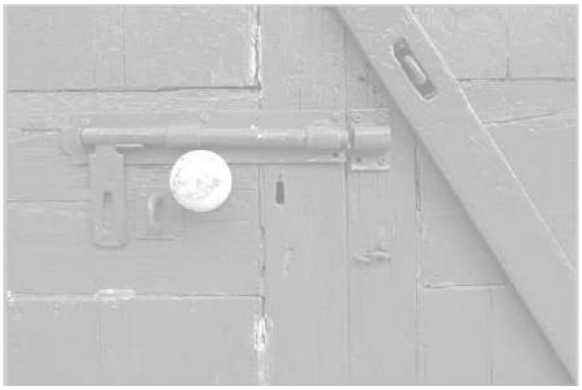


Рисунок 99 - Преобразованное засвеченного изображения при γ = 0.5

Рисунок 100 - Преобразованное засвеченного изображения при γ = 1

Рисунок 101 - Преобразованное засвеченного изображения при γ = 2

Рисунок 102 - Преобразованное засвеченного изображения при γ = 8

Рисунок 103 - Исходное затемненное изображение

Рисунок 104 - Преобразованное засвеченного изображения при γ = 0.1

Рисунок 105 - Преобразованное засвеченного изображения при γ = 0.5

Рисунок 106 - Преобразованное засвеченного изображения при γ = 1

Рисунок 107 - Преобразованное засвеченного изображения при γ = 2

Рисунок 108 - Преобразованное засвеченного изображения при γ = 8

По полученным результатам можно сделать вывод, что при значениях $\gamma < 1$ яркость изображения увеличивается, но при $\gamma > 1$ уже наоборот уменьшается. Таким образом, для улучшения качества засвеченного изображения нужно брать значения > 1, а для затемненного.

5.3 Гистограммы исходных и полученных после гамма преобразования изображений

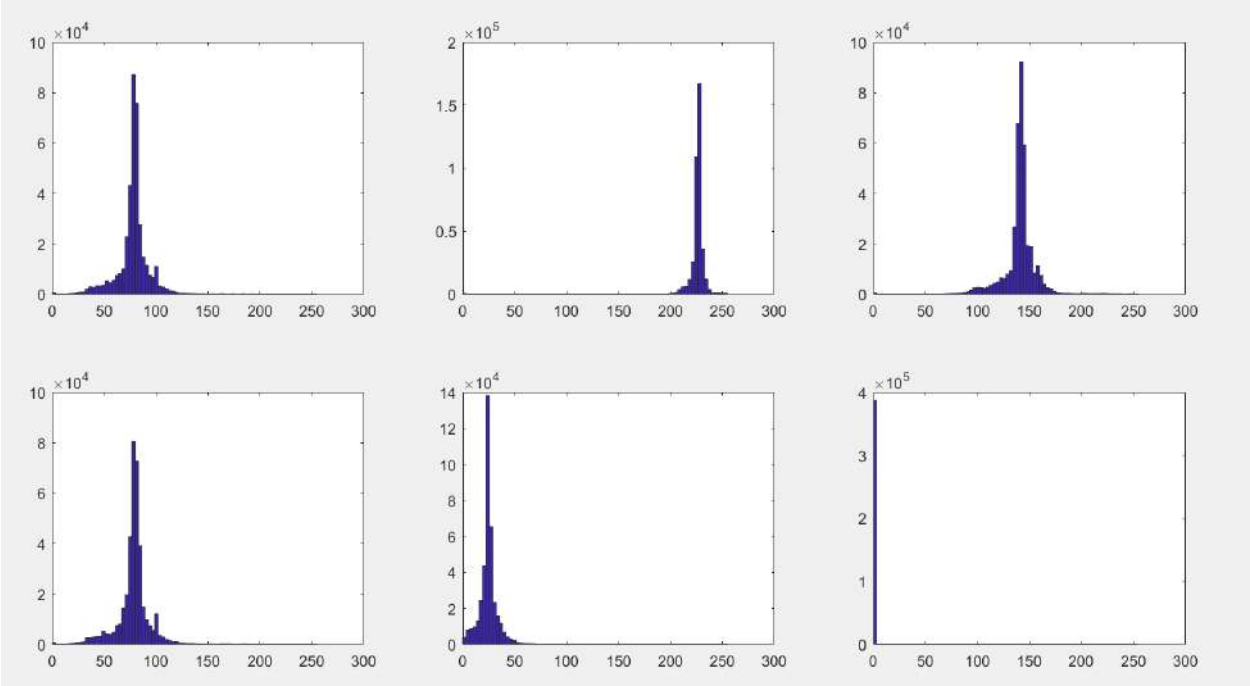


Рисунок 109 - Гистограмма для сбалансированного изображения

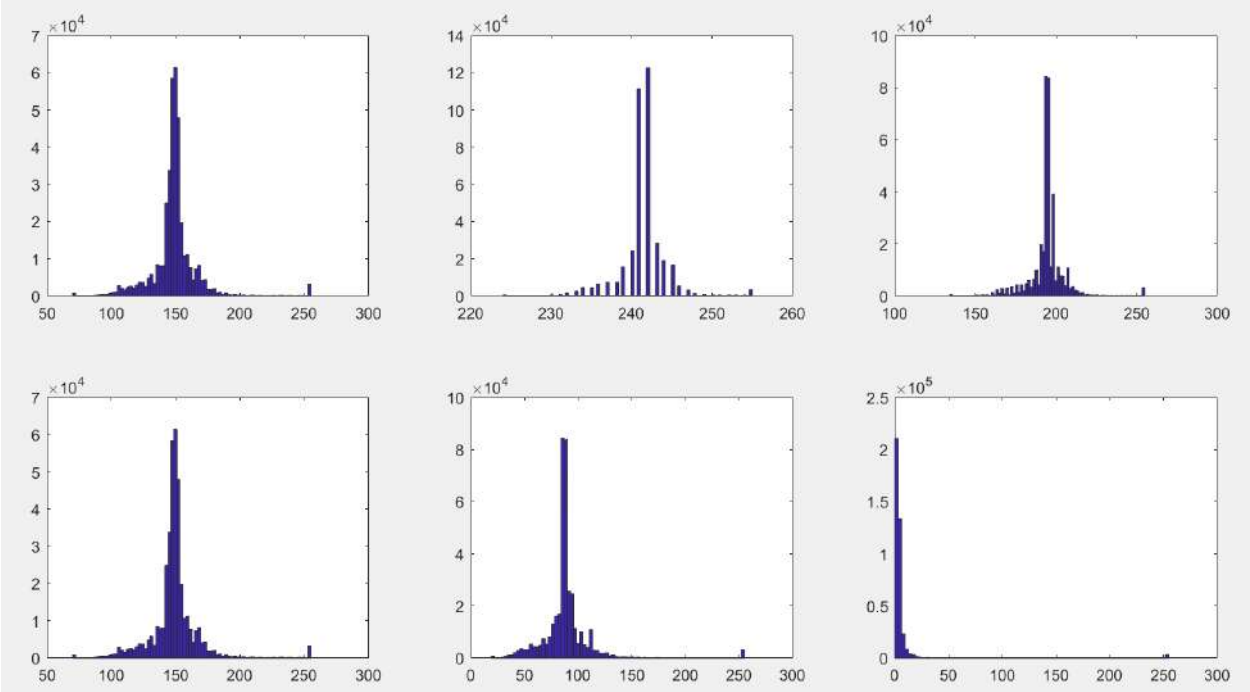


Рисунок 110 - Гистограмма для засвеченного изображения

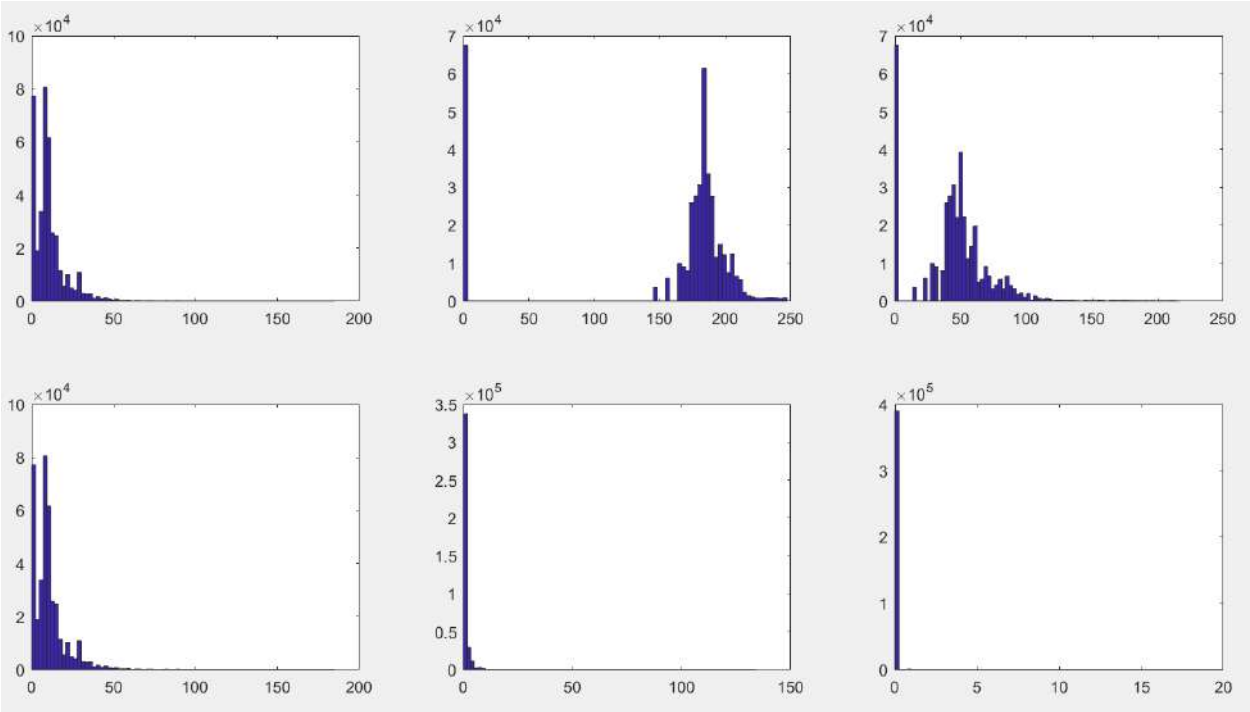


Рисунок 111 - Гистограмма для затемненного изображения

По полученным данным можно убедиться, что вывод, сделанный в предыдущем пункте, верный. Также можно сказать, что гамма преобразование повышает контрастность изображении.

5.4 Синтез засвеченного, затемненного, а также сбалансированного изображения и применение к ним алгоритма выравнивания гистограмм

Рисунок 112 - Исходное сбалансированное изображение

Рисунок 113 - Полученное изображение

Рисунок 114 - Исходное засвеченное изображение

Рисунок 115 - Полученное изображение

Рисунок 116 - Исходное затемненное изображение

Рисунок 117 - Полученное изображение

Полученные изображения одинаковы то есть результаты работа алгоритма совпадают на сбалансированном, засвеченном и затемнённом изображениях.

5.5 Гистограмма

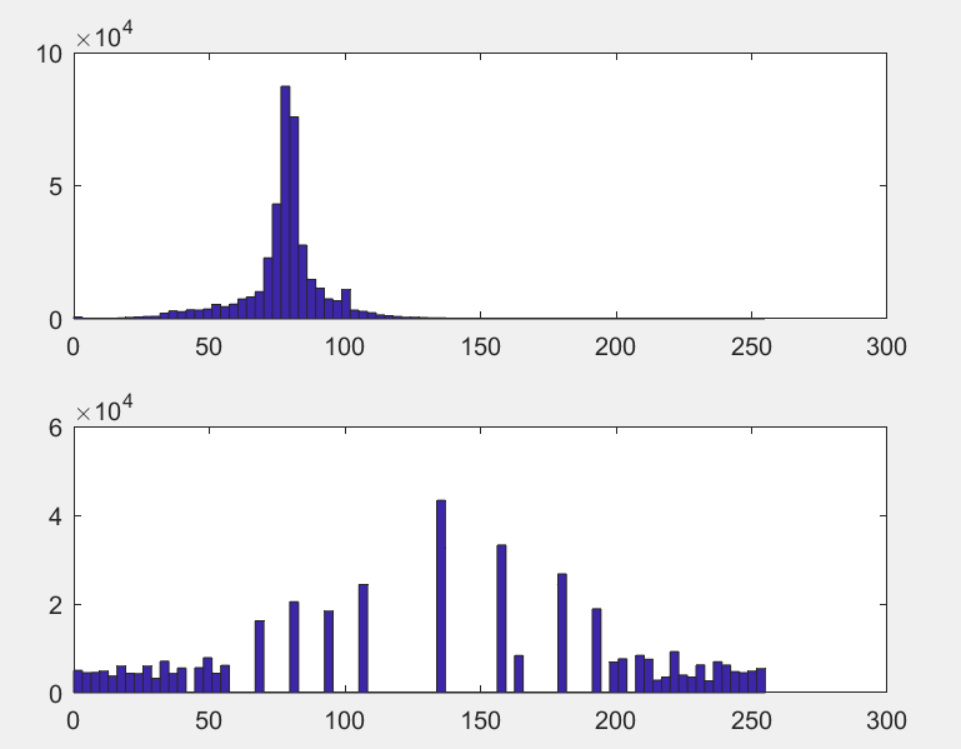


Рисунок 118 - Гистограмма для исходного и полученного сбалансированного изображения

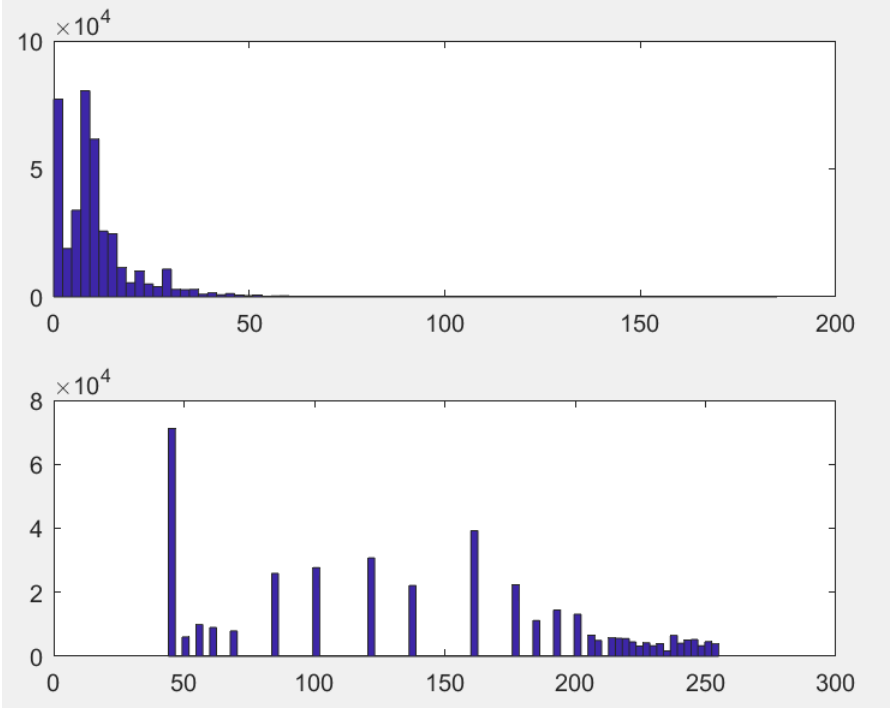


Рисунок 119 - Гистограмма исходного и полученного затемненного изображения

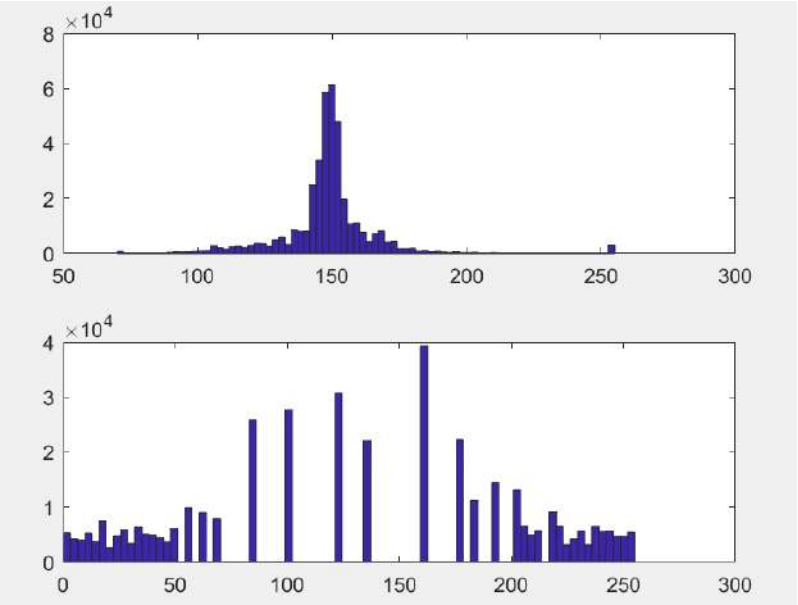


Рисунок 120 - Гистограмма исходного и полученного засвеченного изображения

Заметим, что у затемнённого и засвеченного изображений даже после применения метода выравнивания гистограмм крайние “пики” уходят достаточно неэффективно.

5.6 Методы построения карты контуров на основе градационных преобразований

Метод построение карты контуров заключается в следующем: в зависимости от выбора порога T выходной пиксель бинарного значения соответствует белому (интенсивность соответствующего пикселя исходного изображения превышает порог) или черному цвету (иначе). Необходимо синтезировать градационную функцию преобразования для построения бинарного изображения. Порог T принадлежит интервалу [16; 240] и изменяется с фиксированным шагом. Возьмём этот шаг равным 32.

Рисунок 121 - Бинарное изображение при $T = 16$

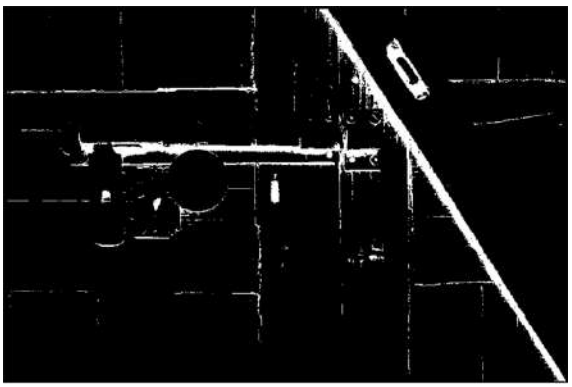


Рисунок 122 - Бинарное изображение при $T = 48$

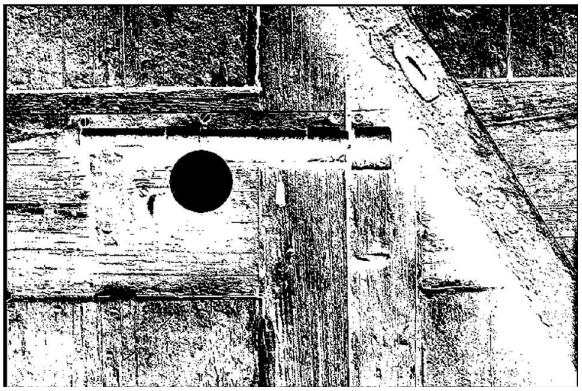


Рисунок 123 - Бинарное изображение при $T = 80$

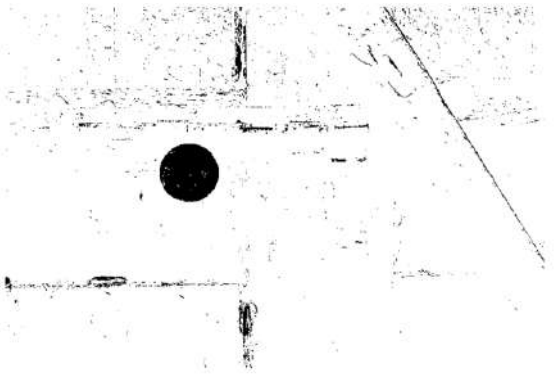


Рисунок 124 - Бинарное изображение при $T = 112$

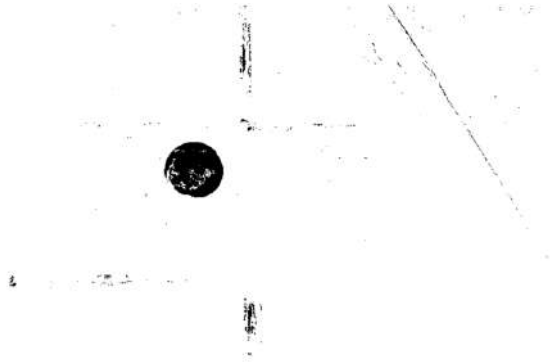


Рисунок 125 -Бинарное изображение при $T = 144$

Рисунок 126 - Бинарное изображение при $T = 176$

Рисунок 127 - Бинарное изображение при T = 208

В результате самым точным изображением оказалось изображение с T = 80. Отсюда можно сделать вывод, что значение T находящееся приблизительно в середине диапазона для сбалансированного изображения является наилучшим. При увеличении и уменьшении T с середины будет уменьшаться количество деталей на изображении.

5.7 Формирование наглядного бинарного изображения

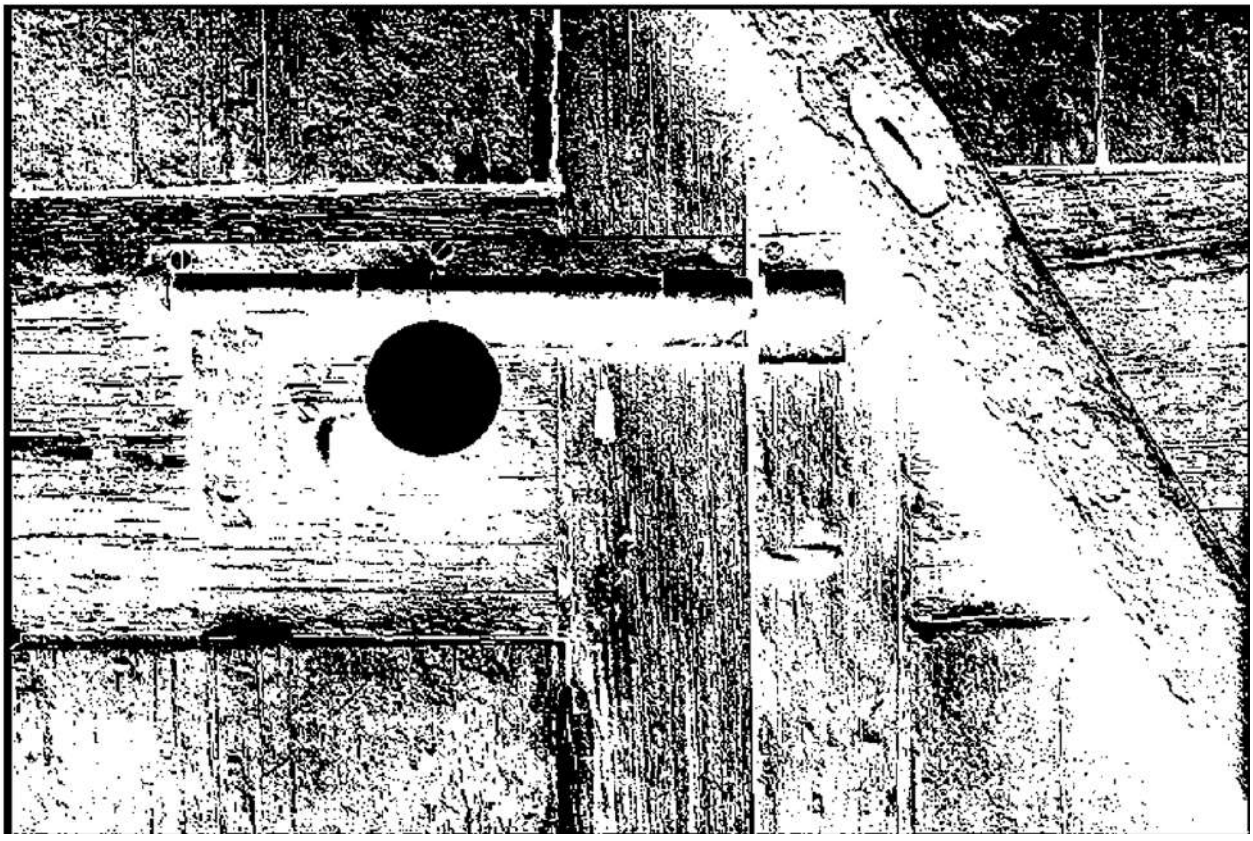


Рисунок 128 - Бинарное изображение при T = 80

Значение T находящееся приблизительно в середине диапазона для сбалансированного изображения является лучшим. Значения гораздо выше середины являются в данном случае худшими, т.к. на изображении просто белое. Значения гораздо больше среднего с увеличением порога T теряют точность, однако разобрать возможно.

Выводы

В ходе выполнения лабораторной работы на исходное изображение были наложены аддитивный и импульсный шумы. Был сделан вывод о том, что импульсный шум сильнее ухудшает качество изображения. Эти ухудшения можно оценить визуально и по низким значениям PSNR. Такие выводы объясняются тем, что пиксели при их изменении принимают максимально и минимально возможный значения: 255 и 0.

В ходе работы были применены три фильтра для подавления шума: фильтр скользящего среднего, гауссовский и медианный. Был сделан вывод о том, что лучшим фильтром с точки зрения PNSR для маленьких значений шума является фильтр Гаусса. Это объясняется тем, что этот фильтр в отличие от остальных, имеет дополнительный параметр $\sigma_{\text{фильтр}}$. Однако стоит заметить, что при больших значениях шума, все три фильтра работают примерно одинаково и слабо подавляют шум. Наихудший результат визуально дает медианный фильтр, т.к. вносит большую нечеткость и зернистость. Однако при наложении импульсного шума этот фильтр дает хорошие результаты.

В ходе работы был применен оператор Лапласа с целью выделения контура изображения. Был сделан вывод о том, что при увеличении α контур становится четче, но увеличивается яркость изображения.

В ходе работы к исходному изображению был применен оператор Собеля. Был сделан вывод о том, что оператор Собеля наиболее точную бинарную карту дает при пороге thr=120 (или же при thr=90), при меньших значениях на изображении очень много белых пикселей, при больших наоборот, было детектировано мало контуров.

Были применены методы на основе опорных точек, выравнивания гистограмм и гамма-преобразование. Метод выравнивания гистограмм для засвеченного и затемнённого изображений приближают картинки к сбалансированному и выравнивают по всему диапазону значений пиксели. При этом наиболее точная карта контуров получается при значении порога T = 80. Отсюда можно сделать вывод, что значение T находящееся приблизительно в середине диапазона для сбалансированного изображения является лучшим. Значения гораздо выше середины являются в данном случае худшими, т.к. на изображении просто белое. Значения гораздо больше среднего с увеличением порога T теряют точность, однако разобрать возможно.

Листинг программы:

Main.cpp

```
#include <iostream>

#include <fstream>

#include <locale.h>

#include <vector>

#include "bmp.h"

#include "AdditiveNoise.h"

#include "ImpulseNoise.h"

#include "AverageMethod.h"

#include "GaussianFilter.h"

#include "MedianFilteringMethod.h"

#include "LaplaceOperator.h"

#include "SobelOperator.h"

#include"AnchorPoints.h"

#include "GammaTransform.h"

#include "HistogramEqualization.h"

#include "GradationTransform.h"


using namespace std;


void get_YCbCr(YCbCr** ycbcr, RGB** rgb, int height, int width)
{
     for (int i = 0; i < height; i++) {
          for (int j = 0; j < width; j++) {
```

```
                ycbcr[i][j].Y = ((double)rgb[i][j].R * 0.299 +
(double)rgb[i][j].G * 0.587 + (double)rgb[i][j].B * 0.114);
                ycbcr[i][j].Cb = (0.5643 * ((double)rgb[i][j].B -
ycbcr[i][j].Y) + 128);
                ycbcr[i][j].Cr = 0.7132 * (((double)rgb[i][j].R -
ycbcr[i][j].Y) + 128);
          }
     }
}



int main() {
     setlocale(LC_ALL, "Russian");

     BITMAPFILEHEADER bfh;
     BITMAPINFOHEADER bih;
     FILE* f = fopen("Original.bmp", "rb");
     if (f == NULL) {
          return -1;
     }
     RGB** rgb = read_bmp(f, &bfh, &bih);
     fclose(f);
     int height = bih.biHeight;
     int width = bih.biWidth;

     YCbCr** ycbcr = new YCbCr * [height];
     for (int i = 0; i < height; i++) {
```

```
            ycbcr[i] = new YCbCr[width];
        }
        get_YCbCr(ycbcr, rgb, height, width);



        cout << "2.1.1 Реализация модели аддитивного шума" << endl;
        int sigma = 80;
        AdditiveNoise additive_noise(ycbcr, rgb, height, width,
&bfh, &bih, sigma);
        additive_noise.get_image("2.1.1GaussianNoise.bmp");
        cout << "Успешно выполенено для sigma = " << sigma << endl
<< endl;



        cout << "2.1.2 Реализация модели импульсного шума" << endl;
        double p_a = 0.05;
        double p_b = 0.05;
        ImpulseNoise impulse_noise(ycbcr, rgb, height, width, &bfh,
&bih, p_a, p_b);
        impulse_noise.get_image("2.1.2ImpulseNoise.bmp");
        cout << "Успешно выполенено для p_a = " << p_a << " и p_b =
" << p_b << endl << endl;

        cout << "2.1.3 Построение графиков PSNR" << endl;
        cout << "Модель аддитивного шума:" << endl;
        additive_noise.PSNR_graph();
```

```
    cout << endl << "Модель импульсного шума:" << endl;
    impulse_noise.PSNR_graph();
    cout << endl;



    cout << "2.1.4.1 Метод скользящего среднего" << endl;
    int R = 1;
    int sigma2 = 30;
    AdditiveNoise additive_noise2(ycbcr, rgb, height, width,
&bfh, &bih, sigma2);
    additive_noise2.get_image("2.1.4.1GaussianNoise.bmp");
    MovingAverageMethod
average_method(ycbcr,additive_noise2.get_YCbCr(), height, width,
&bfh, &bih, R);
    average_method.get_image("2.1.4.1AverageMethod.bmp");
    cout << "Успешно выполенено для R = " << R  << " и sigma = "
<< sigma2 << endl << endl;


    cout << "2.1.4.2 Подбор размера окна для метода скользящего
среднего" << endl;
    vector<int> sigma_vector = { 1,10,30,50,80 };
    int R2 = 1;
    for (int i = 0; i < sigma_vector.size(); i++) {
        cout << "sigma = " << sigma_vector[i] << endl;
        AdditiveNoise a(ycbcr, rgb, height, width, &bfh, &bih,
sigma_vector[i]);
```

```
        MovingAverageMethod average_method(ycbcr,
a.get_YCbCr(), height, width, &bfh, &bih, R2);
        average_method.check_radius();
    }
    cout << endl;



    cout << "2.1.4.3 Фильтр Гаусса" << endl;
    int R3 = 1;
    double sigma_filter = 0.1;
    double sigma3 = 30;
    AdditiveNoise an(ycbcr, rgb, height, width, &bfh, &bih,
sigma3);
    an.get_image("2.1.4.3GaussianNoise.bmp");
    GaussianFilter gaussian_method(rgb, an.get_YCbCr(), ycbcr,
height, width, &bfh, &bih,R3 ,sigma3);
    gaussian_method.get_image("2.1.4.3GaussianMethod.bmp");
    cout << "Успешно выполенено для R = " << R3 << " и sigma = "
<< sigma_filter << endl << endl;



    cout << "2.1.4.5 Графики PSNR" << endl;
    GaussianFilter gf(rgb, ycbcr, height, width, &bfh, &bih);
    //ОЧЕНЬ МЕДЛЕННО
    //gf.check_PSNR();
    cout << "Графики построены" << endl;
```

```
    cout << "plot 'R1sigmaNoise1.txt' w l, 'R1sigmaNoise10.txt'
w l, 'R1sigmaNoise30.txt' w l, 'R1sigmaNoise50.txt' w l,
'R1sigmaNoise80.txt' w l" << endl;

    cout << "plot 'R3sigmaNoise1.txt' w l, 'R3sigmaNoise10.txt'
w l, 'R3sigmaNoise30.txt' w l, 'R3sigmaNoise50.txt' w l,
'R3sigmaNoise80.txt' w l" << endl;

    cout << "plot 'R5sigmaNoise1.txt' w l, 'R5sigmaNoise10.txt'
w l, 'R5sigmaNoise30.txt' w l, 'R5sigmaNoise50.txt' w l,
'R5sigmaNoise80.txt' w l" << endl;

    cout << endl;



    cout << "2.1.4.4 Максимальные значения PSNR" << endl;

    gf.find_max_PSNR();

    cout << endl;



    cout << "2.1.4.7 Метод медианной фильтрации" << endl;

    int R8 = 1;

    int sigma8 = 30;

    AdditiveNoise additive_noise4(ycbcr, rgb, height, width,
&bfh, &bih, sigma8);

    additive_noise4.get_image("2.1.4.7GaussianNoise.bmp");

    MedianFilteringMethod
median_method(ycbcr,additive_noise4.get_YCbCr(), height, width,
&bfh, &bih, R8);

    median_method.get_image("2.1.4.7MedianMethod.bmp");
```

```
    cout << "Успешно выполенено для R = " << R8  << " и sigma =
" << sigma8 << endl << endl;


    //ОЧЕНЬ МЕДЛЕННО РАБОТАЕТ
    cout << "2.1.4.8 Подбор размера окна для метода медианной
фильтрации" << endl;
        vector<int> sigma_vector2 = { 1,10,30,50,80 };
        int R9 = 1;
        for (int i = 0; i < sigma_vector2.size(); i++) {
            cout << "sigma = " << sigma_vector2[i] << endl;
            AdditiveNoise a(ycbcr, rgb, height, width, &bfh,
&bih, sigma_vector2[i]);
            MedianFilteringMethod median_method1(ycbcr,
a.get_YCbCr(), height, width, &bfh, &bih, R9);
            median_method1.check_radius();
        }
        cout << endl;




    //МЕДЛЕННО
    cout << "2.1.4.9 Сравнение результатов при разных методах
фильтрации" << endl;
    vector<int> sigma_vector9 = { 1,10,30,50,80 };
    vector<int> best_R_average = {1,1,2,3,5};
    vector<int> best_R_gaussian = {1,1,5,2,5};
```

```
	vector<int> best_R_median = {1,1,3,4,5};
	vector<double> best_sigma_filter_gaussian = {0.25, 0.75, 1.5, 2, 2};
	vector<double> PSNR_average;
	vector<double> PSNR_gaussian;
	vector<double> PSNR_median;
	ofstream file1;
	file1.open("2.1.4.10AdditiveNoise.txt");
	ofstream file2;
	file2.open("2.1.4.10AverageMethod.txt");
	ofstream file3;
	file3.open("2.1.4.10GaussianMethod.txt");
	ofstream file4;
	file4.open("2.1.4.10MedianMethod.txt");
	for (int i = 0; i < sigma_vector9.size(); i++) {
		cout << "sigma = " << sigma_vector9[i] << endl;
		AdditiveNoise a9(ycbcr, rgb, height, width, &bfh, &bih, sigma_vector9[i]);
		a9.get_image(("2.1.4.9.AdditiveNoise" + to_string(sigma_vector9[i]) + ".bmp").c_str());
		file1 << (int)sigma_vector9[i] << " " << (double)a9.PSNR() << endl;
		MovingAverageMethod am9(ycbcr, a9.get_YCbCr(), height, width, &bfh, &bih, best_R_average[i]);
		am9.get_image(("2.1.4.9.AverageMethod" + to_string(sigma_vector9[i]) + ".bmp").c_str());
		file2 << (int)sigma_vector9[i] << " " << (double)am9.PSNR() << endl;
```

```
        GaussianFilter gf9(rgb, a9.get_YCbCr(), ycbcr, height,
width, &bfh, &bih, best_R_gaussian[i],
best_sigma_filter_gaussian[i]);

        gf9.get_image(("2.1.4.9.GaussianMethod" +
to_string(sigma_vector9[i]) + ".bmp").c_str());

        file3 << (int)sigma_vector9[i] << " " <<
(double)gf9.PSNR() << endl;

        MedianFilteringMethod mm9(ycbcr, a9.get_YCbCr(),
height, width, &bfh, &bih, best_R_median[i]);

        mm9.get_image(("2.1.4.9.MedianMethod" +
to_string(sigma_vector9[i]) + ".bmp").c_str());

        file4 << (int)sigma_vector9[i] << " " <<
(double)mm9.PSNR() << endl;

    }

    file1.close();

    file2.close();

    file3.close();

    file4.close();

    cout << "plot '2.1.4.10AdditiveNoise.txt' w l,
'2.1.4.10AverageMethod.txt' w l, '2.1.4.10GaussianMethod.txt' w
l, '2.1.4.10MedianMethod.txt' w l" << endl;

    cout << endl;



    cout << "2.1.5.2 Вычисление значения PSNR" << endl;

    vector<double> p = { 0.025, 0.05, 0.125, 0.25 };

        for (int i = 0; i < p.size(); i++) {
```

```
                ImpulseNoise impulse_noise1(ycbcr, rgb, height,
width, &bfh, &bih, p[i], p[i]);
                impulse_noise1.get_image(("2.1.5.1ImpulseNoise" +
to_string(p[i]) + ".bmp").c_str());
                impulse_noise1.PSNR();
          }
          cout << endl;

     cout << "2.1.5.3 Метод медианной фильтрации для импульсного
шума" << endl;
     vector<double> p3 = { 0.025, 0.05, 0.125, 0.25 };
     vector<double> R33 = {3};
     for (int i = 0; i < p3.size(); i++) {
          for (int j = 0; j < R33.size(); j++) {
                ImpulseNoise impulse_noise3(ycbcr, rgb, height,
width, &bfh, &bih, p3[i], p3[i]);
                MedianFilteringMethod m3(ycbcr,
impulse_noise3.get_YCbCr(), height, width, &bfh, &bih, R33[j]);
                m3.get_image(("2.1.5.3MedianMethod" +
to_string(p3[i]) + ".bmp").c_str());
          }
     }
     cout << endl;

     cout << "2.1.5.3 График PSNR(R)" << endl;
     vector<double> p35 = { 0.025, 0.05, 0.125, 0.25 };
     vector<double> R35 = { 1,2,3,4};
```

```
    for (int i = 0; i < p35.size(); i++) {
        vector<double> res;
        for (int j = 0; j < R35.size(); j++) {
            ImpulseNoise impulse_noise5(ycbcr, rgb, height,
width, &bfh, &bih, p35[i], p35[i]);
            MedianFilteringMethod m5(ycbcr,
impulse_noise5.get_YCbCr(), height, width, &bfh, &bih, R35[j]);
            res.push_back(m5.PSNR());
        }
        ofstream file_PSNR_Median;
        file_PSNR_Median.open(("2.1.5.3MedianPSNR" +
to_string(p35[i]) + ".txt").c_str());
        for (int k = 0; k < R35.size(); k++)
            file_PSNR_Median << R35[k] << " " << res[k] <<
endl;

    }
    cout << endl;
    cout << "plot '2.1.5.3MedianPSNR0,025000.txt' w l,
'2.1.5.3MedianPSNR0,050000.txt' w l,
'2.1.5.3MedianPSNR0,125000.txt' w l,
'2.1.5.3MedianPSNR0,250000.txt' w l" << endl;
```

```
    //2.2


    cout << "2.2.1.1 Примение оператора Лапласа I_new = L(I) к
изображению" << endl;
    LaplaceOperator lp1(ycbcr, height, width, &bfh, &bih);
    lp1.create_laplace_operator1(1,-4);
    lp1.get_operator_laplace_image("2.2\\2.2.1.1.LaplaceOperator
1.bmp");
    lp1.get_image_response("2.2\\2.2.1.1.LaplaceOperatorImage1.b
mp");


    cout << endl <<  "2.2.1.2 Формирование изображения по
отклику" << endl;
    LaplaceOperator lp2(ycbcr, height, width, &bfh, &bih);
    lp2.create_laplace_operator1(-1, 4);
    lp2.get_operator_laplace_image("2.2\\2.2.1.1.LaplaceOperator
2.bmp");
    lp2.get_image_response("2.2\\2.2.1.1.LaplaceOperatorImage2.b
mp");


    cout << endl << "2.2.1.3 Усиление высоких частот" << endl;
    lp2.high_frequency("2.2\\2.2.1.3.LaplaceOperatorImage.bmp");




    cout << endl << "2.2.1.4-5 Изображения с маской с различными
alpha" << endl;
```

```
    vector<double> alpha = { 1, 1.1, 1.2, 1.3, 1.4, 1.5 };
    for (int i = 0; i < alpha.size(); i++) {
        LaplaceOperator lp(ycbcr, height, width, &bfh, &bih);
        lp.create_high_frequency_alpha(alpha[i],
("2.2\\2.2.1.4.LaplaceOperatorImage" + to_string(alpha[i]) +
".bmp").c_str());
        cout << "alpha = " << alpha[i] << " Средняя яркость = "
<< lp.average_bright() << endl;
        lp.get_freq_new(("2.2\\2.2.1.7.LaplaceOperatorImage" +
to_string(alpha[i]) + ".txt").c_str());
    }
    LaplaceOperator lp_original(ycbcr, height, width, &bfh,
&bih);
    lp_original.get_freq_original("2.2\\2.2.1.7.LaplaceOperatorI
mage.txt");



    cout << "2.2.2.1 Применение оператора Собеля " << endl;
    SobelOperator sobel_original(ycbcr, height, width, &bfh,
&bih);
    sobel_original.sobel_operator(127);
    sobel_original.get_image("2.2\\2.2.2.1.SobelOperator.bmp");

    cout << "2.2.2.4 Применение оператора Собеля с различными
значениями thr" << endl;
    vector<int> thr = { 30, 60, 90, 120, 150, 180, 210};
    for (int i = 0; i < thr.size(); i++) {
```

```
            SobelOperator s(ycbcr, height, width, &bfh, &bih);
            s.sobel_operator(thr[i]);
            s.get_image(("2.2\\2.2.2.4.SobelOperatorImage" +
to_string(thr[i]) + ".bmp").c_str());
      }

      cout << "2.2.2.5 Карта направлений градиентов" << endl;
      sobel_original.gradient("2.2\\2.2.2.5.SobelOperatorGradient.
bmp");


      //2.3
      cout << "2.3.1 Метод опорных точек" << endl;
      AnchorPoints a_original(ycbcr, height, width, &bfh, &bih);
      a_original.get_freq_original("2.3\\2.3.1.3OriginalBefore.txt
");
      a_original.anchor_points(65, 40, 180, 210);
      a_original.get_image("2.3\\2.3.1.2Original2Points.bmp");
      a_original.get_freq_new("2.3\\2.3.1.3OriginalAfter.txt");

      AnchorPoints ad(ycbcr, height, width, &bfh, &bih);
      ad.blackout(70);
      ad.get_image("2.3\\2.3.1.1DarkImage.bmp");
      ad.get_freq_new("2.3\\2.3.1.3DarkImageBefore.txt");
      ad.anchor_points(20,70,140,220);
      ad.get_image("2.3\\2.3.1.2DarkImage2Points.bmp");
      ad.get_freq_new("2.3\\2.3.1.3DarkImageAfter.txt");
```

```
	AnchorPoints al(ycbcr, height, width, &bfh, &bih);
	al.lightening(70);
	al.get_image("2.3\\2.3.1.1LightImage.bmp");
	al.get_freq_new("2.3\\2.3.1.3LightImageBefore.txt");
	al.anchor_points(120,20,210,180);
	al.get_image("2.3\\2.3.1.2LightImage2Points.bmp");
	al.get_freq_new("2.3\\2.3.1.3LightImageAfter.txt");



	cout << "2.3.2 Гамма преобразование" << endl;
	vector<double> gamma = { 0.1, 0.5, 1, 2, 8 };
	//original
	for (int i = 0; i < gamma.size(); i++) {
		GammaTransform g_original(ycbcr, height, width, &bfh,
&bih);
		g_original.gamma_transform(1, gamma[i]);

	g_original.get_image(("2.3\\2.3.2.GammaTransformOriginal" +
to_string(gamma[i]) + ".bmp").c_str());

	g_original.get_freq_new(("2.3\\2.3.3.GammaTransformOriginalA
fter" + to_string(gamma[i]) + ".txt").c_str());
	}

	//dark
	for (int i = 0; i < gamma.size(); i++) {
```

```
          GammaTransform g_dark(ycbcr, height, width, &bfh,
&bih);
          g_dark.blackout(70);
          g_dark.gamma_transform(1, gamma[i]);
          g_dark.get_image(("2.3\\2.3.2.GammaTransformDark" +
to_string(gamma[i]) + ".bmp").c_str());

     g_dark.get_freq_new(("2.3\\2.3.3.GammaTransformDarkAfter" +
to_string(gamma[i]) + ".txt").c_str());
     }

     //light
     for (int i = 0; i < gamma.size(); i++) {
          GammaTransform g_light(ycbcr, height, width, &bfh,
&bih);
          g_light.lightening(70);
          g_light.gamma_transform(1, gamma[i]);
          g_light.get_image(("2.3\\2.3.2.GammaTransformLight" +
to_string(gamma[i]) + ".bmp").c_str());

     g_light.get_freq_new(("2.3\\2.3.3.GammaTransformLightAfter"
+ to_string(gamma[i]) + ".txt").c_str());
     }

     cout << "2.3.3 Метод выравнивания гистограмм" << endl;
     HistogramEqualization h_original(ycbcr, height, width, &bfh,
&bih);
```

```
    h_original.histogram_equalization();
    h_original.get_image("2.3\\2.3.3.3HistogramOriginalAfter.bmp
");
    h_original.get_freq_new("2.3\\2.3.3.3HistogramOriginalAfter.
txt");

    HistogramEqualization h_dark(ycbcr, height, width, &bfh,
&bih);
    h_dark.blackout(70);
    h_dark.histogram_equalization();
    h_dark.get_image("2.3\\2.3.3.3HistogramDarkAfter.bmp");
    h_dark.get_freq_new("2.3\\2.3.3.3HistogramDarkAfter.txt");

    HistogramEqualization h_light(ycbcr, height, width, &bfh,
&bih);
    h_light.lightening(70);
    h_light.histogram_equalization();
    h_light.get_image("2.3\\2.3.3.3HistogramLightAfter.bmp");
    h_light.get_freq_new("2.3\\2.3.3.3HistogramLightAfter.txt");

    cout << "2.3.4 5.6 Методы построения карты контуров на
основе градационных преобразований" << endl;
    vector<int> T = { 16, 48, 80, 112, 144, 176, 208, 240 };
    GradationTransform grad(ycbcr, height, width, &bfh, &bih);
    for (int i = 0; i < T.size(); i++) {
        grad.gradation_transform(T[i]);
```

```
          grad.get_image(("2.3\\2.3.4GradationTransform" +
to_string(T[i]) + ".bmp").c_str());

     }




     return 0;

}
```

AdditiveNoise.h

```
#ifndef gaussianNoise
#define gaussianNoise
#include <iostream>
#include <fstream>
#include <vector>
#include "bmp.h"

#define Pi 3.141592653589793

using namespace std;

class AdditiveNoise {
private:
     BITMAPFILEHEADER* bfh;
     BITMAPINFOHEADER* bih;
     YCbCr** new_ycbcr;
     YCbCr** ycbcr;
     int height;
     int width;
     RGB** rgb;
     double** noise;
     double sigma;
     unsigned char min_R = 0;;
     unsigned char min_G = 0;
     unsigned char min_B = 0;
     unsigned char max_R = 0;
     unsigned char max_G = 0;
     unsigned char max_B = 0;
public:
     AdditiveNoise(YCbCr** y, RGB** color, int h, int w,
BITMAPFILEHEADER* bf, BITMAPINFOHEADER* bi, double s) {
          height = h;
          width = w;
          rgb = color;
          bfh = bf;
          bih = bi;
          sigma = s;
```

```
            ycbcr = y;

            new_ycbcr = new YCbCr * [height];
            noise = new double* [height];
            for (int i = 0; i < height; i++) {
                 new_ycbcr[i] = new YCbCr[width];
                 noise[i] = new double[width];
            }

            //find_min_max();
            sigma = s;
            noise_overlay();

      }

      void get_image(const char* filename) {
            FILE* file;
            file = fopen(filename, "wb");
            write_bmp_ycbcr(file, new_ycbcr, bfh, bih, height,
width);
            fclose(file);
      }
      ~AdditiveNoise() {
            for (int i = 0; i < height; i++) {
                 delete(noise[i]);
                 delete(new_ycbcr[i]);
            }
            delete noise;
            delete new_ycbcr;
      }

      void noise_overlay() {
            srand(time(NULL));
            for (int i = 0; i < height; i++) {
                 for (int j = 0; j < width; j += 2) {
                      double r = (double)(rand() % 2000 - 1000) /
1000;
                      double phi = (double)(rand() % 2000 - 1000) /
1000;
                      double s = (r * r) + (phi * phi);
                      while (s > 1 || s == 0) {
                           r = (double)(rand() % 2000 - 1000) /
1000;
                           phi = (double)(rand() % 2000 - 1000) /
1000;
                           s = (r * r) + (phi * phi);
                      }
                      noise[i][j] = sigma * r * sqrt(-2 * log(s) /
s);
                      noise[i][j + 1] = sigma * phi * sqrt(-2 *
log(s) / s);
                 }
            }
```

```
            for (int i = 0; i < height; i++) {
                for (int j = 0; j < width; j++) {
                    new_ycbcr[i][j].Y = clipping(ycbcr[i][j].Y +
noise[i][j]);
                }
            }
        }


        /*void find_min_max() {
            for (int i = 0; i < height; i++) {
                for (int j = 0; j < width; j++) {
                    if (rgb[i][j].R > max_R) max_R = rgb[i][j].R;
                    if (rgb[i][j].G > max_G) max_G = rgb[i][j].G;
                    if (rgb[i][j].B > max_B) max_B = rgb[i][j].B;
                    if (rgb[i][j].R < min_R) min_R = rgb[i][j].R;
                    if (rgb[i][j].G < min_G) min_G = rgb[i][j].G;
                    if (rgb[i][j].B < min_B) min_B = rgb[i][j].B;
                }
            }
        }*/
        double clipping(double value) {
            if (value > 255.0) {
                value = 255.0;
            }
            if (value < 0.0) {
                value = 0;
            }
            return round(value);
        }

        double PSNR() {
            double tmp = width * height * pow(256 - 1, 2);
                double PSNR= 0;
                for (int i = 0; i < height; i++) {
                    for (int j = 0; j < width; j++) {
                        PSNR += pow((ycbcr[i][j].Y -
new_ycbcr[i][j].Y), 2);
                    }
                }
                PSNR = 10 * log10(tmp / PSNR);
                cout << "PSNR = " << PSNR << endl;
                return PSNR;
        }

        void PSNR_graph() {
            ofstream file;
            file.open("2.1.3PSNR(sigma).txt");
            vector<double> x = {1,10,30,50,80};
            for (int i = 0; i < x.size(); i++) {
                sigma = x[i];
                noise_overlay();
                file << sigma << " " << PSNR() << endl;
```

```
            }
            file.close();
      }
      YCbCr** get_YCbCr() {
            return new_ycbcr;
      }
};

#endif gaussianNoise
```

ImpulseNoise.h

```
#include <iostream>
#include <fstream>
#include "bmp.h"
#define Pi 3.141592653589793

using namespace std;


class ImpulseNoise {
private:
      BITMAPFILEHEADER* bfh;
      BITMAPINFOHEADER* bih;
      YCbCr** ycbcr;
      YCbCr** new_ycbcr;
      int height;
      int width;
      RGB** rgb;
      double** noise;
      double p_a = 0;
      double p_b = 0;
public:
      ImpulseNoise(YCbCr** y, RGB** color, int h, int w,
BITMAPFILEHEADER* bf, BITMAPINFOHEADER* bi, double p1, double
p2) {
            height = h;
            width = w;
            rgb = color;
            bfh = bf;
            bih = bi;
            p_a = p1;
            p_b = p2;
            ycbcr = y;

            new_ycbcr = new YCbCr * [height];
            noise = new double* [height];
            for (int i = 0; i < height; i++) {
                  new_ycbcr[i] = new YCbCr[width];
                  noise[i] = new double[width];
            }
```

```
            noise_overlay();
        }
        void get_image(const char* filename) {
            FILE* file;
            file = fopen(filename, "wb");
            write_bmp_ycbcr(file, new_ycbcr, bfh, bih, height,
width);
            fclose(file);
        }
        ~ImpulseNoise() {
            for (int i = 0; i < height; i++) {
                delete(noise[i]);
                delete(new_ycbcr[i]);
            }
            delete noise;
            delete new_ycbcr;
        }

        void noise_overlay() {
            srand(time(NULL));
            for (int i = 0; i < height; i++) {
                for (int j = 0; j < width; j++) {
                    double tmp = (double)(rand() % 100) / 100;
                    if (tmp <= p_a) {
                        new_ycbcr[i][j].Y = 0;
                        continue;
                    }
                    if (tmp <= (p_a + p_b)) {
                        new_ycbcr[i][j].Y = 255;
                        continue;
                    }
                    new_ycbcr[i][j].Y = ycbcr[i][j].Y;
                }
            }
        }

        double PSNR() {
            double tmp = width * height * pow(256 - 1, 2);
            double PSNR = 0;
            for (int i = 0; i < height; i++) {
                for (int j = 0; j < width; j++) {
                    PSNR += pow((ycbcr[i][j].Y -
new_ycbcr[i][j].Y), 2);
                }
            }
            PSNR = 10 * log10(tmp / PSNR);
            cout << "PSNR = " << PSNR << endl;
            return PSNR;
        }

        void PSNR_graph() {
            ofstream file;
```

```
            file.open("2.1.3PSNR(p_a,p_b).txt");
            vector<double> p = { 0.025,0.05,0.125,0.25};
            for (int i = 0; i < p.size(); i++) {
                  p_a = p[i];
                  p_b = p[i];
                  noise_overlay();
                  file << p[i] << " " << PSNR() << endl;
            }
            file.close();
      }

      YCbCr** get_YCbCr() {
            return new_ycbcr;
      }
};
```

AverageMethod.h

```
#include <iostream>
#include <fstream>
#include <vector>
#include "bmp.h"

class MovingAverageMethod {
private:
      BITMAPFILEHEADER* bfh;
      BITMAPINFOHEADER* bih;
      YCbCr** noise_ycbcr;
      YCbCr** result_ycbcr;
      YCbCr** ycbcr;
      int height;
      int width;
      int R = 0;

public:
      MovingAverageMethod(YCbCr** y1, YCbCr** y, int h, int w,
BITMAPFILEHEADER* bf, BITMAPINFOHEADER* bi, int r) {
            noise_ycbcr = y;
            ycbcr = y1;
            height = h;
            width = w;
            bfh = bf;
            bih = bi;
            R = r;


            result_ycbcr = new YCbCr * [height];
            for (int i = 0; i < height; i++)
                  result_ycbcr[i] = new YCbCr[width];

            average_method();
```

```
	}

	void get_image(const char* filename) {
		FILE* file;
		file = fopen(filename, "wb");
		write_bmp_ycbcr(file, result_ycbcr, bfh, bih, height,
width);
		fclose(file);
	}

	~MovingAverageMethod() {
		for (int i = 0; i < height; i++)
			delete (result_ycbcr[i]);
		delete result_ycbcr;
	}

	void average_method() {
		YCbCr** new_ycbcr = new YCbCr * [height + R * 2];
		for (int i = 0; i < height + R * 2; i++) {
			new_ycbcr[i] = new YCbCr[width + R * 2];
			for (int j = 0; j < width + R * 2; j++) {
				new_ycbcr[i][j].Y = 0;
			}
		}
		for (int i = 0; i < height; i++) {
			for (int j = 0; j < width; j++) {
				new_ycbcr[i + R][j + R].Y =
noise_ycbcr[i][j].Y;
			}
		}
		for (int i = 0; i < height; i++) {
			for (int j = 0; j < width; j++) {
				double tmp = 0;
				for (int k = -R; k <= R; k++) {
					for (int m = -R; m <= R; m++) {
						tmp += new_ycbcr[i + R + k][j + R +
m].Y;
					}
				}
				result_ycbcr[i][j].Y = tmp / pow((2 * R + 1),
2);
			}
		}
		for (int i = 0; i < height + 2 * R; i++)
			delete (new_ycbcr[i]);
		delete new_ycbcr;
	}

	double PSNR() {
		double tmp = width * height * pow(256 - 1, 2);
		double PSNR = 0;
		for (int i = 0; i < height; i++) {
			for (int j = 0; j < width; j++) {
```

```
                    PSNR += pow((ycbcr[i][j].Y -
result_ycbcr[i][j].Y), 2);
                }
            }
            PSNR = 10 * log10(tmp / PSNR);
            cout << "PSNR = " << PSNR << endl;
            return PSNR;
        }
        int check_radius() {
            double max_PSNR = 0;
            int max_R = 0;
            vector<int> radius = {1,2,3,4,5};
                for (int i = 0; i < radius.size(); i++)
                {
                    R = radius[i];
                    average_method();
                    double psnr = PSNR();
                    if (psnr > max_PSNR) {
                        max_PSNR = psnr;
                        max_R = R;
                    }
                }
                cout << "max PSNR = " << max_PSNR << " R = " <<
max_R << endl;
                return max_R;
        }

};
```

GaussianFilter.h

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <list>
#include "AdditiveNoise.h"
#include "bmp.h"

class GaussianFilter {
private:
    BITMAPFILEHEADER* bfh;
    BITMAPINFOHEADER* bih;
    YCbCr** noise_ycbcr;
    YCbCr** result_ycbcr;
    YCbCr** ycbcr;
    RGB** rgb;
    int height;
    int width;
    int R = 0;
    double sigma;
    double max_PSNR = 0;
```

```
     int max_R = 0;
     double max_sigma_filter = 0;

public:
     GaussianFilter(RGB** color, YCbCr** y, YCbCr** y1, int h,
int w, BITMAPFILEHEADER* bf, BITMAPINFOHEADER* bi, int r, double
s) {
          rgb = color;
          ycbcr = y1;
          noise_ycbcr = y;
          height = h;
          width = w;
          bfh = bf;
          bih = bi;
          R = r;
          sigma = s;

          result_ycbcr = new YCbCr * [height];
          for (int i = 0; i < height; i++)
               result_ycbcr[i] = new YCbCr[width];

          gaussian_method();

     }

     GaussianFilter(RGB** color, YCbCr** y1, int h, int w,
BITMAPFILEHEADER* bf, BITMAPINFOHEADER* bi) {
          rgb = color;
          ycbcr = y1;
          height = h;
          width = w;
          bfh = bf;
          bih = bi;


          result_ycbcr = new YCbCr * [height];
          for (int i = 0; i < height; i++)
               result_ycbcr[i] = new YCbCr[width];

     }
     void get_image(const char* filename) {
          FILE* file;
          file = fopen(filename, "wb");
          write_bmp_ycbcr(file, result_ycbcr, bfh, bih, height,
width);
          fclose(file);
     }


     ~GaussianFilter() {
          for (int i = 0; i < height; i++)
               delete (result_ycbcr[i]);
          delete result_ycbcr;
```

```
	}

	void gaussian_method() {
		YCbCr** new_ycbcr = new YCbCr * [height + R * 2];
		for (int i = 0; i < height + R * 2; i++) {
			new_ycbcr[i] = new YCbCr[width + R * 2];
			for (int j = 0; j < width + R * 2; j++) {
				new_ycbcr[i][j].Y = 0;
			}
		}
		for (int i = 0; i < height; i++) {
			for (int j = 0; j < width; j++) {
				new_ycbcr[i + R][j + R].Y =
noise_ycbcr[i][j].Y;
			}
		}

		for (int i = 0; i < height; i++) {
			for (int j = 0; j < width; j++) {
				double tmp = 0;
				double sum_w = 0;
				for (int k = -R; k <= R; k++) {
					for (int m = -R; m <= R; m++) {
						double w = exp((-((pow(k,2) +
pow(m,2))) / (2 * pow(sigma,2))));
						sum_w += w;
						tmp += w * new_ycbcr[i + R + k][j +
R + m].Y;
					}
				}
				result_ycbcr[i][j].Y = tmp / sum_w;
			}
		}

		for (int i = 0; i < height + 2 * R; i++)
			delete (new_ycbcr[i]);
		delete new_ycbcr;
	}

	double PSNR() {
		double tmp = width * height * pow(256 - 1, 2);
		double PSNR = 0;
		for (int i = 0; i < height; i++) {
			for (int j = 0; j < width; j++) {
				PSNR += pow((ycbcr[i][j].Y -
result_ycbcr[i][j].Y), 2);
			}
		}
		PSNR = 10 * log10(tmp / PSNR);
		//cout << "PSNR = " << PSNR << endl;
		return PSNR;
	}
```

```
     void write_to_file(const char* filename, double res) {
          ofstream file;
          file.open(filename, ios::app);
          file << sigma << " " << res << endl;
          file.close();
     }

     void find_max_PSNR() {
          vector<int> sigma_noise = { 1,10,30,50,80 };
          vector<int> radius = { 1,3,5 };
          for (int i = 0; i < sigma_noise.size(); i++) {
               max_PSNR = 0;
               max_R = 0;
               max_sigma_filter = 0;
               for (int j = 0; j < radius.size(); j++) {

                    double PSNR_from_file = 0;
                    double sigma_filter_from_file = 0;
                    int R_from_file = radius[j];
                    string filename = "R" +
to_string(R_from_file) + "sigmaNoise" +
to_string(sigma_noise[i]) + ".txt";
                    ifstream in(filename);
                    if (in.is_open())
                    {
                         while (in >> sigma_filter_from_file >>
PSNR_from_file)
                         {
                              if (max_PSNR < PSNR_from_file)
                              {
                                   max_PSNR = PSNR_from_file;
                                   max_R = R_from_file;
                                   max_sigma_filter =
sigma_filter_from_file;
                              }
                         }
                    }
                    in.close();
               }
               cout << "sigma_noise = " << sigma_noise[i] <<
endl;
               cout << "max PSNR = " << max_PSNR << " R = " <<
max_R << " sigma_filter = " << max_sigma_filter << endl << endl;
          }
     }

     void get_best_picrute() {
          sigma = max_sigma_filter;
          R = max_R;
          AdditiveNoise gaussian_noise(ycbcr, rgb, height, width,
bfh, bih, 1);
          noise_ycbcr = gaussian_noise.get_YCbCr();
          gaussian_method();
```

```
            get_image("2.1.4.6BestGaussianPSNR.bmp");
        }

        void check_PSNR() {
            vector<int> sigma_noise = { 1,10,30,50,80 };
            vector<int> radius = { 1,3,5 };
            vector<double> sigma_filter = { 0.1, 0.25, 0.5, 0.75,
1, 1.25, 1.5, 2 };
            for (int n = 0; n < radius.size(); n++) {
                R = radius[n];
                for (int i = 0; i < sigma_noise.size(); i++) {
                    AdditiveNoise gaussian_noise(ycbcr, rgb,
height, width, bfh, bih, sigma_noise[i]);
                    noise_ycbcr = gaussian_noise.get_YCbCr();
                    for (int j = 0; j < sigma_filter.size(); j++)
{
                        sigma = sigma_filter[j];
                        gaussian_method();
                        double psnr = PSNR();
                        string filename = "R" + to_string(R) +
"sigmaNoise" + to_string(sigma_noise[i]) + ".txt";
                        write_to_file(filename.c_str(), psnr);
                    }
                }
            }
        }
};
```

MedianFilteringMethod.h

```
#include <iostream>
#include <fstream>
#include <vector>
#include "bmp.h"
#include <algorithm>

using namespace std;

class MedianFilteringMethod {

private:
    BITMAPFILEHEADER* bfh;
    BITMAPINFOHEADER* bih;
    YCbCr** noise_ycbcr;
    YCbCr** result_ycbcr;
    YCbCr** ycbcr;
    int height;
    int width;
    int R = 0;

public:
```

```
     MedianFilteringMethod(YCbCr** y1, YCbCr** y, int h, int w,
BITMAPFILEHEADER* bf, BITMAPINFOHEADER* bi, int r) {
          noise_ycbcr = y;
          ycbcr = y1;
          height = h;
          width = w;
          bfh = bf;
          bih = bi;
          R = r;

          result_ycbcr = new YCbCr * [height];
          for (int i = 0; i < height; i++)
               result_ycbcr[i] = new YCbCr[width];

          median_method();
     }

     ~MedianFilteringMethod() {
          for (int i = 0; i < height; i++)
               delete (result_ycbcr[i]);
          delete result_ycbcr;
     }

     void get_image(const char* filename) {
          FILE* file;
          file = fopen(filename, "wb");
          write_bmp_ycbcr(file, result_ycbcr, bfh, bih, height,
width);
          fclose(file);
     }

     void median_method() {
          YCbCr** new_ycbcr = new YCbCr * [height + R * 2];
          for (int i = 0; i < height + R * 2; i++) {
               new_ycbcr[i] = new YCbCr[width + R * 2];
               for (int j = 0; j < width + R * 2; j++) {
                    new_ycbcr[i][j].Y = 0;
               }
          }
          for (int i = 0; i < height; i++) {
               for (int j = 0; j < width; j++) {
                    new_ycbcr[i + R][j + R].Y =
noise_ycbcr[i][j].Y;
               }
          }
          for (int i = 0; i < height; i++) {
               for (int j = 0; j < width; j++) {
                    double tmp = 0;
                    vector<double> A;
                    for (int k = -R; k <= R; k++) {
                         for (int m = -R; m <= R; m++) {
                              A.push_back(new_ycbcr[i + R + k][j
+ R + m].Y);
```

```
                    }
                }
                sort(A.begin(), A.end());
                result_ycbcr[i][j].Y = A[A.size() / 2];
            }
        }
        for (int i = 0; i < height + 2 * R; i++)
            delete (new_ycbcr[i]);
        delete new_ycbcr;
    }
    double PSNR() {
        double tmp = width * height * pow(256 - 1, 2);
        double PSNR = 0;
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                PSNR += pow((ycbcr[i][j].Y -
result_ycbcr[i][j].Y), 2);
            }
        }
        PSNR = 10 * log10(tmp / PSNR);
        cout << "PSNR = " << PSNR << endl;
        return PSNR;
    }

    int check_radius() {
        double max_PSNR = 0;
        int max_R = 0;
        vector<int> radius = { 1,2,3,4,5 };
        for (int i = 0; i < radius.size(); i++)
        {
            R = radius[i];
            median_method();
            double psnr = PSNR();
            if (psnr > max_PSNR) {
                max_PSNR = psnr;
                max_R = R;
            }
        }
        cout << "max PSNR = " << max_PSNR << " R = " << max_R
<< endl;
        return max_R;
    }



};
```

LaplaceOperator.h

```
#include <iostream>
#include <fstream>
#include <vector>
#include "bmp.h"
```

```
using namespace std;

class LaplaceOperator {
private:
	YCbCr** ycbcr;
	int height;
	int width;
	BITMAPFILEHEADER* bfh;
	BITMAPINFOHEADER* bih;
	double** laplace_operator;
	YCbCr** new_ycbcr;
public:

	LaplaceOperator(YCbCr** y, int h, int w, BITMAPFILEHEADER*
bf, BITMAPINFOHEADER* bi) {
		ycbcr = y;
		height = h;
		width = w;
		bfh = bf;
		bih = bi;

		laplace_operator = new double* [height];
		for (int i = 0; i < height; i++) {
			laplace_operator[i] = new double[width];
		}

		new_ycbcr = new YCbCr * [height];
		for (int i = 0; i < height; i++)
			new_ycbcr[i] = new YCbCr[width];
	}

	void create_laplace_operator1(int w, int w_center) {
		for (int i = 0; i < height; i++) {
			for (int j = 0; j < width; j++) {
				if (i + 1 < height && i - 1 >= 0 && j + 1 <
width && j - 1 >= 0) {
					laplace_operator[i][j] =
w*ycbcr[i][j+1].Y + w*ycbcr[i][j-1].Y + w*ycbcr[i+1][j].Y +
w*ycbcr[i-1][j].Y + w_center * ycbcr[i][j].Y;
				}
				else
				{
					laplace_operator[i][j] = 0;
				}
			}
		}
	}
	void get_operator_laplace_image(const char* filename) {

		for (int i = 0; i < height; i++) {
			for (int j = 0; j < width; j++) {
```

```
                    new_ycbcr[i][j].Y =
clipping(laplace_operator[i][j]);
                }
            }
            FILE* file;
            file = fopen(filename, "wb");
            write_bmp_ycbcr(file, new_ycbcr, bfh, bih, height,
width);
            fclose(file);
        }

        void get_image_response(const char* filename) {
            for (int i = 0; i < height; i++) {
                for (int j = 0; j < width; j++) {
                    new_ycbcr[i][j].Y =
clipping(laplace_operator[i][j] + 128);
                }
            }
            FILE* file;
            file = fopen(filename, "wb");
            write_bmp_ycbcr(file, new_ycbcr, bfh, bih, height,
width);
            fclose(file);
        }

        double clipping(double value) {
            if (value > 255.0) {
                value = 255.0;
            }
            if (value < 0.0) {
                value = 0;
            }
            return round(value);
        }
        void high_frequency(const char* filename) {
            for (int i = 0; i < height; i++) {
                for (int j = 0; j < width; j++) {
                    new_ycbcr[i][j].Y =
clipping(laplace_operator[i][j] + ycbcr[i][j].Y);
                }
            }
            FILE* file;
            file = fopen(filename, "wb");
            write_bmp_ycbcr(file, new_ycbcr, bfh, bih, height,
width);
            fclose(file);
        }

        void create_high_frequency_alpha(double alpha, const char*
filename) {
            const int R = 1;
            double w[3][3] = { { 0, -1, 0 } ,{ -1, alpha + 4, -1
},{ 0, -1, 0 } };
```

```
            for (int i = 0; i < height; i++) {
                for (int j = 0; j < width; j++) {
                    double result = 0;
                    for (int k = -R; k <= R; k++) {
                        for (int m = -R; m <= R; m++) {
                            int x = i + k;
                            int y = j + m;
                            if (x < 0)
                                x = 0;
                            if (x > height - 1)
                                x = height - 1;
                            if (y < 0)
                                y = 0;
                            if (y > width - 1)
                                y = width - 1;
                            double tmp = ycbcr[x][y].Y;
                            result += tmp * w[k + 1][m + 1];
                        }
                    }
                    new_ycbcr[i][j].Y = (clipping(ycbcr[i][j].Y *
(alpha - 1) + result));
                }
            }
            FILE* file;
            file = fopen(filename, "wb");
            write_bmp_ycbcr(file, new_ycbcr, bfh, bih, height,
width);
            fclose(file);
        }

        double average_bright() {
            long double sum = 0;
            for (int i = 0; i < height; i++) {
                for (int j = 0; j < width; j++) {
                    sum += new_ycbcr[i][j].Y;
                }
            }
            return sum / (height * width);
        }

        void get_freq_original(const char* filename) {
            get_frequency(ycbcr, filename);
        }

        void get_freq_new(const char* filename) {
            get_frequency(new_ycbcr, filename);
        }
        void get_frequency(YCbCr** y, const char* filename) {
            ofstream out;
            out.open(filename);
            for (int i = 0; i < height; i++) {
                for (int j = 0; j < width; j++) {
                    out << y[i][j].Y << endl;
```

```
                }
            }
            out.close();
        }
};
```

SobelOperator.h

```
#include <iostream>
#include <fstream>
#include <vector>
#include "bmp.h"

using namespace std;

class SobelOperator {
private:
     YCbCr** ycbcr;
     int height;
     int width;
     BITMAPFILEHEADER* bfh;
     BITMAPINFOHEADER* bih;
     YCbCr** new_ycbcr;
     RGB** new_rgb;
     double** teta;
     double** Gh;
     double** Gv;
public:

     SobelOperator(YCbCr** y, int h, int w, BITMAPFILEHEADER* bf,
BITMAPINFOHEADER* bi) {
          ycbcr = y;
          height = h;
          width = w;
          bfh = bf;
          bih = bi;


          new_ycbcr = new YCbCr * [height];
          for (int i = 0; i < height; i++)
               new_ycbcr[i] = new YCbCr[width];

          new_rgb = new RGB * [height];
          for (int i = 0; i < height; i++)
               new_rgb[i] = new RGB[width];

          teta = new double * [height];
          for (int i = 0; i < height; i++)
               teta[i] = new double[width];

          Gh = new double* [height];
```

```
            for (int i = 0; i < height; i++)
                Gh[i] = new double[width];

            Gv = new double* [height];
            for (int i = 0; i < height; i++)
                Gv[i] = new double[width];
        }

        void sobel_operator(double thr) {
            const int R = 1;
            double I;
            const double mask_h[3][3] = { { -1, 0, 1 } ,{ -2, 0, 2
},{ -1, 0, 1 } };
            const double mask_v[3][3] = { { 1, 2, 1 } ,{ 0, 0, 0
},{ -1, -2, -1 } };
            for (int i = 0; i < height; i++) {
                for (int j = 0; j < width; j++) {
                    double h = 0, v = 0;
                    for (int k = -R; k <= R; k++) {
                        for (int m = -R; m <= R; m++) {
                            int x = i + k;
                            int y = j + m;
                            if (x < 0) x = 0;
                            if (x > (height - 1)) x = height -
1;

                            if (y < 0) y = 0;
                            if (y > (width - 1)) y = width - 1;
                            double cur = ycbcr[x][y].Y;
                            h += cur * mask_h[k + 1][m + 1];
                            v += cur * mask_v[k + 1][m + 1];
                        }
                    }
                    Gh[i][j] = h;
                    Gv[i][j] = v;
                    double I = sqrt(pow(h, 2) + pow(v, 2));
                    if (I > thr) I = 255; // если детктирован как
контур, то белый
                    else I = 0;
                    teta[i][j] = atan2(v, h);
                    new_ycbcr[i][j].Y = I;
                }
            }
        }
        void get_image(const char* filename) {
            FILE* file;
            file = fopen(filename, "wb");
            write_bmp_ycbcr(file, new_ycbcr, bfh, bih, height,
width);
            fclose(file);
        }

        void gradient(const char* filename) {
            RGB blue = { 0, 0, 255 };
```

```
            RGB green = { 0, 255, 0 };
            RGB red = { 255, 0, 0 };
            RGB white = { 255, 255, 255 };
            for (size_t i = 0; i < height; i++) {
                 for (size_t j = 0; j < width; j++) {
                      if (Gh[i][j] > 0 && Gv[i][j] > 0)
                           new_rgb[i][j] = red;
                      if (Gh[i][j] < 0 && Gv[i][j] > 0)
                           new_rgb[i][j] = green;
                      if (Gh[i][j] < 0 && Gv[i][j] < 0)
                           new_rgb[i][j] = blue;
                      if (Gh[i][j] > 0 && Gv[i][j] < 0)
                           new_rgb[i][j] = white;
                 }
            }
            FILE* file;
            file = fopen(filename, "wb");
            write_bmp(file, new_rgb, bfh, bih, height, width);
            fclose(file);
      }
};
```

AnchorPoints.h

```
#include <iostream>
#include <fstream>
#include <vector>
#include "bmp.h"

class AnchorPoints {
private:
      YCbCr** ycbcr;
      int height;
      int width;
      BITMAPFILEHEADER* bfh;
      BITMAPINFOHEADER* bih;
      YCbCr** new_ycbcr;
public:

      AnchorPoints(YCbCr** y, int h, int w, BITMAPFILEHEADER* bf,
BITMAPINFOHEADER* bi) {
            ycbcr = y;
            height = h;
            width = w;
            bfh = bf;
            bih = bi;

            new_ycbcr = new YCbCr * [height];
            for (int i = 0; i < height; i++)
                 new_ycbcr[i] = new YCbCr[width];

            for (int i = 0; i < height; i++) {
                 for (int j = 0; j < width; j++) {
```

```
                    new_ycbcr[i][j].Y = ycbcr[i][j].Y;
                }
            }

        }
        void get_image(const char* filename) {
            FILE* file;
            file = fopen(filename, "wb");
            write_bmp_ycbcr(file, new_ycbcr, bfh, bih, height,
width);
            fclose(file);
        }

        double clipping(double value) {
            if (value > 255.0) {
                value = 255.0;
            }
            if (value < 0.0) {
                value = 0;
            }
            return round(value);
        }

        void blackout(double val) {
            for (int i = 0; i < height; i++) {
                for (int j = 0; j < width; j++) {
                    new_ycbcr[i][j].Y = clipping(ycbcr[i][j].Y -
val);
                }
            }
        }

        void lightening(double val) {
            for (int i = 0; i < height; i++) {
                for (int j = 0; j < width; j++) {
                    new_ycbcr[i][j].Y = clipping(ycbcr[i][j].Y +
val);
                }
            }
        }

        unsigned char linear_inter(double x0, double y0, double x1,
double y1, double x) {
            double res = 0;
            res = y0 + (x - x0) * (y1 - y0) / (x1 - x0);
            return (unsigned char)round(res);
        }


        void anchor_points(unsigned char x0, unsigned char y0,
unsigned char x1, unsigned char y1) {
            const unsigned char R = 0;
            const unsigned char S = 255;
```

```
            for (size_t i = 0; i < height; i++) {
                for (size_t j = 0; j < width; j++) {
                    unsigned char res = new_ycbcr[i][j].Y;
                    if (res == x0) res = y0;
                    else if (res == x1) res = y1;
                    else if (res < x0 && res > R) res =
linear_inter(R, R, x0, y0,
                        res);
                    else if (res > x0 && res < x1) res =
linear_inter(x0, y0, x1, y1,
                        res);
                    else if (res > x1 && res < S) res =
linear_inter(x1, y1, S, S,
                        res);
                    new_ycbcr[i][j].Y = res;
                }
            }
        }
        void get_freq_original(const char* filename) {
            get_frequency(ycbcr, filename);
        }

        void get_freq_new(const char* filename) {
            get_frequency(new_ycbcr, filename);
        }
        void get_frequency(YCbCr** y, const char* filename) {
            ofstream out;
            out.open(filename);
            for (int i = 0; i < height; i++) {
                for (int j = 0; j < width; j++) {
                    out << y[i][j].Y << endl;
                }
            }
            out.close();
        }
};
```

GammaTransform.h

```
#include <iostream>
#include <fstream>
#include <vector>
#include "bmp.h"

class GammaTransform {
private:
        YCbCr** ycbcr;
        int height;
        int width;
        BITMAPFILEHEADER* bfh;
        BITMAPINFOHEADER* bih;
        YCbCr** new_ycbcr;
public:
```

```
     GammaTransform(YCbCr** y, int h, int w, BITMAPFILEHEADER*
bf, BITMAPINFOHEADER* bi) {
          ycbcr = y;
          height = h;
          width = w;
          bfh = bf;
          bih = bi;

          new_ycbcr = new YCbCr * [height];
          for (int i = 0; i < height; i++)
               new_ycbcr[i] = new YCbCr[width];

          for (int i = 0; i < height; i++) {
               for (int j = 0; j < width; j++) {
                    new_ycbcr[i][j].Y = ycbcr[i][j].Y;
               }
          }

     }
     void get_image(const char* filename) {
          FILE* file;
          file = fopen(filename, "wb");
          write_bmp_ycbcr(file, new_ycbcr, bfh, bih, height,
width);
          fclose(file);
     }
     double clipping(double value) {
          if (value > 255.0) {
               value = 255.0;
          }
          if (value < 0.0) {
               value = 0;
          }
          return round(value);
     }
     void gamma_transform(double c, double gamma) {
          for (int i = 0; i < height; i++) {
               for (int j = 0; j < width; j++) {
                    double tmp = new_ycbcr[i][j].Y / 255;
                    tmp = c * pow(tmp, gamma);
                    new_ycbcr[i][j].Y = clipping(tmp * 255);
               }
          }
     }

     void blackout(double val) {
          for (int i = 0; i < height; i++) {
               for (int j = 0; j < width; j++) {
                    new_ycbcr[i][j].Y = clipping(ycbcr[i][j].Y -
val);
               }
          }
     }
```

```
	void lightening(double val) {
		for (int i = 0; i < height; i++) {
			for (int j = 0; j < width; j++) {
				new_ycbcr[i][j].Y = clipping(ycbcr[i][j].Y +
val);
			}
		}
	}
	void get_freq_original(const char* filename) {
		get_frequency(ycbcr, filename);
	}

	void get_freq_new(const char* filename) {
		get_frequency(new_ycbcr, filename);
	}
	void get_frequency(YCbCr** y, const char* filename) {
		ofstream out;
		out.open(filename);
		for (int i = 0; i < height; i++) {
			for (int j = 0; j < width; j++) {
				out << y[i][j].Y << endl;
			}
		}
		out.close();
	}

};
```

HistogramEqualization.h

```
#include <iostream>
#include <fstream>
#include <vector>
#include "bmp.h"

class HistogramEqualization {
private:
	YCbCr** ycbcr;
	int height;
	int width;
	BITMAPFILEHEADER* bfh;
	BITMAPINFOHEADER* bih;
	YCbCr** new_ycbcr;
	int* freq = new int[256];
public:
	HistogramEqualization(YCbCr** y, int h, int w,
BITMAPFILEHEADER* bf, BITMAPINFOHEADER* bi) {
		ycbcr = y;
		height = h;
		width = w;
		bfh = bf;
```

```
            bih = bi;

            new_ycbcr = new YCbCr * [height];
            for (int i = 0; i < height; i++)
                  new_ycbcr[i] = new YCbCr[width];

            for (int i = 0; i < height; i++) {
                  for (int j = 0; j < width; j++) {
                        new_ycbcr[i][j].Y = ycbcr[i][j].Y;
                  }
            }

            for (int i = 0; i < 256; i++)
                  freq[i] = 0;

      }
      void get_image(const char* filename) {
            FILE* file;
            file = fopen(filename, "wb");
            write_bmp_ycbcr(file, new_ycbcr, bfh, bih, height,
width);
            fclose(file);
      }
      double clipping(double value) {
            if (value > 255.0) {
                  value = 255.0;
            }
            if (value < 0.0) {
                  value = 0;
            }
            return round(value);
      }
      void histogram_equalization() {
            for (size_t i = 0; i < height; i++) {
                  for (size_t j = 0; j < width; j++) {
                        freq[(int)new_ycbcr[i][j].Y]++;
                  }
            }

            unsigned char* new_freq = new unsigned char[256];
            for (size_t i = 0; i < 256; i++) {
                  double tmp = 0;
                  for (size_t j = 0; j <= i; j++) {
                        tmp += freq[j];
                  }
                  tmp = tmp * 255 / (height * width);
                  new_freq[i] = (clipping(tmp));
            }
            for (size_t i = 0; i < height; i++) {
                  for (size_t j = 0; j < width; j++) {
                        new_ycbcr[i][j].Y =
(new_freq[(int)new_ycbcr[i][j].Y]);
                  }
```

```
            }
        }

        void blackout(double val) {
            for (int i = 0; i < height; i++) {
                for (int j = 0; j < width; j++) {
                    new_ycbcr[i][j].Y = clipping(ycbcr[i][j].Y -
val);
                }
            }
        }

        void lightening(double val) {
            for (int i = 0; i < height; i++) {
                for (int j = 0; j < width; j++) {
                    new_ycbcr[i][j].Y = clipping(ycbcr[i][j].Y +
val);
                }
            }
        }
        void get_freq_original(const char* filename) {
            get_frequency(ycbcr, filename);
        }

        void get_freq_new(const char* filename) {
            get_frequency(new_ycbcr, filename);
        }
        void get_frequency(YCbCr** y, const char* filename) {
            ofstream out;
            out.open(filename);
            for (int i = 0; i < height; i++) {
                for (int j = 0; j < width; j++) {
                    out << y[i][j].Y << endl;
                }
            }
            out.close();
        }

};
```

GradationTransform.h

```
#include <iostream>
#include <fstream>
#include <vector>
#include "bmp.h"

class GradationTransform {
private:
    YCbCr** ycbcr;
    int height;
    int width;
    BITMAPFILEHEADER* bfh;
    BITMAPINFOHEADER* bih;
```

```
	YCbCr** new_ycbcr;
	int* freq = new int[256];
public:
	GradationTransform(YCbCr** y, int h, int w,
BITMAPFILEHEADER* bf, BITMAPINFOHEADER* bi) {
		ycbcr = y;
		height = h;
		width = w;
		bfh = bf;
		bih = bi;

		new_ycbcr = new YCbCr * [height];
		for (int i = 0; i < height; i++)
			new_ycbcr[i] = new YCbCr[width];

	}
	void get_image(const char* filename) {
		FILE* file;
		file = fopen(filename, "wb");
		write_bmp_ycbcr(file, new_ycbcr, bfh, bih, height,
width);
		fclose(file);
	}
	void gradation_transform(int T) {
		for (int i = 0; i < height; i++) {
			for (int j = 0; j < width; j++) {
				if (ycbcr[i][j].Y > T)
					new_ycbcr[i][j].Y = 0;
				else
					new_ycbcr[i][j].Y = 255;
			}
		}
	}

};
```

Bmp.h

```
#ifndef bmp
#define bmp
#include <iostream>
using namespace std;

struct BITMAPFILEHEADER {
	short bfType;
	int bfSize;
	short bfReserved1;
	short bfOffBits;;
	int bfReserved2;
};

struct BITMAPINFOHEADER {
	int biSize;
```

```
	int biWidth;
	int biHeight;
	short int biPlanes;
	short int biBitCount;
	int biCompression;
	int biSizeImage;
	int biXPelsPerMeter;
	int biYPelsPerMeter;
	int biClrUsed;
	int biClrImportant;
};

struct RGB {
	unsigned char B;
	unsigned char G;
	unsigned char R;
};

struct YCbCr {
	double Cr;
	double Cb;
	double Y;
};

RGB** read_bmp(FILE* f, BITMAPFILEHEADER* bfh, BITMAPINFOHEADER*
bih)
{
	int k = 0;
	k = fread(bfh, sizeof(*bfh) - 2, 1, f);
	if (k == 0)
	{
		cout << "Error";
		return 0;
	}

	k = fread(bih, sizeof(*bih), 1, f);
	if (k == NULL)
	{
		cout << "Error";
		return 0;
	}
	int a = abs(bih->biHeight);
	int b = abs(bih->biWidth);
	RGB** rgb = new RGB * [a];
	for (int i = 0; i < a; i++)
	{
		rgb[i] = new RGB[b];
	}
	int pad = 4 - (b * 3) % 4;
	for (int i = 0; i < a; i++)
	{
		fread(rgb[i], sizeof(RGB), b, f);
		if (pad != 4)
```

```
            {
                fseek(f, pad, SEEK_CUR);
            }
        }
        return rgb;
}

void write_bmp(FILE* f, RGB** rgb, BITMAPFILEHEADER* bfh,
BITMAPINFOHEADER* bih, int height, int width)
{
        bih->biHeight = height;
        bih->biWidth = width;
        fwrite(bfh, sizeof(*bfh) - 2, 1, f);
        fwrite(bih, sizeof(*bih), 1, f);
        int pad = 4 - ((width) * 3) % 4;
        char buf = 0;
        for (int i = 0; i < height; i++)
        {
            fwrite((rgb[i]), sizeof(RGB), width, f);
            if (pad != 4)
            {
                fwrite(&buf, 1, pad, f);
            }
        }
}

void write_bmp_ycbcr(FILE* f, YCbCr** ycbcr, BITMAPFILEHEADER*
bfh, BITMAPINFOHEADER* bih, int height, int width)
{
        RGB** rgb = new RGB * [height];
        for (int i = 0; i < height; i++) {
            rgb[i] = new RGB[width];
            for (int j = 0; j < width; j++) {
                rgb[i][j].B = ycbcr[i][j].Y;
                rgb[i][j].R = ycbcr[i][j].Y;
                rgb[i][j].G = ycbcr[i][j].Y;
            }
        }
        bih->biHeight = height;
        bih->biWidth = width;
        fwrite(bfh, sizeof(*bfh) - 2, 1, f);
        fwrite(bih, sizeof(*bih), 1, f);
        int pad = 4 - ((width) * 3) % 4;
        char buf = 0;
        for (int i = 0; i < height; i++)
        {
            fwrite((rgb[i]), sizeof(RGB), width, f);
            if (pad != 4)
            {
                fwrite(&buf, 1, pad, f);
            }
        }
}
```

```
#endif bmp
```