

Цель работы:

Получение практических навыков использования алгоритмов улучшения качества изображений, применяемых в пространственной области, на примерах методов шумоподавления, выделения контуров и градационных преобразований (для упрощения анализа рассматриваются только однокомпонентные изображения).

1. Реализация модели аддитивного шума:

При выполнении пунктов лабораторной работы будет проводиться работа с яркостной составляющей изображения Y-компонентой, значение в пикселе которой вычисляется по следующей формуле:

$$Y = 0.299R + 0.587G + 0.114B,$$

где R, G и B – значения цветовых компонент формата RGB для соответствующего пикселя.

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

$$I'_{x,y} = \text{Clipp}(I_{x,y} + 0.255N_{x,y}),$$

где $N_{x,y}$ – значение шума на позиции пикселя с координатами (x, y), а Clipp – операция клиппирования, соответствующая следующей формуле:

$$\text{Clipp}(I_{x,y}, I^{\min}, I^{\max}) = \begin{cases} I^{\min}, & I_{x,y} < I^{\min} \\ I^{\max}, & I_{x,y} > I^{\max} \\ I_{x,y}, & \text{иначе} \end{cases}$$

Значение шума $N_{x,y}$ можно сгенерировать, воспользовавшись преобразованием Бокса-Мюллера.

Пусть r и φ – независимые случайные величины, равномерно распределенные на интервале $(0,1]$ и z_0 и z_1 определены как

$$\begin{aligned} z_0 &= \cos(2\pi\varphi)\sqrt{-2\ln(r)} \\ z_1 &= \sin(2\pi\varphi)\sqrt{-2\ln(r)} \end{aligned}$$

Тогда z_0 и z_1 – независимы и имеют нормальное распределение с математическим ожиданием 0 и дисперсией 1.

Чтобы перейти к общему нормальному распределению воспользуемся формулой:

$$\varepsilon = \mu + \sigma z,$$

где μ -математическое ожидание, σ -стандартное отклонение, а ε - случайная величина с Гауссовским распределением $N(\mu, \sigma^2)$.

Результаты формирования аддитивного шума:



Рис. 1. Изображение с аддитивным шумом при $\sigma = 30$.

Из полученного изображения видно, что при значении $\sigma = 30$ можно заметить шум на картинке.

2. Реализация модели импульсного шума:

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

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

где p_a и p_b – параметры системы.

Результаты формирования импульсного шума:



Рис. 2. Изображение с импульсным шумом при $p_a = p_b = 0.25$.

Как видно на изображении импульсный шум более выражен чем аддитивный, так как зашумленные пиксели принимают крайние значения 0 или 255.

3. Построение графиков PSNR для аддитивного и импульсного шумов:

PSNR можно вычислить по формуле:

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

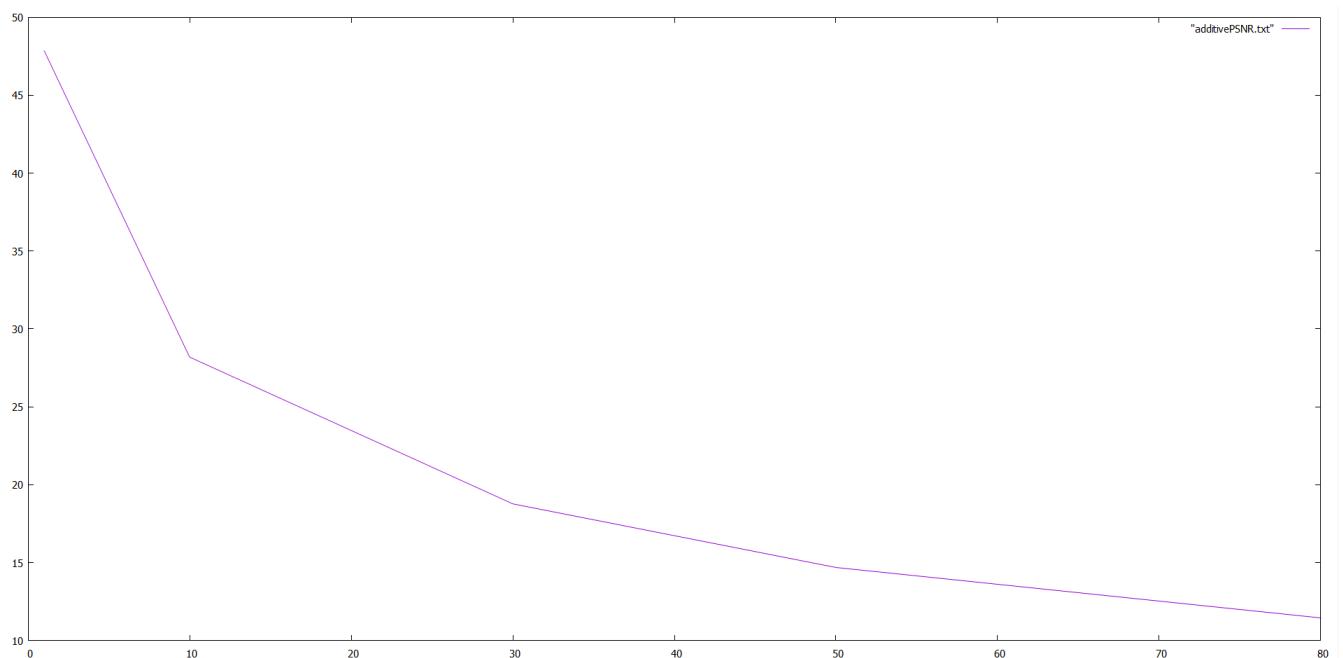


Рис. 3. График зависимости PSNR от σ .

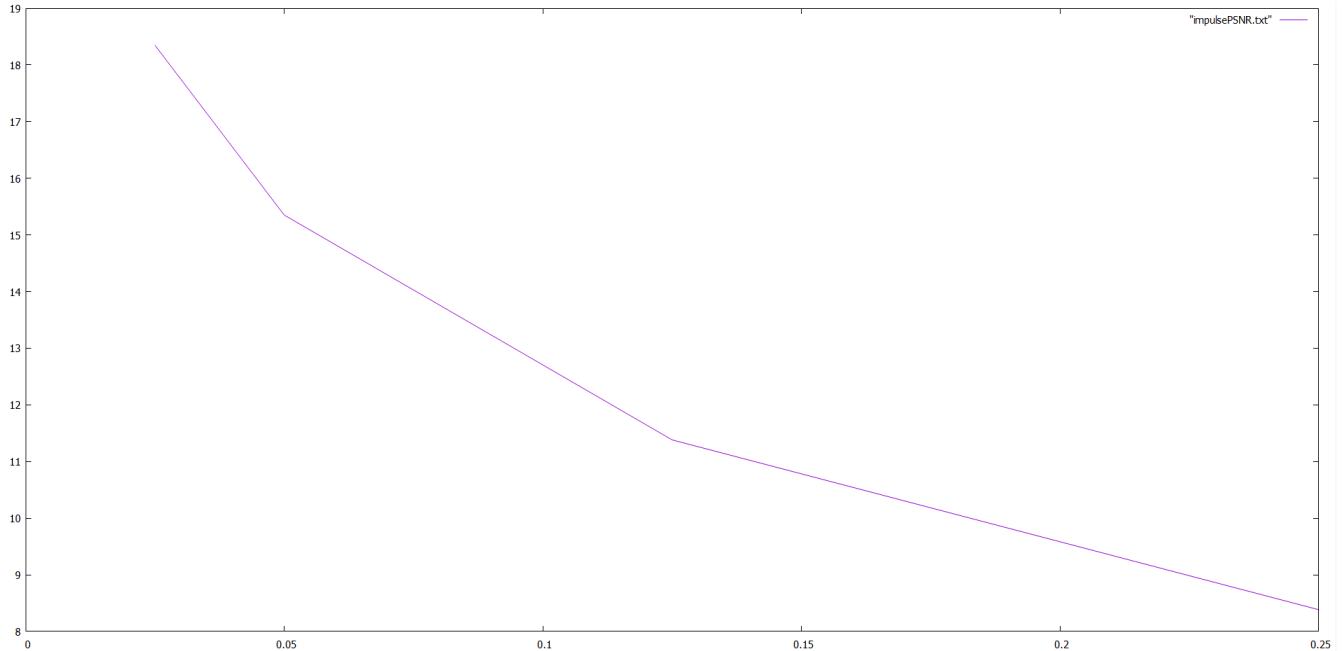


Рис. 4. График зависимости PSNR от $p_a = p_b$.

По данным графикам видно какие искажения вносят импульсный и аддитивные шумы. Импульсный шум даже при малых параметрах очень ухудшает качество картинки, так как максимальное значение PSNR равно 18, в то время как у аддитивного шума максимальный PSNR равен 48. Это объясняется тем, что импульсный шум изменяет компоненту либо в 0, либо в 255. Аддитивный шум на малых размерах изменяет компоненту на небольшую величину, тем самым PSNR падает не так критично.

4. Обработка изображений с аддитивным шумом:

В процессе выполнения фильтрации входного (зашумленного) изображения $I_{[H \times W]}$ формируется новое изображение $I'_{[H \times W]}$ с теми же размерами. Каждый пиксель $I'_{y,x}$ формируется в результате применения некоторого оператора к пикселью $I_{y,x}$ и подмножеству соседних с ним пикселей, образующих так называемую (апертуру) фильтра. Применяемый оператор может быть как линейным, так и нелинейным. При линейной фильтрации используется следующее правило для расчета значений отфильтрованных пикселей:

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

где R – радиус фильтра, определяющий апертуру, $w_{k,m}$ – весовые коэффициенты фильтра, Z – коэффициент нормировки, определяемый по формуле:

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

Следует отметить, что использование термина “радиус” является общепринятым, хотя в реальности апертура имеет форму квадрата со стороной $2R+1$. На краях массива I , где апертура выходит за границы изображения, недостающее значение, как правило, заменяется интенсивностью ближайшего по Евклидову расстоянию пикселя.

4.1. Реализация метода скользящего среднего:

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

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

Результаты применения фильтра:



Рис. 5. Изображение с аддитивным шумом при $\sigma = 30$.



Рис. 6. Изображение после применения фильтра при $R = 1$.

Как видно из приведенных рисунков – отфильтрованное изображение получилось немного размытым, но шум удалось “сгладить”. Значение PSNR увеличилось с 19.0006 до 21.6939.

4.2. Подбор оптимального размера окна R:

Определим значение R для $\sigma = 1, 10, 30, 50$ и 80 такое, что значение PSNR будет максимальным.

```

SIGMA = 1
R = 1 PSNR = 33.8364
R = 2 PSNR = 29.926
R = 3 PSNR = 28.2642
R = 4 PSNR = 27.3522
R = 5 PSNR = 26.7749
SIGMA = 10
R = 1 PSNR = 32.3196
R = 2 PSNR = 29.6574
R = 3 PSNR = 28.1667
R = 4 PSNR = 27.3044
R = 5 PSNR = 26.7473
SIGMA = 30
R = 1 PSNR = 27.1449
R = 2 PSNR = 27.9887
R = 3 PSNR = 27.4459
R = 4 PSNR = 26.9028
R = 5 PSNR = 26.4867
SIGMA = 50
R = 1 PSNR = 23.6008
R = 2 PSNR = 25.9017
R = 3 PSNR = 26.2485
R = 4 PSNR = 26.1259
R = 5 PSNR = 25.9119
SIGMA = 80
R = 1 PSNR = 20.3839
R = 2 PSNR = 23.2106
R = 3 PSNR = 24.1269
R = 4 PSNR = 24.379
R = 5 PSNR = 24.397

```

Рис. 7. Результат работы программы.

Таким образом, максимальное значение PSNR для всех значений σ достигается при размерах окна R=1. С увеличением значения сигмы шума уменьшается и получаемый PSNR.

4.3. Реализация метода Гауссовой фильтрации:

Поскольку на фотoreалистичном изображении соседние пиксели сильно коррелируют, значения весовых коэффициентов как правило, рассчитываются на базе некоторой функции от расстояния до центральной позиции $I_{i,j}$, значения которой убывают по мере удаления позиции коэффициента от центральной. Однако такой подход имеет негативный эффект с точки зрения визуального восприятия. В окрестности контуров (резких перепадов) применение линейных фильтров приводит к “размытию” контура. Поэтому данный класс фильтров также называют фильтрами размытия. Функция расстояния для расчёта 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 σ – параметр фильтра, определяющий скорость убывания коэффициентов по мере удаления от позиции центрального пикселя.

Результат применения фильтра:



Рис. 8. Изображение с аддитивным шумом при $\sigma = 30$.



Рис. 9. Изображение после применения фильтра при $R = 3$, $\sigma = 1$.

После фильтрации так же присутствует эффект размытия. Значение PSNR увеличилось с 18.9862 до 22.0362.

5.5. Построение графиков PSNR(σ):

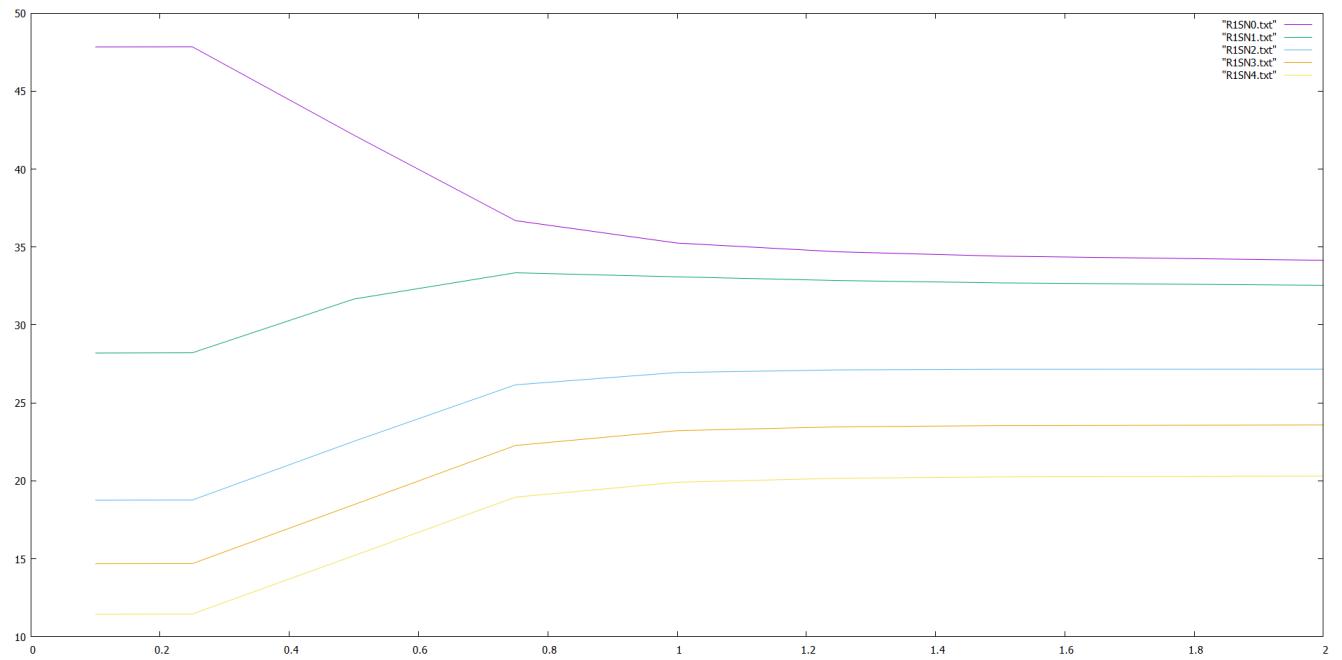


Рис. 10. График $PSNR(\sigma)$ при $R=1$.

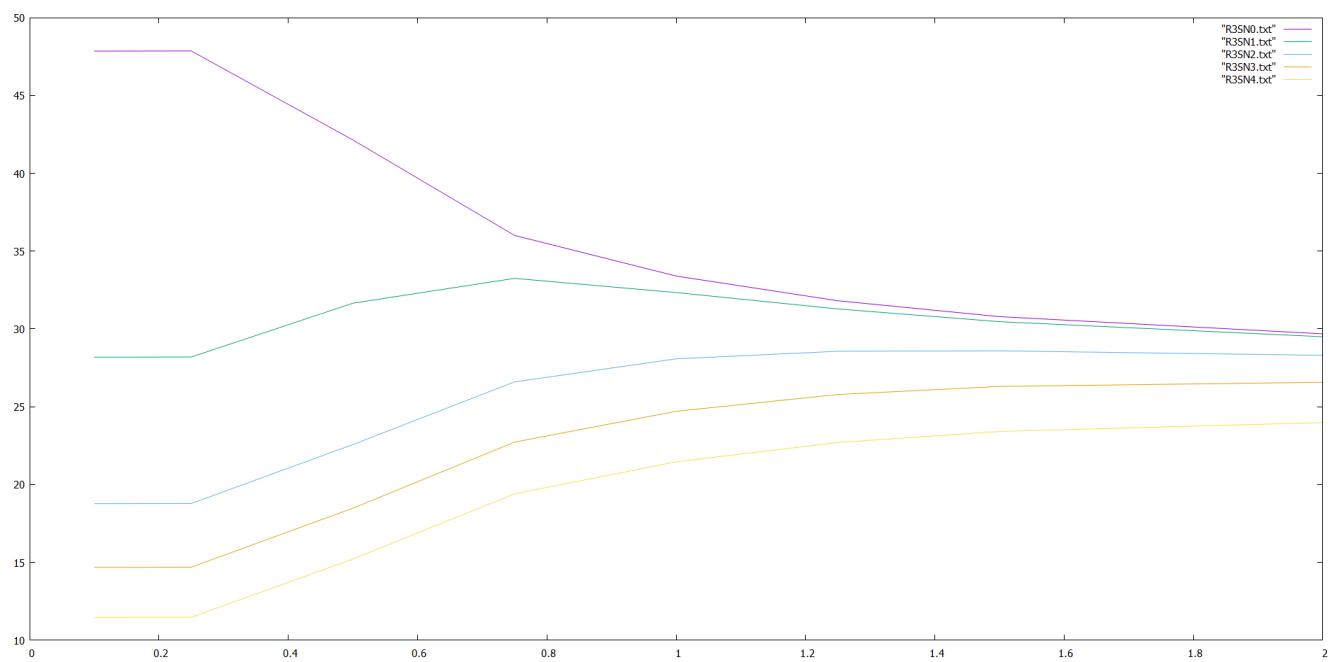


Рис. 11. График $PSNR(\sigma)$ при $R=3$.

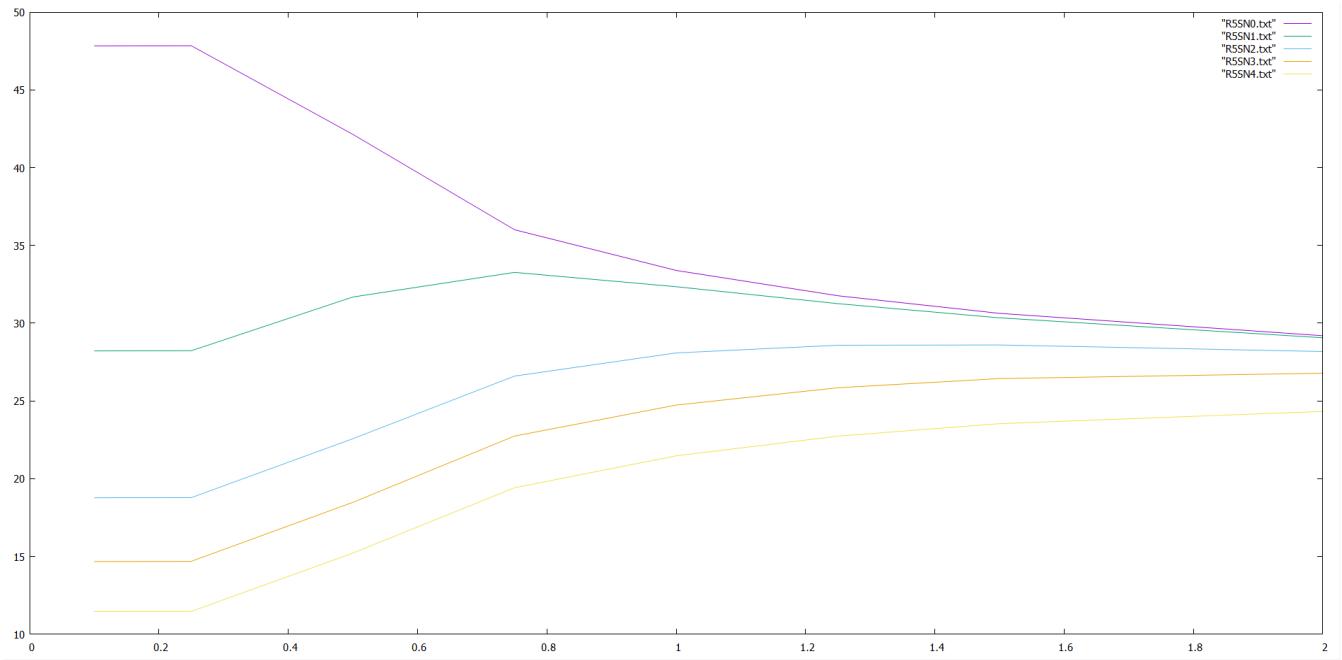


Рис. 12. График $PSNR(\sigma)$ при $R=5$.

Из полученных графиков можно сделать вывод о том, что при значениях $\sigma_{\text{фильтра}} > 1$ значения PSNR при всех рассматриваемых шумах и значениях R перестают увеличиваться, а при малых шумах начинает уже уменьшаться.

4.4. Подбор оптимальных параметров:

Исходя из графиков, полученных в пункте 5.4, определим оптимальные параметры для каждого значения шума (R и σ), при которых значение PSNR будет максимальным:

- При $\sigma_{\text{шума}} = 1$: R=1, $\sigma_{\text{фильтра}} = 0.25$, PSNR = 47.8443
- При $\sigma_{\text{шума}} = 10$: R=1, $\sigma_{\text{фильтра}} = 0.5$, PSNR = 31.6627
- При $\sigma_{\text{шума}} = 30$: R=1, $\sigma_{\text{фильтра}} = 0.75$, PSNR = 26.1587
- При $\sigma_{\text{шума}} = 50$: R=3, $\sigma_{\text{фильтра}} = 0.75$, PSNR = 24.7076
- При $\sigma_{\text{шума}} = 80$: R=5, $\sigma_{\text{фильтра}} = 2$, PSNR = 24.3262

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

5.7. Реализация метода Медианной фильтрации:

Медианный фильтр является фильтром, основанным на порядковых статистиках – частном случае нелинейное фильтрации. 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.

Взятие медианы по массиву A происходит следующим образом:

- Производится сортировка элементов одномерного массива A (по убыванию или возрастанию);
- Сохранить в $I'_{i,j}$ элемент, находящийся в середине отсортированного массива A. Для апертуры радиуса R необходимо взять из отсортированного массива элемент с индексом $\left(\left\lfloor \frac{(2R+1)^2}{2} \right\rfloor + 1\right)$.

Результат применения фильтра:



Рис. 13. Изображение с аддитивным шумом при $\sigma = 30$.



Рис. 14. Изображение после применения фильтра при $R = 2$.

На выходе фильтра получение очень нечеткое изображение. Шум на изображении удалось сгладить, а контуры размыты. Значение PSNR увеличилось с 18.7613 до 19.7304. Визуально воспринимать отфильтрованное изображение сложнее, чем зашумленное.

5.8. Подбор оптимального параметра R:

Определим значение R для $\sigma = 1, 10, 30, 50$ и 80 такое, что значение PSNR будет максимальным.

```
SIGMA = 1
R = 1 PSNR = 36.5243
R = 2 PSNR = 31.544
R = 3 PSNR = 29.3776
R = 4 PSNR = 28.3626
R = 5 PSNR = 27.765
SIGMA = 10
R = 1 PSNR = 32.8692
R = 2 PSNR = 30.7721
R = 3 PSNR = 29.0514
R = 4 PSNR = 28.1769
R = 5 PSNR = 27.6253
SIGMA = 30
R = 1 PSNR = 25.7861
R = 2 PSNR = 27.6917
R = 3 PSNR = 27.5948
R = 4 PSNR = 27.2464
R = 5 PSNR = 26.9183
SIGMA = 50
R = 1 PSNR = 21.7151
R = 2 PSNR = 24.7474
R = 3 PSNR = 25.7469
R = 4 PSNR = 25.9957
R = 5 PSNR = 25.989
SIGMA = 80
R = 1 PSNR = 17.8971
R = 2 PSNR = 21.4559
R = 3 PSNR = 23.2556
R = 4 PSNR = 24.1662
R = 5 PSNR = 24.6181
```

Рис. 15. Результат работы программы.

Таким образом, при зашумленности $\sigma < 30$ PSNR принимает максимальное значение при $R=1$. В случае большой зашумленности, когда $\sigma = 80$, наибольшее значение достигается при $R=5$. С увеличением значения сигмы шума уменьшается и получаемый PSNR.

5.6, 5.9. Анализ PSNR при применении различных фильтров:

Сформируем изображения с различными аддитивными шумами при $\sigma = 1, 10, 30, 50, 80$. Проведем фильтрацию этого изображения различными фильтрами и проанализируем максимальное значение PSNR, полученные в результате работы каждого фильтра.

- При σ шума равной 1:



Рис. 16. Изображение с шумом $PSNR=47.85$.



Рис. 17. Изображение после фильтра Гаусса $PSNR=47.8612$.



Рис. 18. Изображение после фильтра методом скользящего среднего $PSNR= 33.8351$.



Рис. 19. Изображение после медианного фильтра $PSNR= 36.5262$.

- При σ шума равной 10:



Рис. 20. Изображение с шумом $PSNR=28.2005$.



Рис. 21. Изображение после фильтра Гаусса $PSNR=28.2121$.



Рис. 22. Изображение после фильтра методом скользящего среднего $PSNR= 32.3225$.



Рис. 23. Изображение после медианного фильтра $PSNR= 32.8278$.

- При σ шума равной 30:



Рис. 24. Изображение с шумом $PSNR=18.7683$.



Рис. 25. Изображение после фильтра Гаусса $PSNR=26.1584$.



Рис. 26. Изображение после фильтра методом скользящего среднего $PSNR= 27.1227$.



Рис. 27. Изображение после медианного фильтра $PSNR= 25.7853$.

- При σ шума равной 50:



Рис. 28. Изображение с шумом $PSNR= 14.6718$.



Рис. 29. Изображение после фильтра Гаусса $PSNR= 22.7385$.



Рис. 30. Изображение после фильтра методом скользящего среднего $PSNR= 23.5836$.



Рис. 31. Изображение после медианного фильтра $PSNR= 21.6988$.

- При σ шума равной 80:



Рис. 32. Изображение с шумом $PSNR= 11.4613$.



Рис. 33. Изображение после фильтра Гаусса $PSNR= 22.7602$.



Рис. 34. Изображение после фильтра методом скользящего среднего $PSNR= 20.3376$.



Рис. 35. Изображение после медианного фильтра $PSNR= 21.4365$.

Визуально можно заметить, что метод Гаусса и скользящего среднего эффективнее улучшают изображение, чем метод медианной фильтрации, который даже при слабом шуме заметно ухудшает изображение. В некоторых случаях значение $PSNR$ лучше, чем у фильтра Гаусса и

скользящего среднего, однако воспринимается данное изображение значительно хуже, чем даже зашумленное изображение.

4.9 Анализ PSNR после применения различных фильтров:

Построим графики зависимости $PSNR(\sigma)$ для $PSNR$ между исходным и зашумленным изображениями и максимальных $PSNR$ между исходным и отфильтрованными изображениями для каждого фильтра.

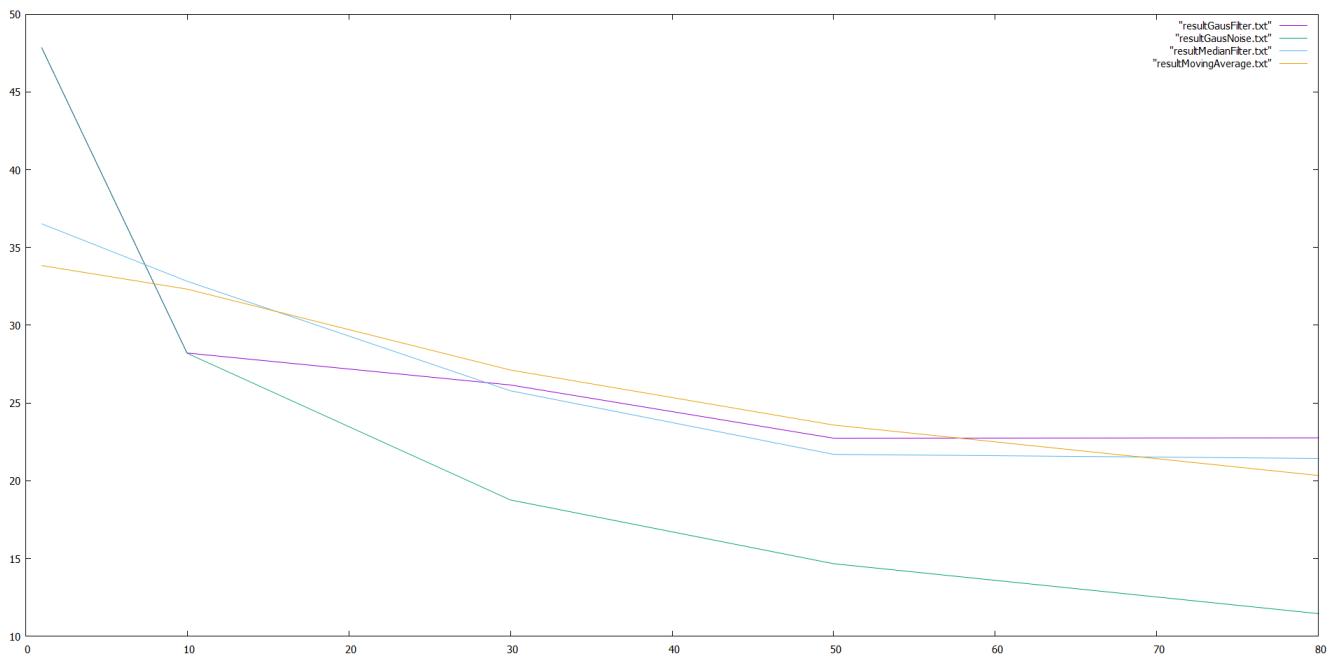


Рис. 36. Графики зависимости $PSNR$.

По графикам видно, что для изображений с $\sigma = 1$ лучшей фильтрацией является фильтр Гаусса. Для изображений с $\sigma = 10$ лучшей фильтрацией являются фильтры скользящего среднего и медианный. Для больших значений сигмы все фильтры дают примерно одинаковый результат, однако визуально для медианного фильтра изображения воспринимается гораздо хуже.

5. Обработка изображений с импульсным шумом:

5.1. Наложение шума на изображение:

Наложим на изображение импульсные шумы так, чтобы доли искаженных пикселей составляли: 5%, 10%, 25% и 50%.



Рис. 37. Изображение с импульсным шумом при $p_a = p_b = 0.025$.



Рис. 38. Изображение с импульсным шумом при $p_a = p_b = 0.05$.



Рис. 39. Изображение с импульсным шумом при $p_a = p_b = 0.125$.



Рис. 40. Изображение с импульсным шумом при $p_a = p_b = 0.25$.

6.2. Вычисление PSNR по зашумленным изображениям:

Определим значения PSNR для изображений, сформированных в предыдущем пункте. Результаты вычислений:

5% PSNR = 18.3094
10% PSNR = 15.3219
25% PSNR = 11.3839
50% PSNR = 8.36814

Рис. 41. Результат работы программы.

6.3. Применение медианной фильтрации: Результат работы:



Рис. 42. Отфильтрованное изображение с импульсным шумом при $p_a = p_b = 0.025$ при $R = 1$.



Рис. 43. Отфильтрованное изображение с импульсным шумом при $p_a = p_b = 0.05$ при $R = 1$.



Рис. 44. Отфильтрованное изображение с импульсным шумом при $p_a = p_b = 0.125$ при $R = 1$.



Рис. 45. Отфильтрованное изображение с импульсным шумом при $p_a = p_b = 0.25$ при $R = 3$.

По полученным изображениям 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.

6.4. Анализ PSNR разных вариантов фильтрации:

Построим зависимость $PSNR(R)$ для всех вариантов зашумленных изображений и проведем анализ полученных результатов.

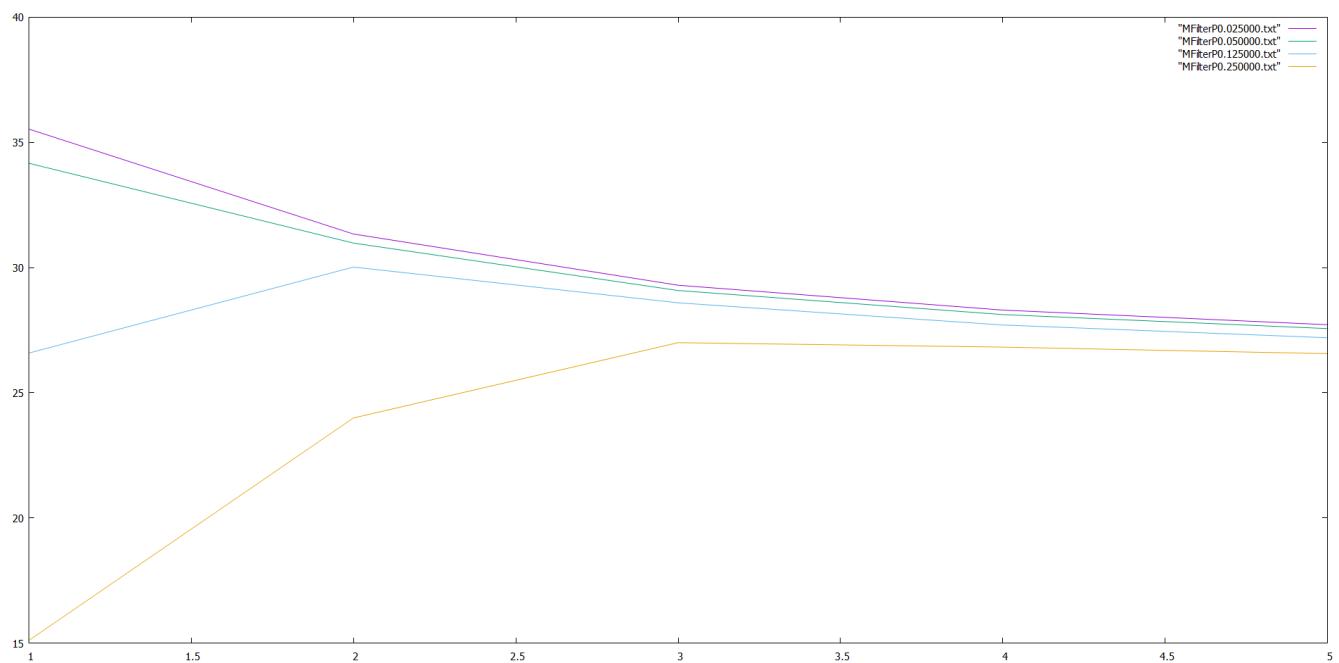


Рис. 46. Графики зависимости $PSNR(R)$.

Из полученных графиков видно, что при зашумлённости 5%, 10%, 25% наибольшие значения PSNR достигаются при размере окна R=1. При зашумленности 50% наибольшее значение PSNR достигается при R=3.

6. Реализация методов выделения контуров:

7.1,7.2. Применение оператора Лапласа и формирование изображения по отклику:

Необходимо применить оператор Лапласа $L(I)$ к изображению I и получить отклик I' , воспользовавшись формулой

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

где в качестве весовых коэффициентов $w_{k,m}$ будут использоваться элементы масок:

0	1	0
1	-4	1
0	1	0

0	-1	0
-1	4	-1
0	-1	0

Рис. 47. Маски.



Рис. 48. Отклик после применения первой маски.



Рис. 49. Отклик после применения второй маски.

По полученному отклику оператора Лапласа Γ' необходимо сформировать изображение. Значение пикселя на позиции (x, y) вычисляется по следующей формуле:

$$Clipp(I'_{x,y} + 128, 0, 255)$$

В итоге получаем:

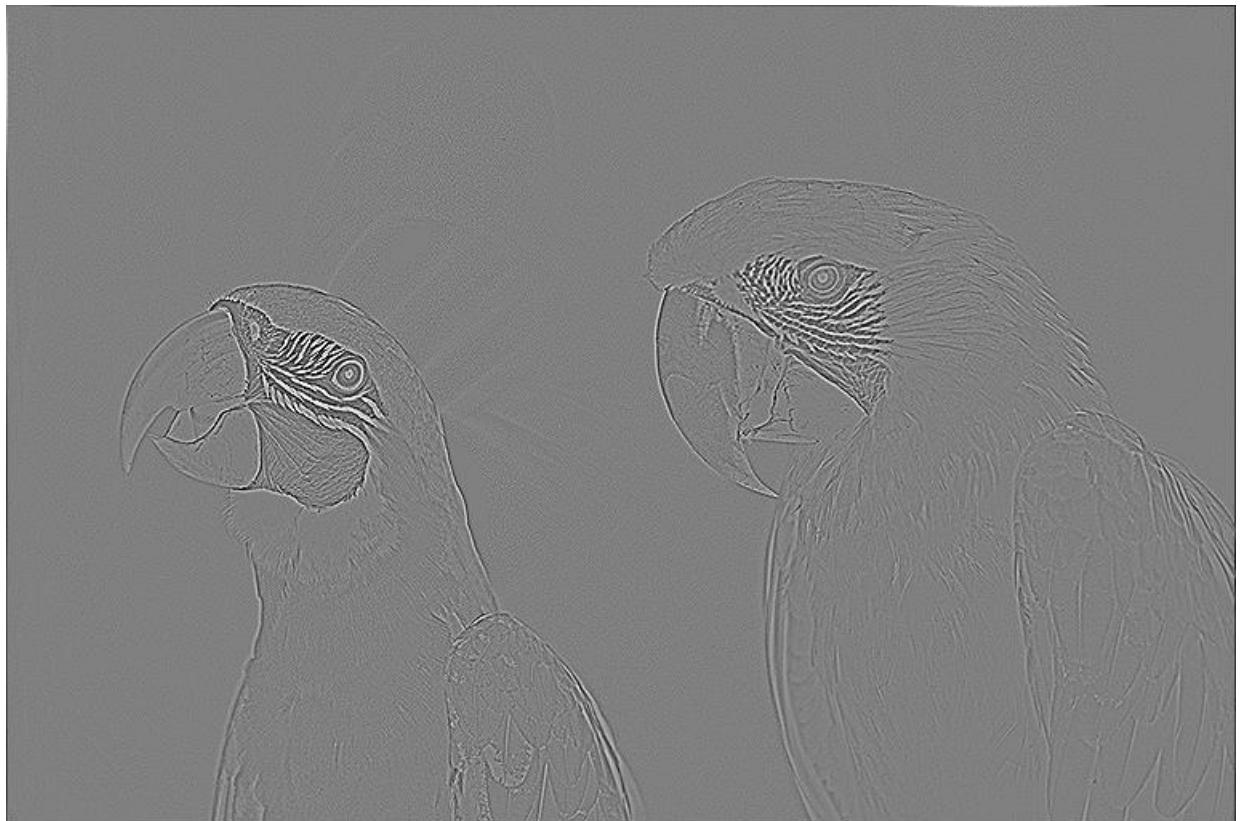


Рис. 50. Изображение после применения первой маски.

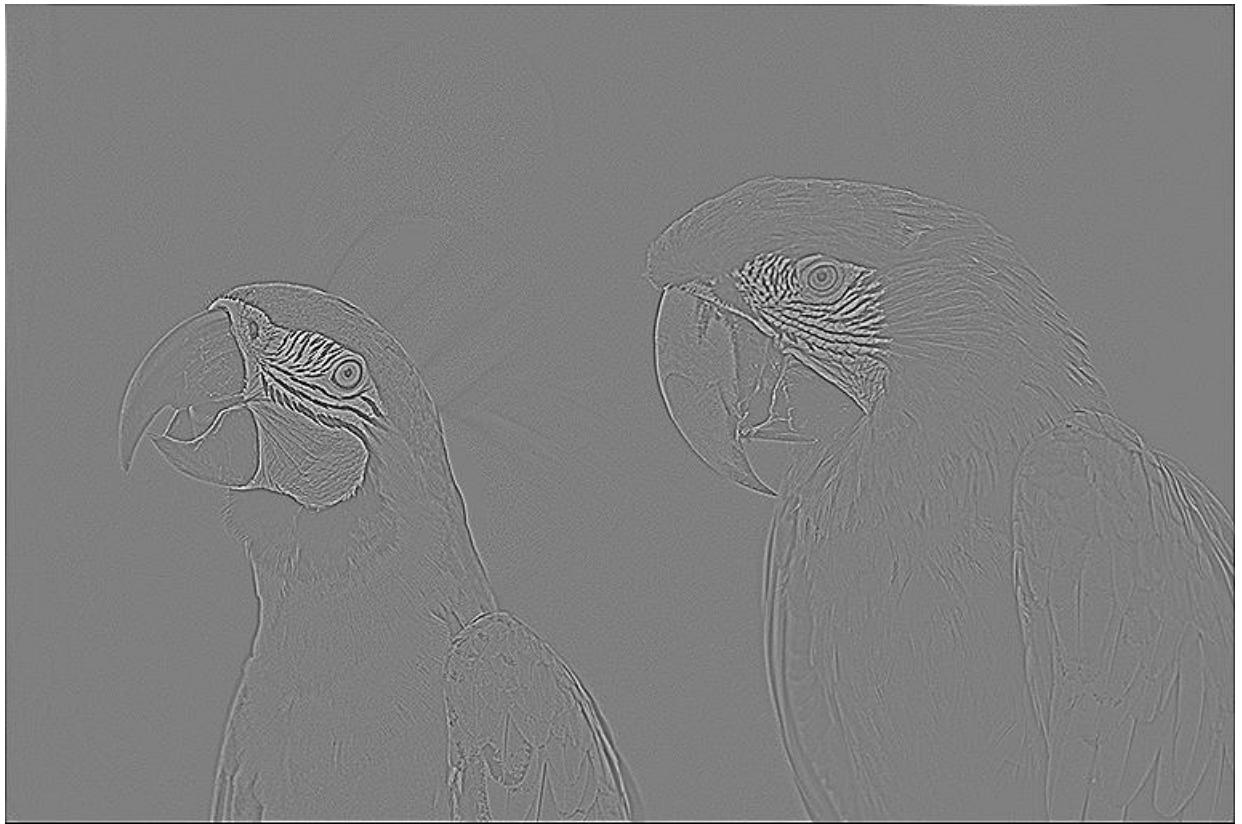


Рис. 51. Изображение после применения второй маски.

По полученным изображениям можно увидеть разницу в применении масок. Например, после использования первой маски белые контуры у попугаев, а после второй маски – черные.

7.3. Синтез изображения с усилением высоких частот:

Необходимость синтезировать изображение с усилением высоких частот, используя разность исходного изображения I и отклика оператора Лапласа I' , применив следующую маску:

$$\begin{array}{|c|c|c|} \hline 0 & -1 & 0 \\ \hline -1 & \alpha+4 & -1 \\ \hline 0 & -1 & 0 \\ \hline \end{array}$$

Рис. 52. Маска.

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

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

Полученный результат:



Рис. 53. Изображение до изменений.



Рис. 54. Синтезированное изображение на основе второго отклика.

Сравнивая исходное изображение и синтезированное, можно сделать вывод о том, что при прибавлении к исходному изображению отклик оператора Лапласа контуры объектов становятся четче видны.

7.4, 7.5. Синтез изображений с различными значениями параметра α :

Необходимо синтезировать изображения, применив маску, приведенную на рис. 52 для разных значений параметра α . Параметр α изменяется в интервале от 1 до 1.5 с шагом 0.1.

Результаты синтеза:



Рис. 55. Синтезированное изображение при $\alpha=1.0$.



Рис. 56. Синтезированное изображение при $\alpha=1.1$.



Рис. 57. Синтезированное изображение при $\alpha=1.2$.



Рис. 58. Синтезированное изображение при $\alpha=1.3$.



Рис. 59. Синтезированное изображение при $\alpha=1.4$.



Рис. 60. Синтезированное изображение при $\alpha=1.5$.

По полученным изображениям можно сделать вывод, что при увеличении α общая яркость изображения увеличивается, а резкость – уменьшается.

Синтезированное изображение совпадает с Γ' при $\alpha=1$. Докажем данное утверждение:

$$\begin{aligned} I''_{x,y} &= I'_{x,y} + I_{x,y} = (4I_{x,y} - I_{x-1,y} - I_{x+1,y} - I_{x,y-1} - I_{x,y+1}) + I_{x,y} \\ &= 5I_{x,y} - I_{x-1,y} - I_{x+1,y} - I_{x,y-1} - I_{x,y+1}, \end{aligned}$$

что соответствует маске на рисунке 52 при $\alpha=1$.

7.6. Средние значения яркости:

Необходимо рассчитать среднее значение яркости для каждого из изображений, полученных в предыдущем пункте. В итоге имеем:

Alpha = 1.0: 109.34
Alpha = 1.1: 129.861
Alpha = 1.2: 148.26
Alpha = 1.3: 164.352
Alpha = 1.4: 179.13
Alpha = 1.5: 192.65

Рис. 61. Результат работы программы.

6.1. Построение и анализ гистограмм:

Построим гистограмму для исходного изображения, а также гистограммы для изображений, полученных в пункте 7.4.

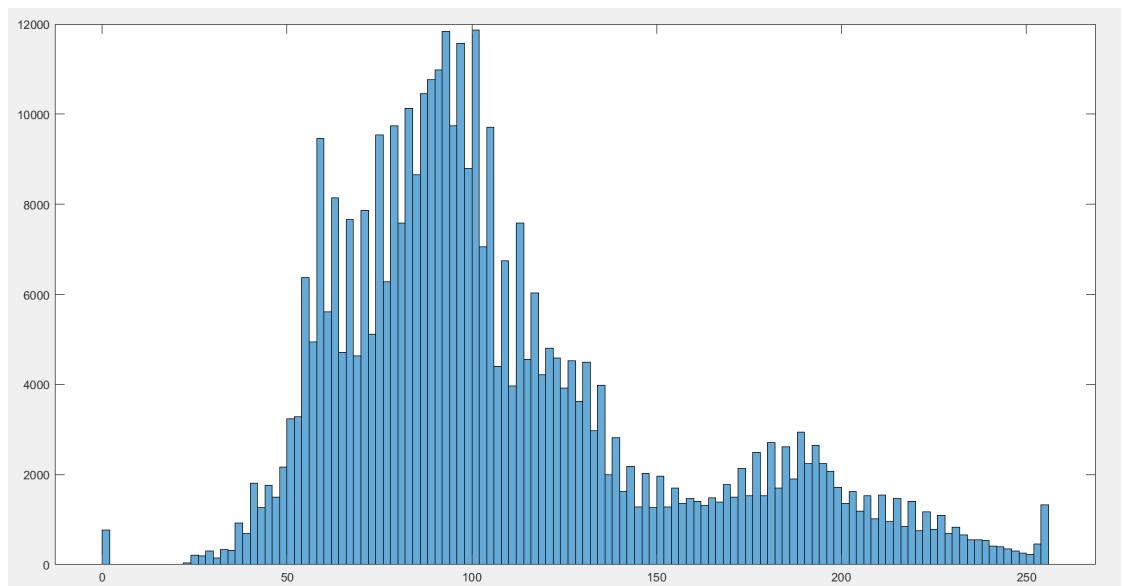


Рис. 62. Гистограмма исходного изображения.

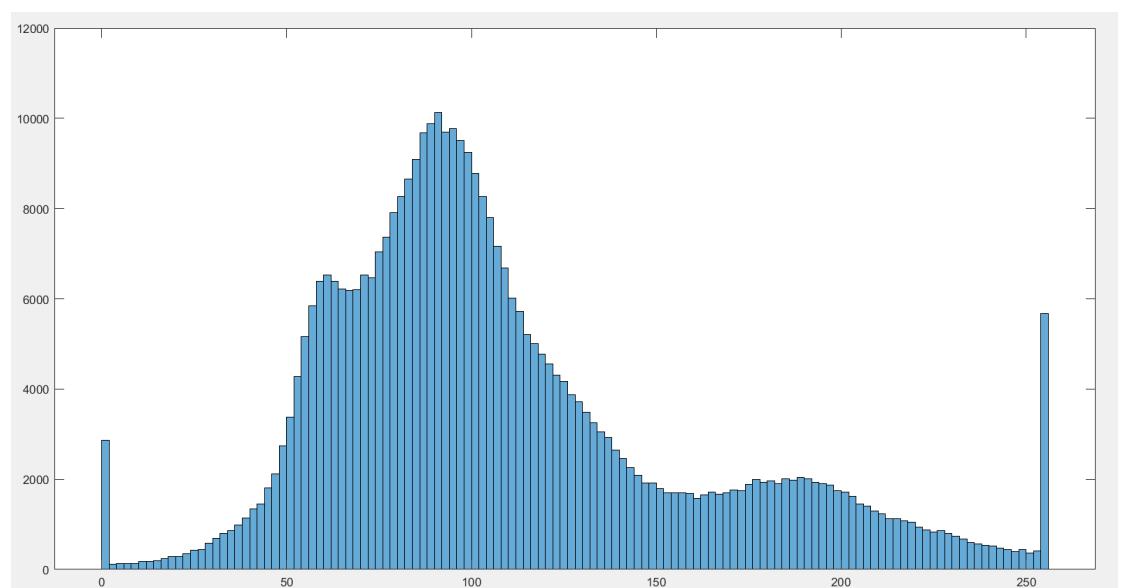


Рис. 63. Гистограмма изображения при $\alpha=1.0$.

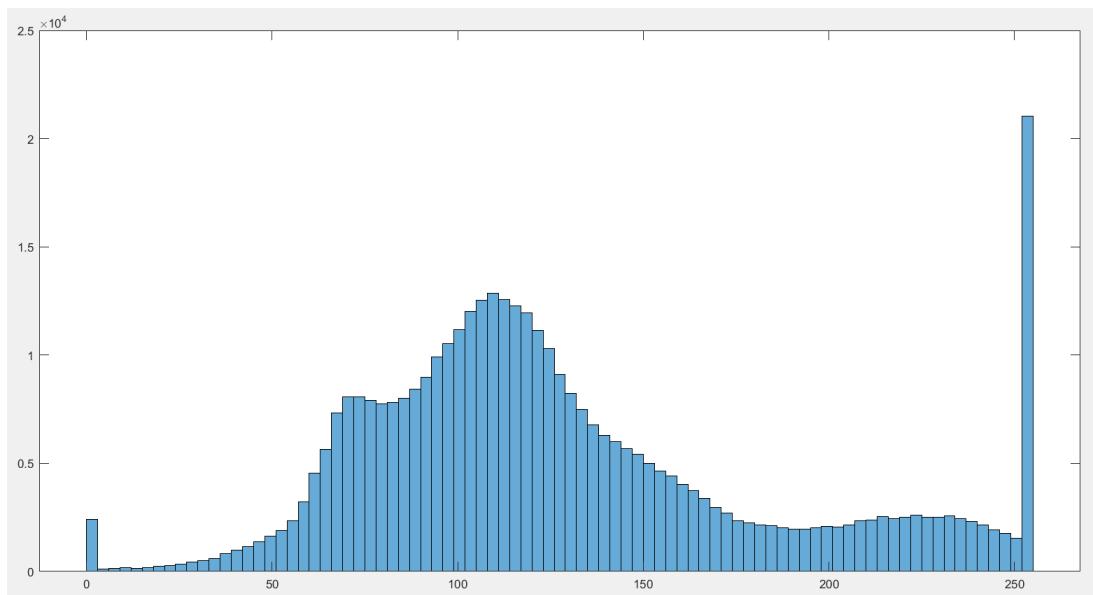


Рис. 64. Гистограмма изображения при $\alpha=1.1$.

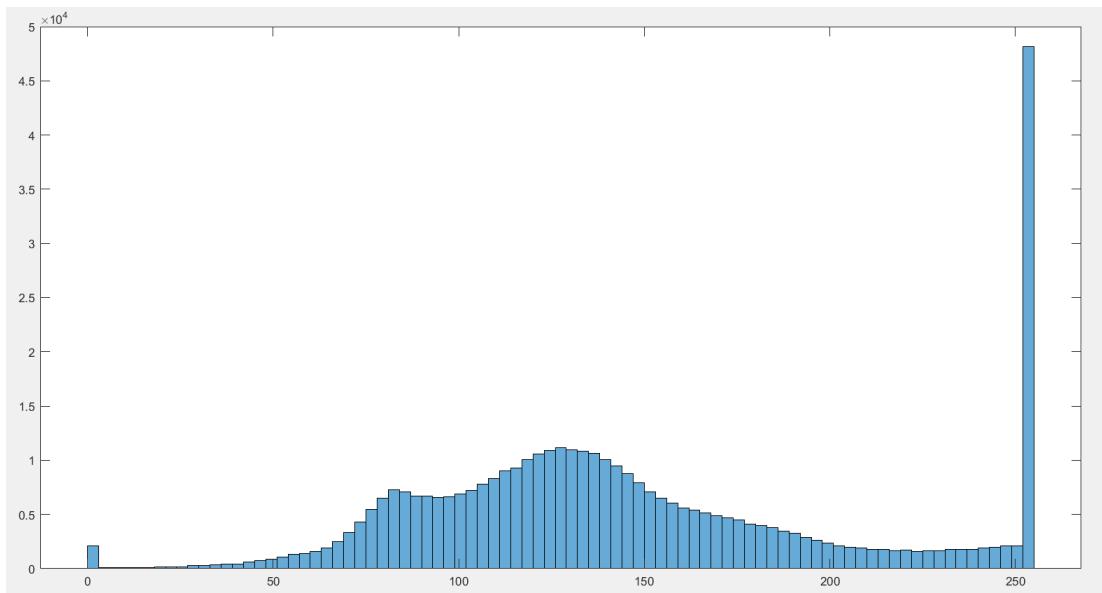


Рис. 65. Гистограмма изображения при $\alpha=1.2$.

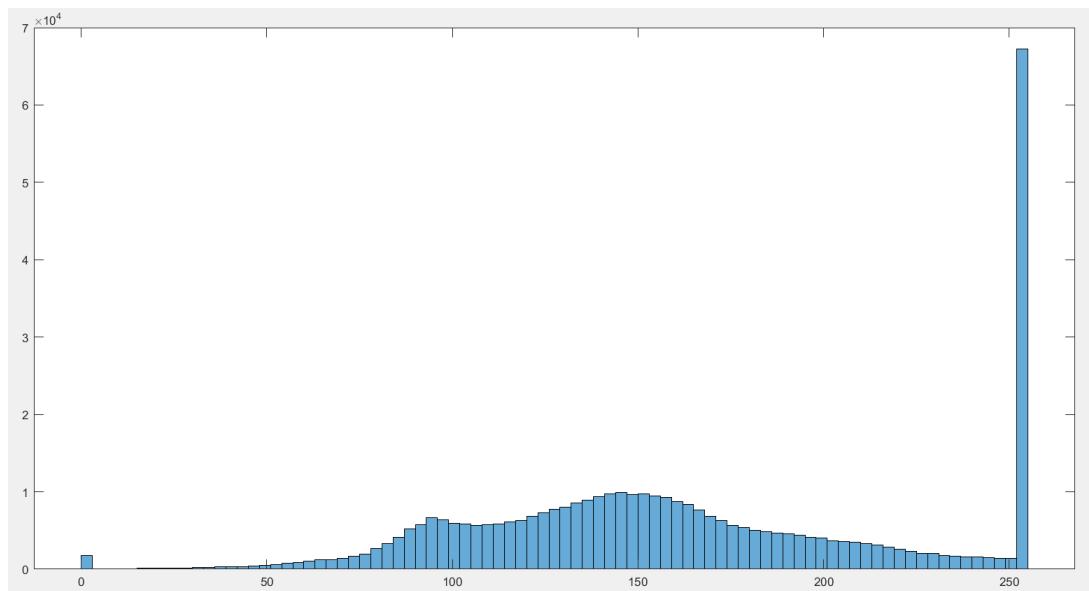


Рис. 66. Гистограмма изображения при $\alpha=1.3$.

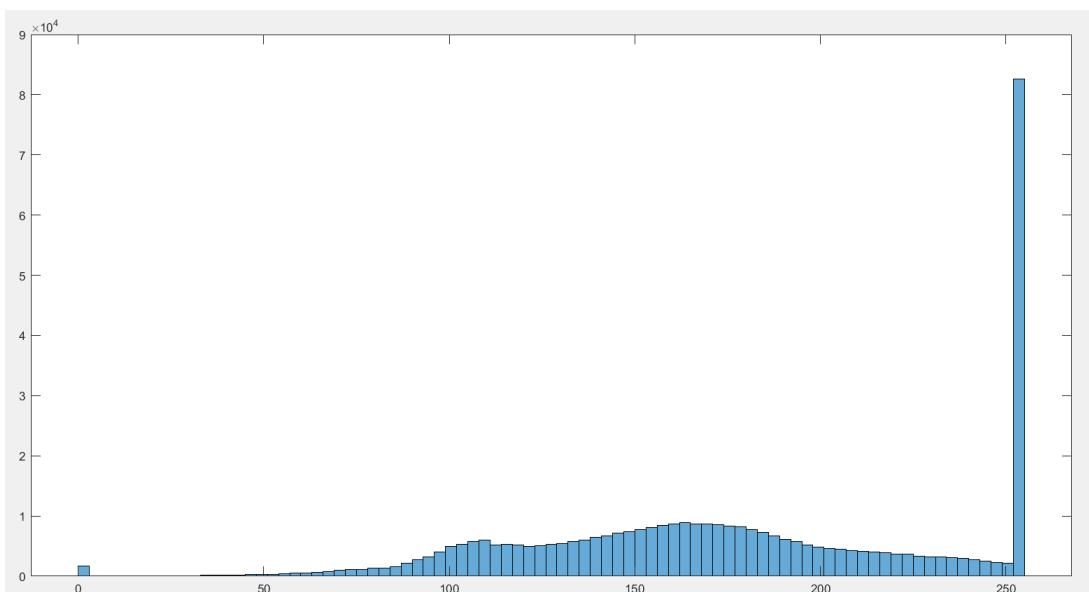


Рис. 67. Гистограмма изображения при $\alpha=1.4$.

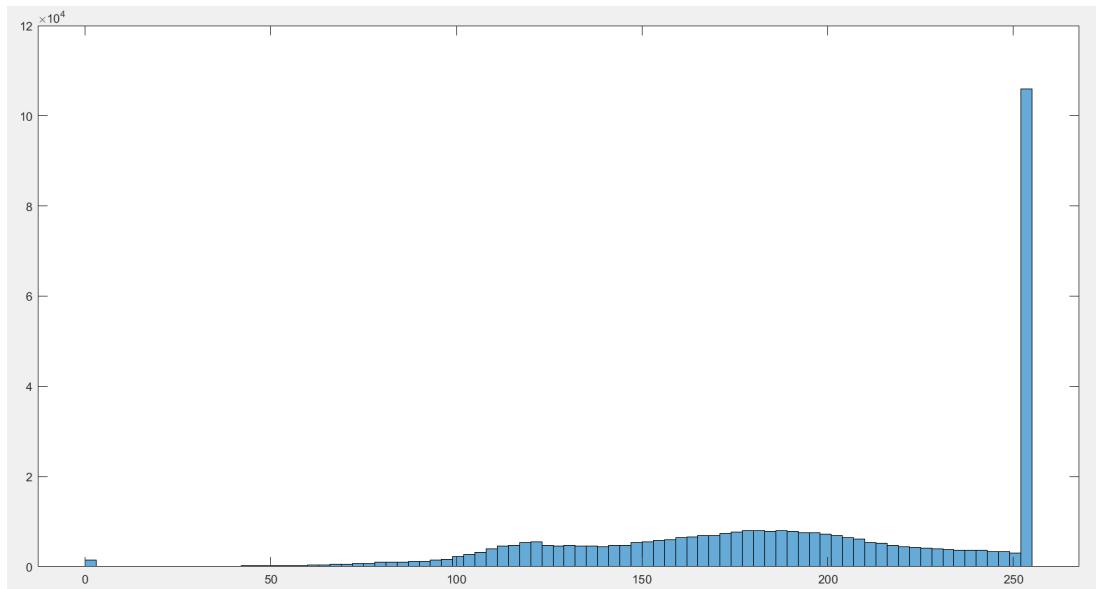


Рис. 68. Гистограмма изображения при $\alpha=1.5$.

По полученным гистограммам можно заметить, 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 освещением изображения – все больше компонент пикселей принимают свое максимальное значение.

7.8,7.9. Реализация оператора Собеля:

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:

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

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

Рис. 69. Маски фильтров в операторе Собеля.

Алгоритм применения оператора:

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

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

- 3) Рассчитать направление градиента $\theta_{x,y} = \arctg \frac{G_{x,y}^v}{G_{x,y}^h}$

Результат:



Рис. 70. Результат применения оператора Собеля.

7.10,7.11. Формирование изображений с различными порогами thr:



Рис. 71. Результат применения оператора Собеля при $thr=30$.



Рис. 72. Результат применения оператора Собеля при $thr=60$.



Рис. 73. Результат применения оператора Собеля при $thr=90$.



Рис. 74. Результат применения оператора Собеля при $thr=120$.



Рис. 75. Результат применения оператора Собеля при $thr=150$.



Рис. 76. Результат применения оператора Собеля при $thr=180$.

По сформированным изображениям можно сделать вывод, что наиболее точными бинарными картами являются карты с порогами $thr = 120$ и $thr = 90$. При $th=30$ и $th = 60$ изображения в некоторых местах перегружены белыми пикселями. При $thr = 150$ и $thr = 180$ некоторые контуры становятся четче различимых, а некоторые исчезают вовсе.

7.12. Карта направлений градиентов:

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

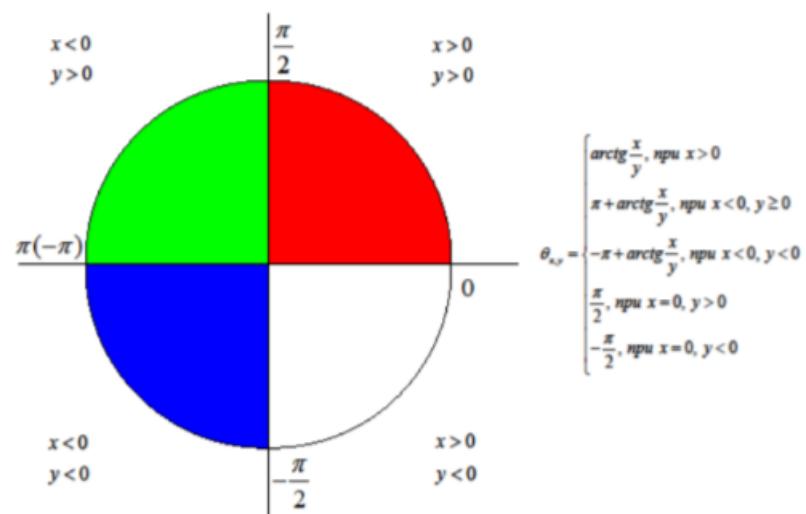


Рис. 77. Правило построения 4-цветной карты.

Результат:

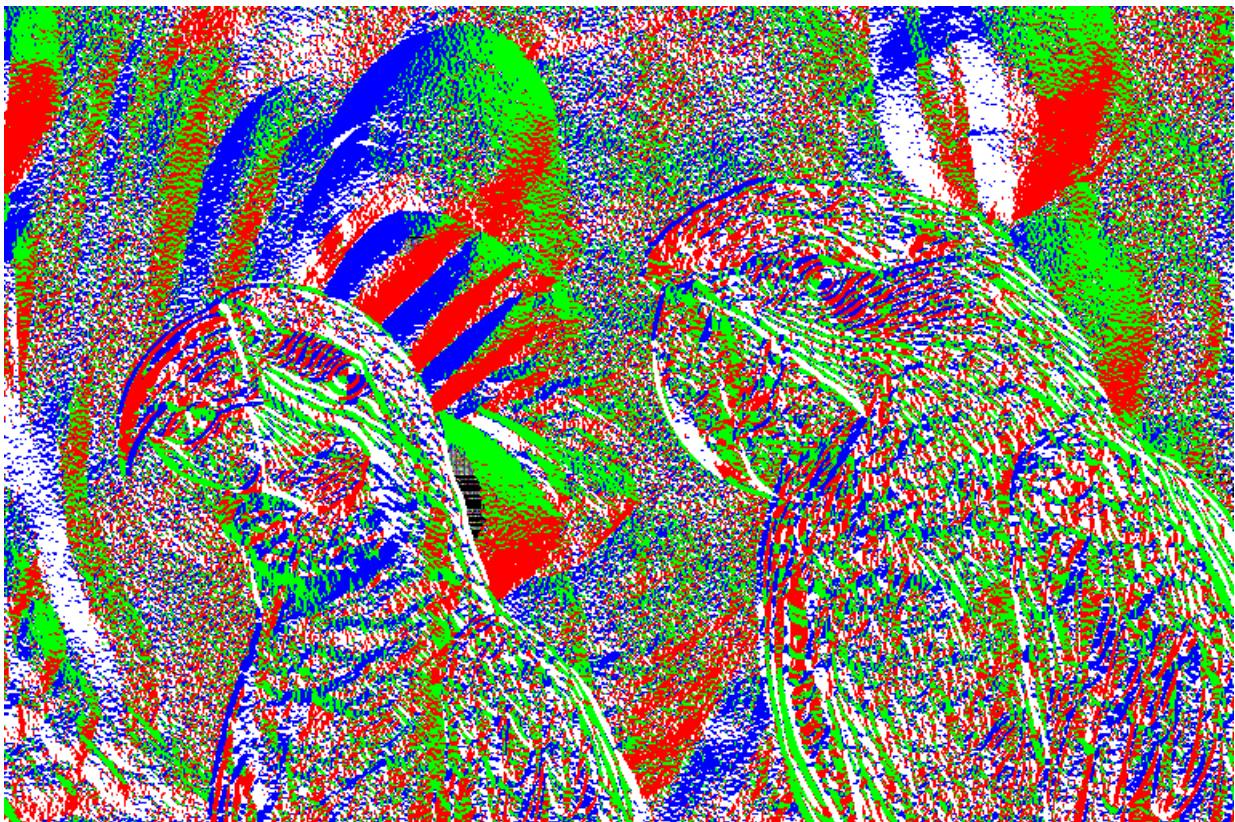


Рис. 78. Карта направлений градиентов.

7. Реализация градационных преобразований:
8.1. Синтез засвеченного, затемненного, сбалансированного изображений:



Рис. 79. Сбалансированное изображение.



Рис. 80. Синтезированное засвеченное изображение.



Рис. 81. Синтезированное затемненное изображение.

8.2. Преобразование на базе двух опорных точек:

К полученным в предыдущем пункте изображениям нужно применить градационное преобразование на базе двух опорных точек для повышения их качества. Метод опорных точек – один

из простейших методов улучшения качества изображения на основе анализа и видоизменения гистограммы основывается на использовании метода опорных точек. На основе точек (0,0) и (255,255) и двух опорных точек, которые задает человек, входное значение преобразуется с помощью линейной интерполяции в новое значение.

Результаты преобразования:



Рис. 82. Исходное сбалансированное изображение.



Рис. 83. Преобразование изображения на основе опорных точек (65,40) и (180,210).



Рис. 84. Исходное затемненное изображение.



Рис. 85. Преобразование изображения на основе опорных точек (90,170) и (130,230).



Рис. 86. Исходное засветленное изображение.



Рис. 87. Преобразование изображения на основе опорных точек (110,30) и (200,190).

8.3. Формирование гистограмм:

Необходимость сформировать и проанализировать гистограммы для исходных и полученных изображений из предыдущего пункта.

Результат:

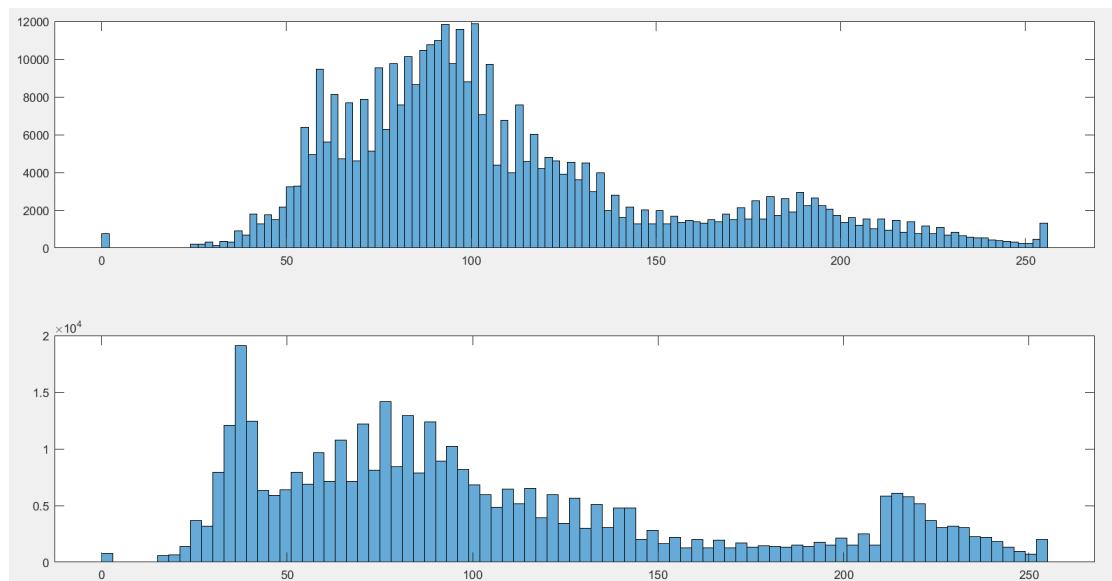


Рис. 88. Гистограмма сбалансированного изображения до и после преобразования.

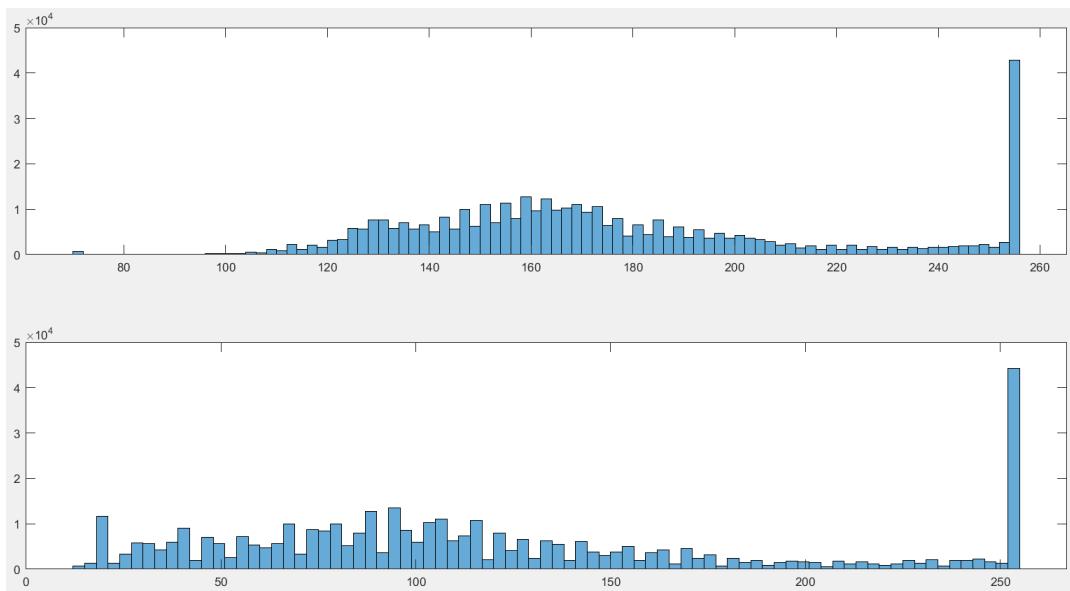


Рис. 89. Гистограмма засветленного изображения до и после преобразования.

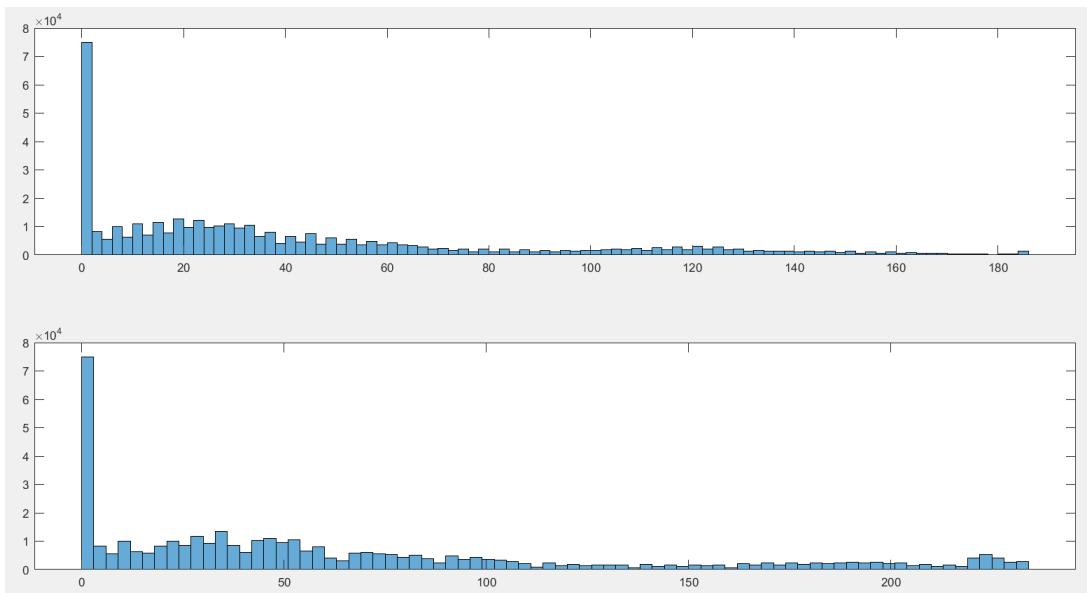


Рис. 90. Гистограмма затемненного изображения до и после преобразования.

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

8.4, 8.5. Гамма-преобразования:

Необходимо получить графики уравнения $I'_{x,y} = c(I_{x,y})^\gamma$ для различных значений параметра γ для каждого из трёх входных изображений. Однако перед применением формулы выше следует привести входное значение $I_{x,y}$ в диапазон $[0;1]$. Выходное значение $I'_{x,y}$ возвращают в диапазон $[0;255]$ путем умножения на 255.

Результат:



Рис. 91. Исходное сбалансированное изображение.



Рис. 92. Преобразованное исходное сбалансированное изображение при $\gamma = 0,1$.



Рис. 93. Преобразование исходное сбалансированное изображение при $\gamma = 0,5$.



Рис. 94. Преобразование исходное сбалансированное изображение при $\gamma = 1$.



Рис. 95. Преобразование исходное сбалансированное изображение при $\gamma = 2$.



Рис. 96. Преобразование исходное сбалансированное изображение при $\gamma = 8$.



Рис. 97. Исходное затемненное изображение.

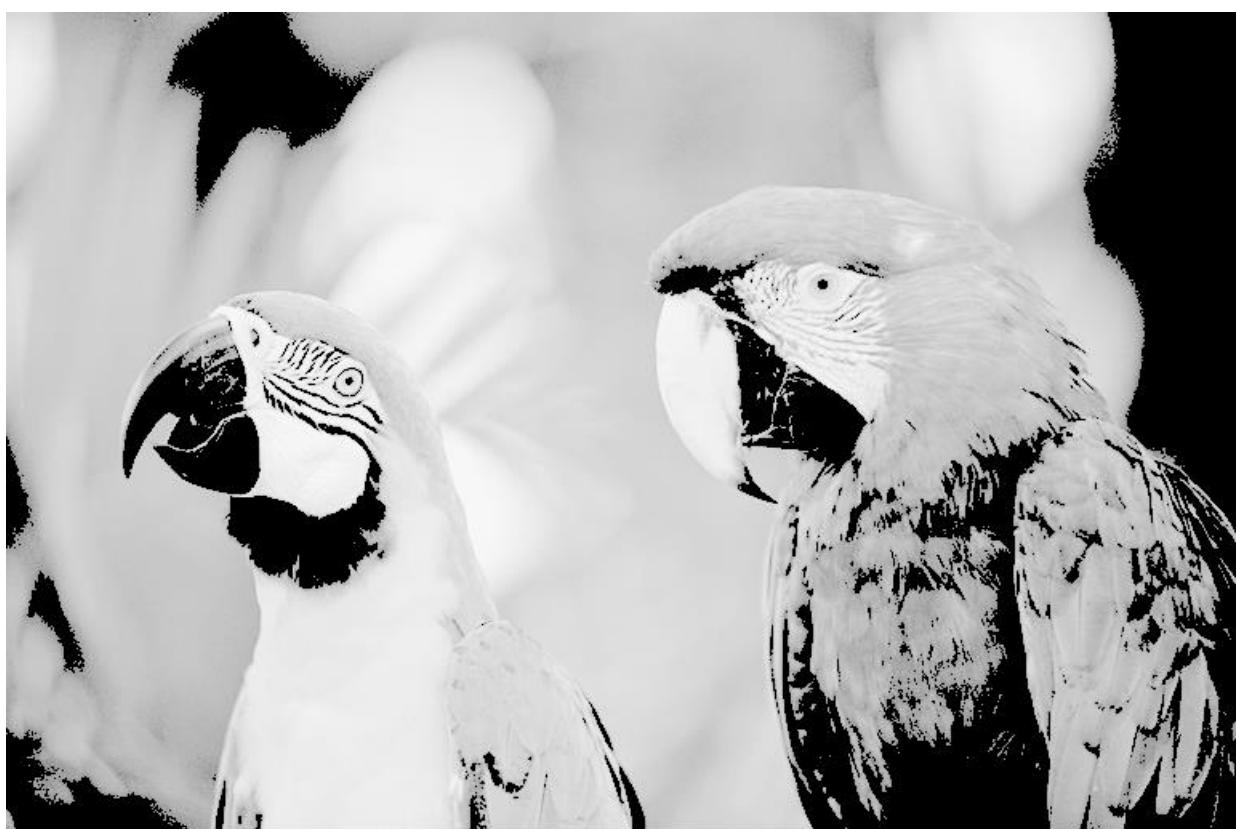


Рис. 98. Преобразование исходное затемненное изображение при $\gamma = 0,1$.



Рис. 99. Преобразование исходное затемненное изображение при $\gamma = 0,5$.



Рис. 100. Преобразование исходное затемненное изображение при $\gamma = 1$.



Рис. 101. Преобразование исходное затемненное изображение при $\gamma = 2$.

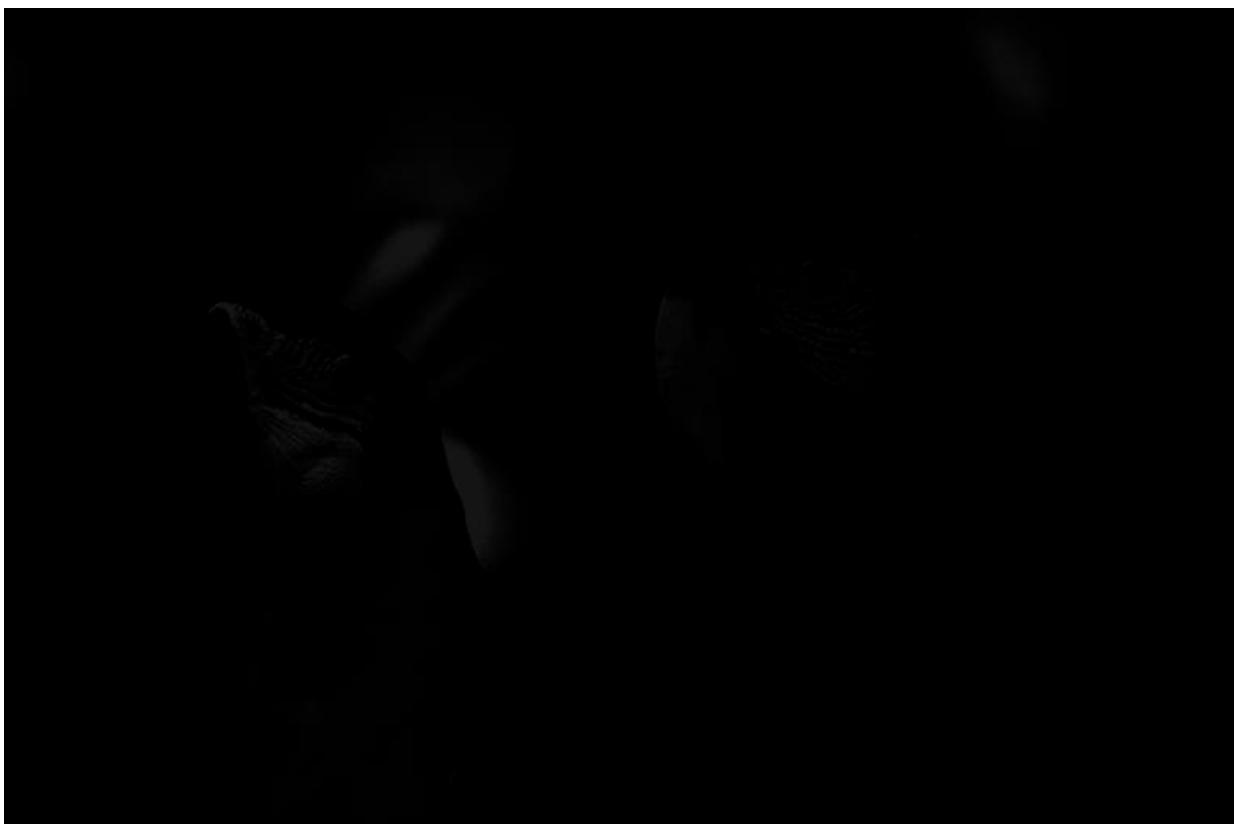


Рис. 102. Преобразование исходное затемненное изображение при $\gamma = 8$.



Рис. 103. Исходное засвеченное изображение.



Рис. 104. Преобразование исходное засвеченное изображение при $\gamma = 0,1$.



Рис. 105. Преобразование исходное засвеченное изображение при $\gamma = 0,5$.



Рис. 106. Преобразование исходное засвеченное изображение при $\gamma = 1$.



Рис. 107. Преобразование исходное засвеченное изображение при $\gamma = 2$.



Рис. 108. Преобразование исходное засвеченное изображение при $\gamma = 8$.

По полученным результатам можно сделать вывод, что при значениях $\gamma < 1$ яркость изображения увеличивается, а при $\gamma > 1$ уменьшается. Таким образом, для улучшения качества засвеченного изображения нужно брать значения $\gamma > 1$, для затемненного $\gamma < 1$. При $\gamma = 1$ изображение совпадает с исходным.

8.6. Формирование гистограмм:

Необходимо сформировать и проанализировать гистограммы для исходных и полученных изображений из предыдущего пункта.

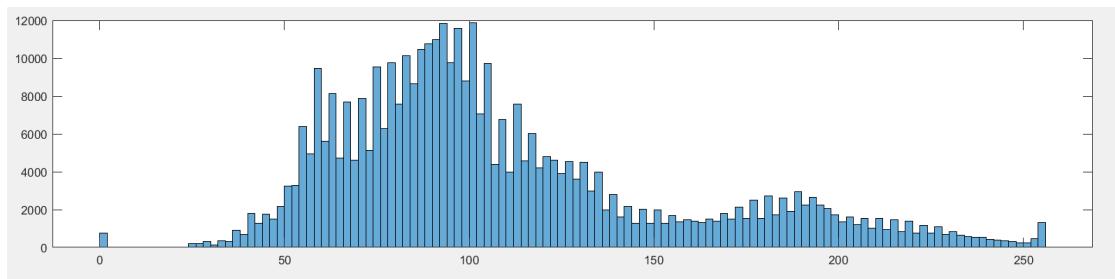


Рис. 109. Гистограмма исходного сбалансированного изображения.

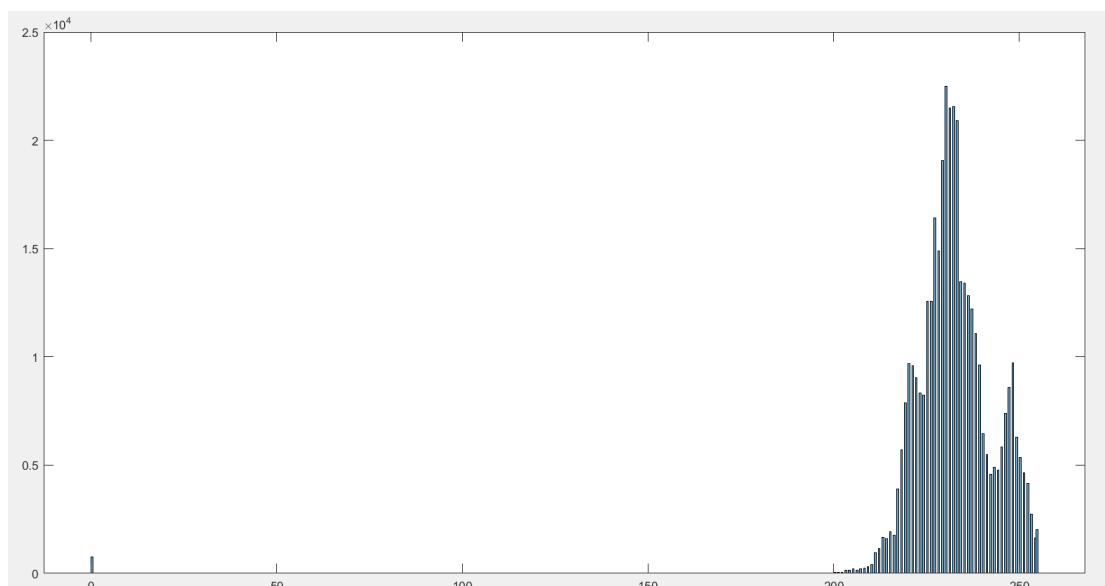


Рис. 110. Гистограмма исходного сбалансированного изображения при $\gamma = 0,1$.

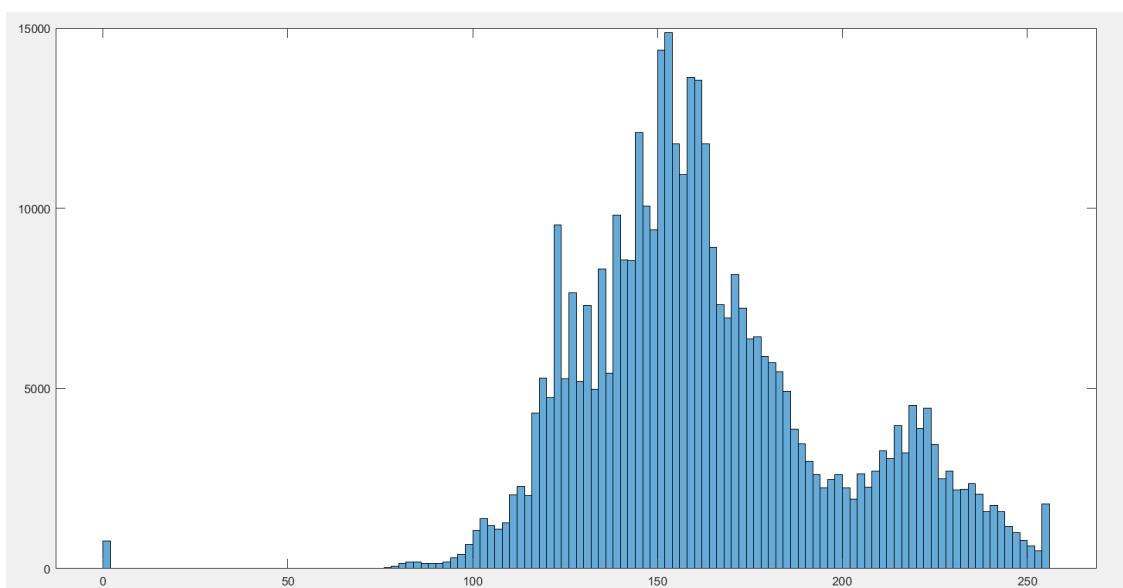


Рис. 111. Гистограмма исходного сбалансированного изображения при $\gamma = 0,5$.

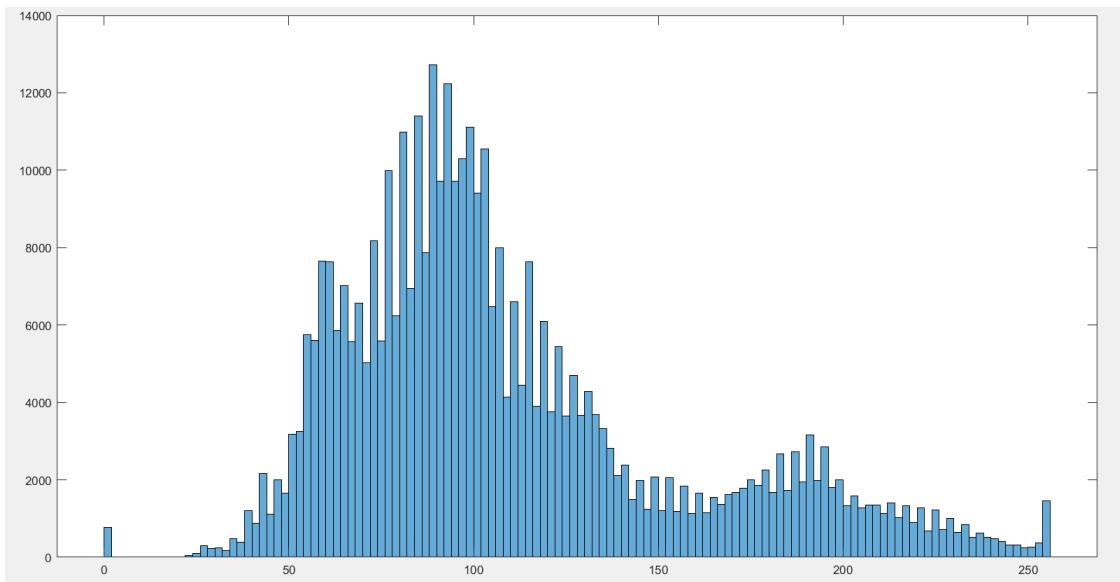


Рис. 112. Гистограмма исходного сбалансированного изображения при $\gamma = 1$.

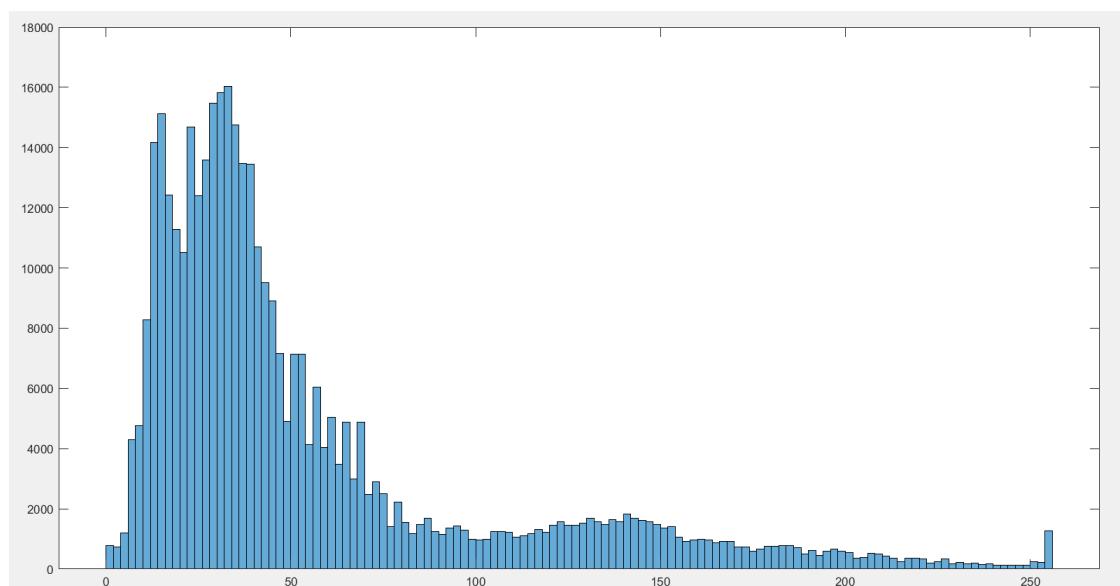


Рис. 113. Гистограмма исходного сбалансированного изображения при $\gamma = 2$.

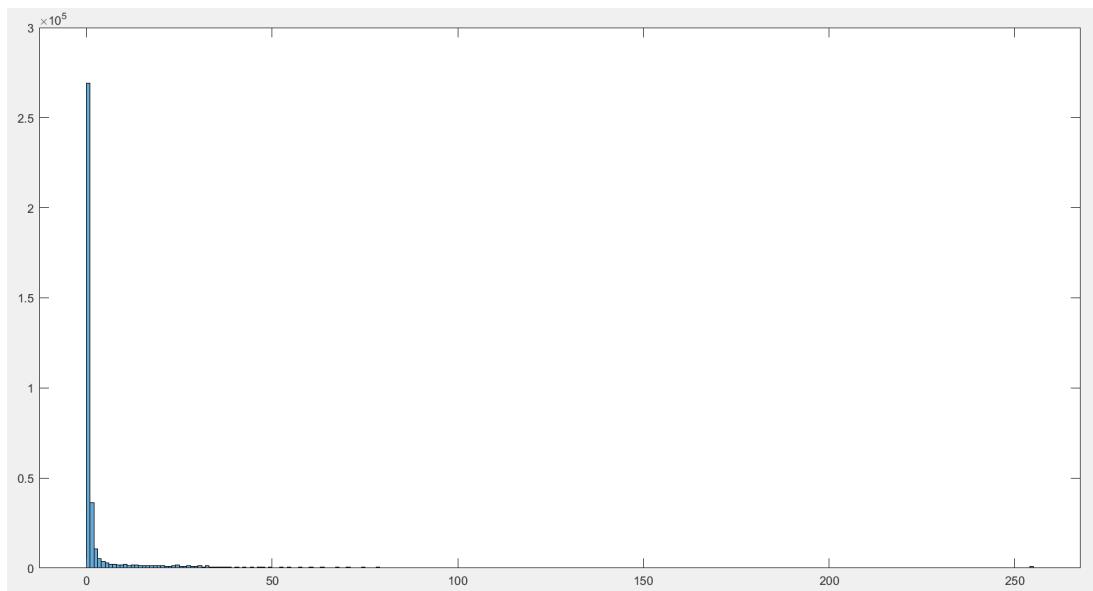


Рис. 114. Гистограмма исходного сбалансированного изображения при $\gamma = 8$.

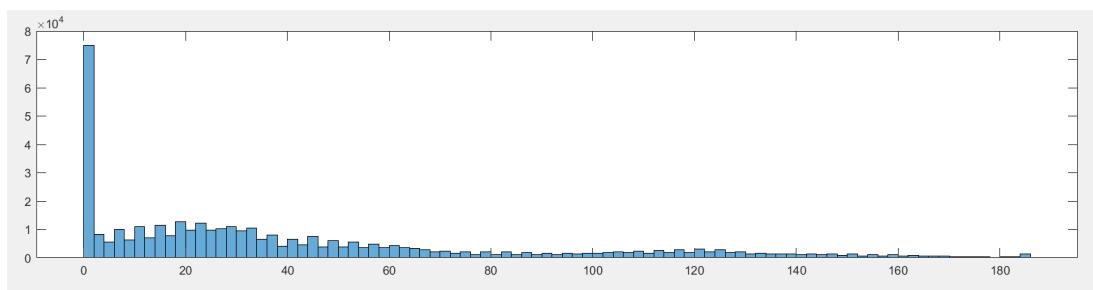


Рис. 115. Гистограмма исходного затемненного изображения.

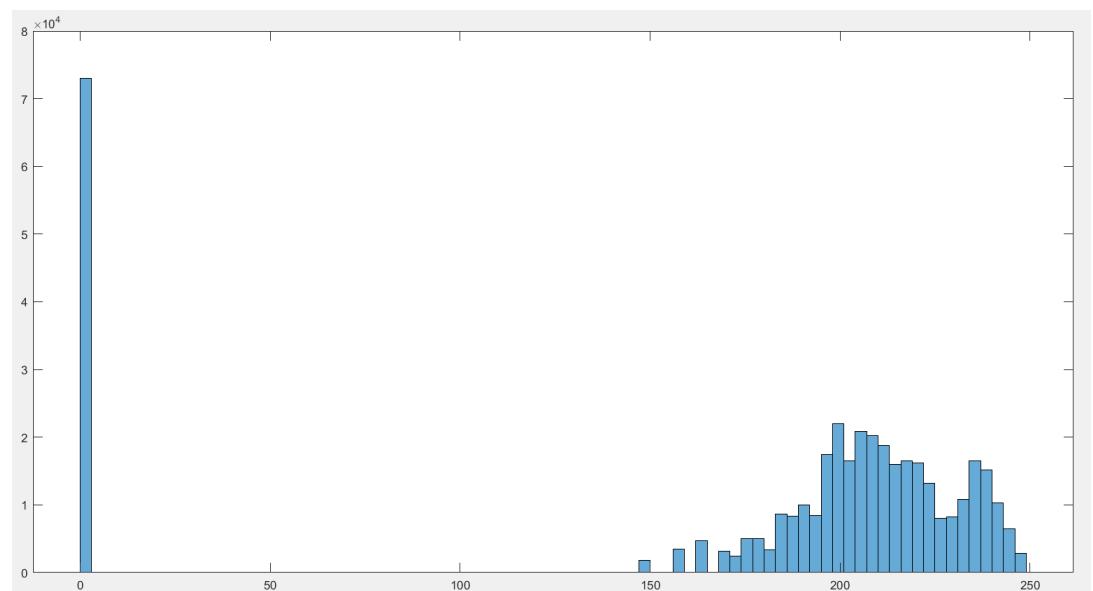


Рис. 116. Гистограмма исходного затемненного изображения при $\gamma = 0,1$.

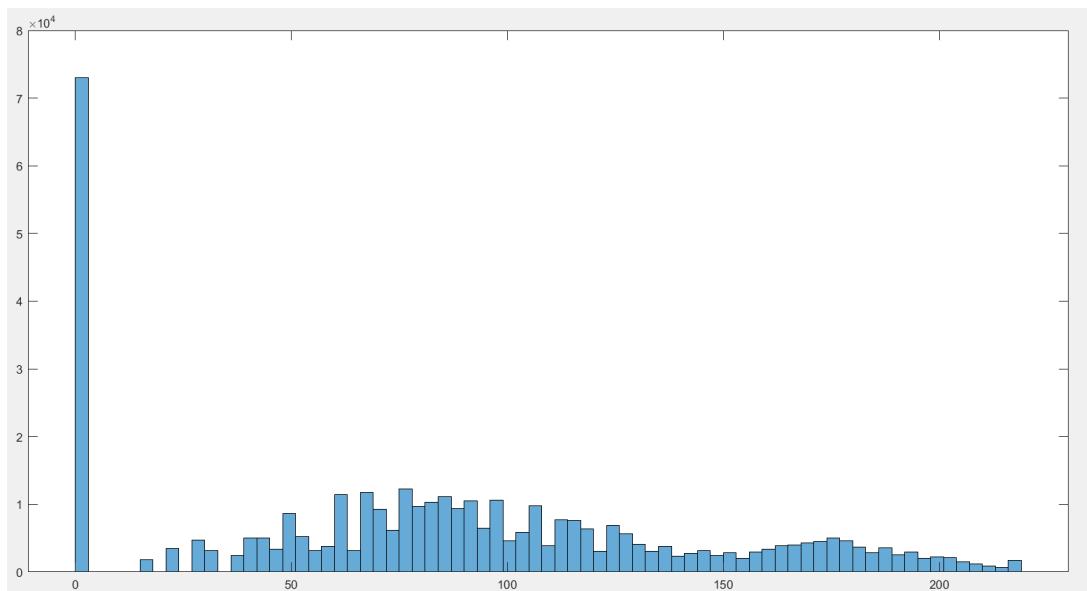


Рис. 117. Гистограмма исходного затемненного изображения при $\gamma = 0,5$.

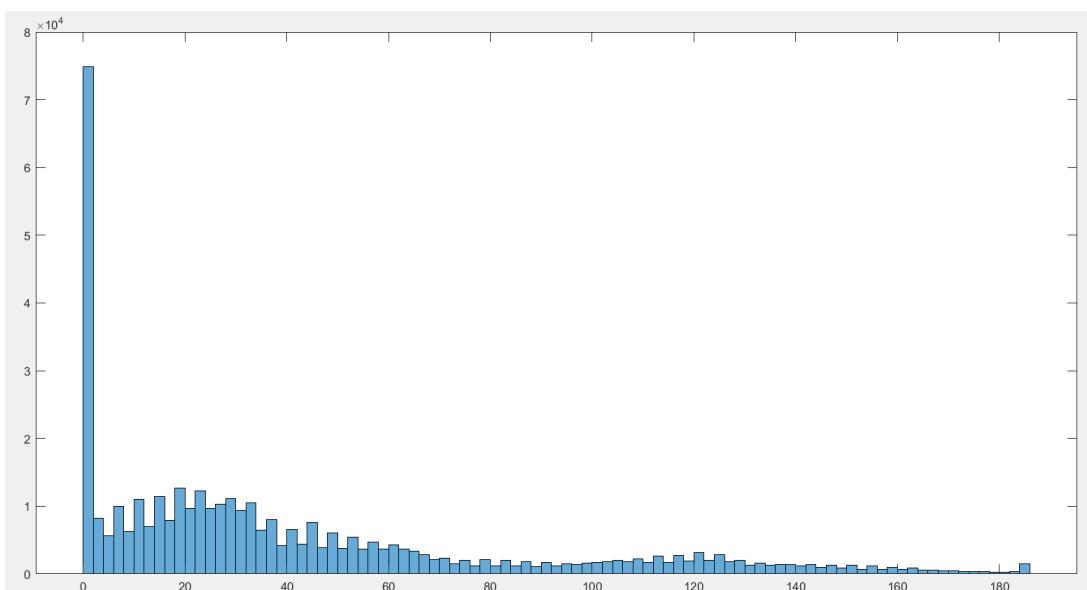


Рис. 118. Гистограмма исходного затемненного изображения при $\gamma = 1$.

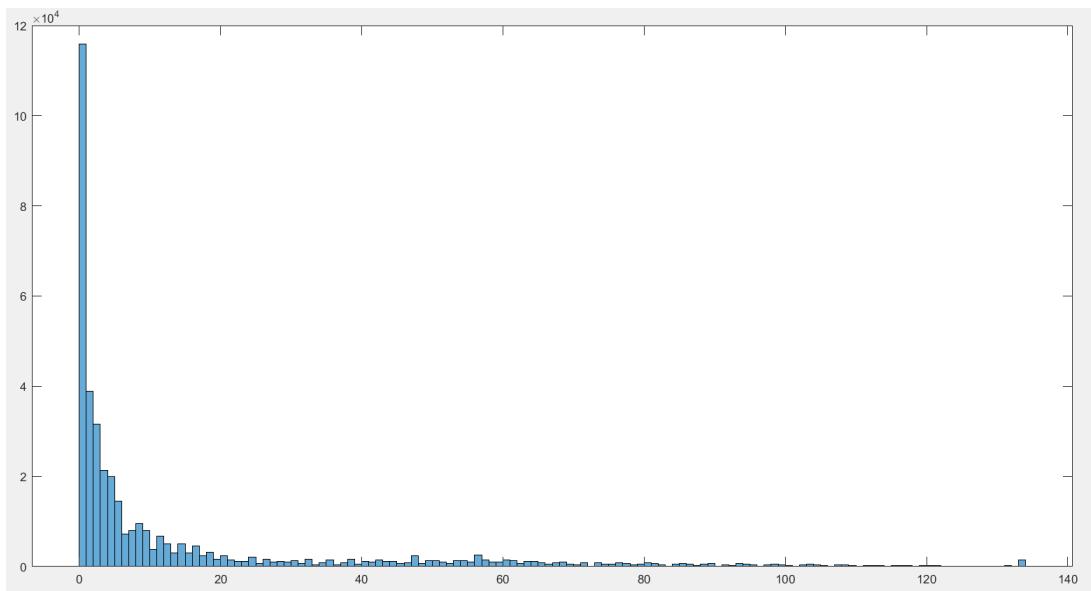


Рис. 119. Гистограмма исходного затемненного изображения при $\gamma = 2$.

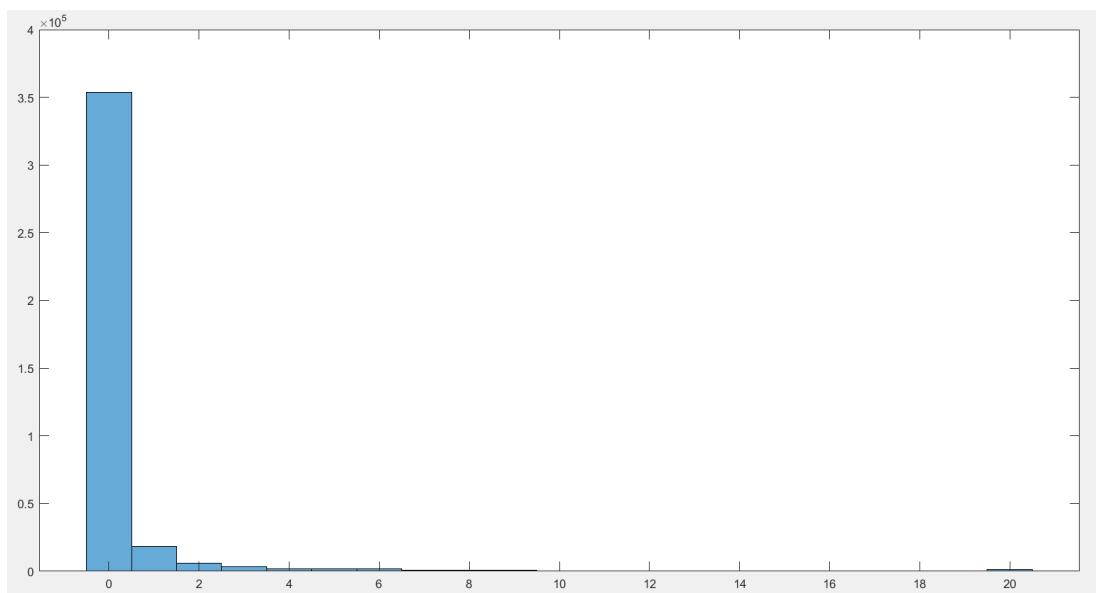


Рис. 120. Гистограмма исходного затемненного изображения при $\gamma = 8$.

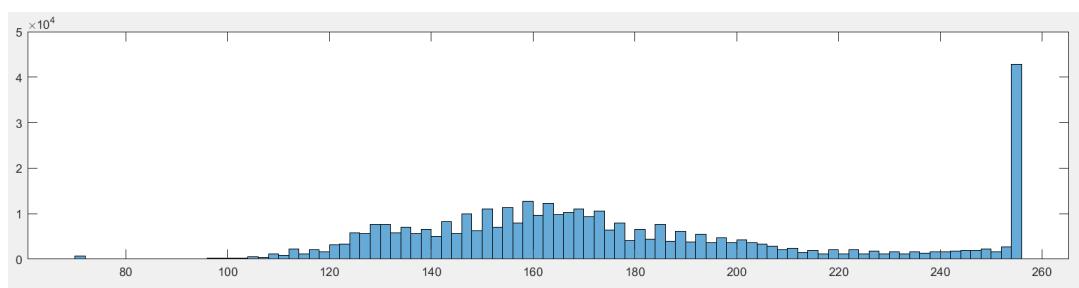


Рис. 121. Гистограмма исходного засвеченного изображения.

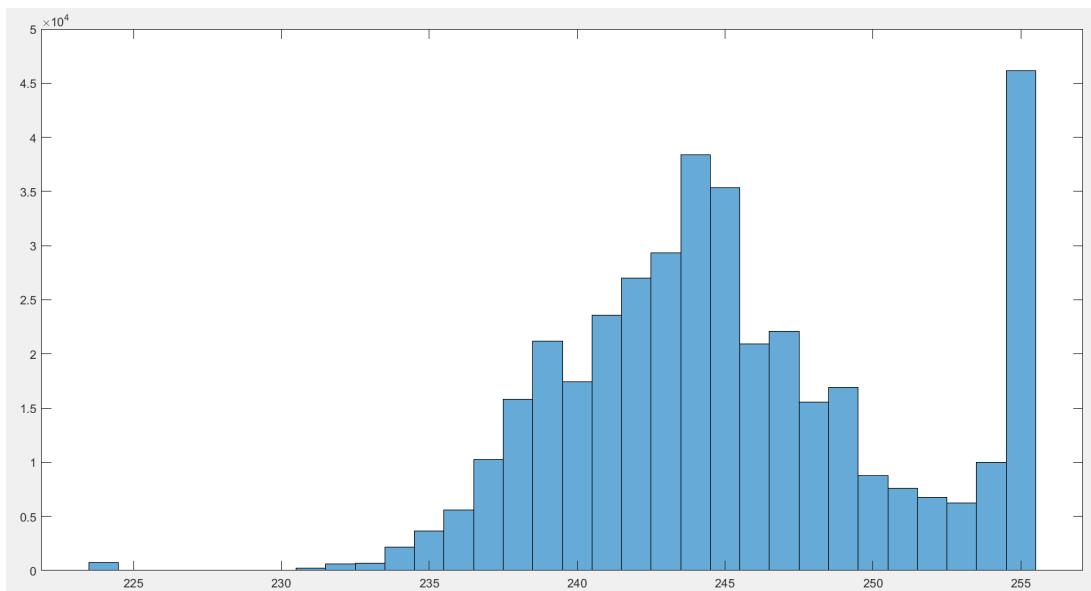


Рис. 122. Гистограмма исходного засвеченного изображения при $\gamma = 0,1$.

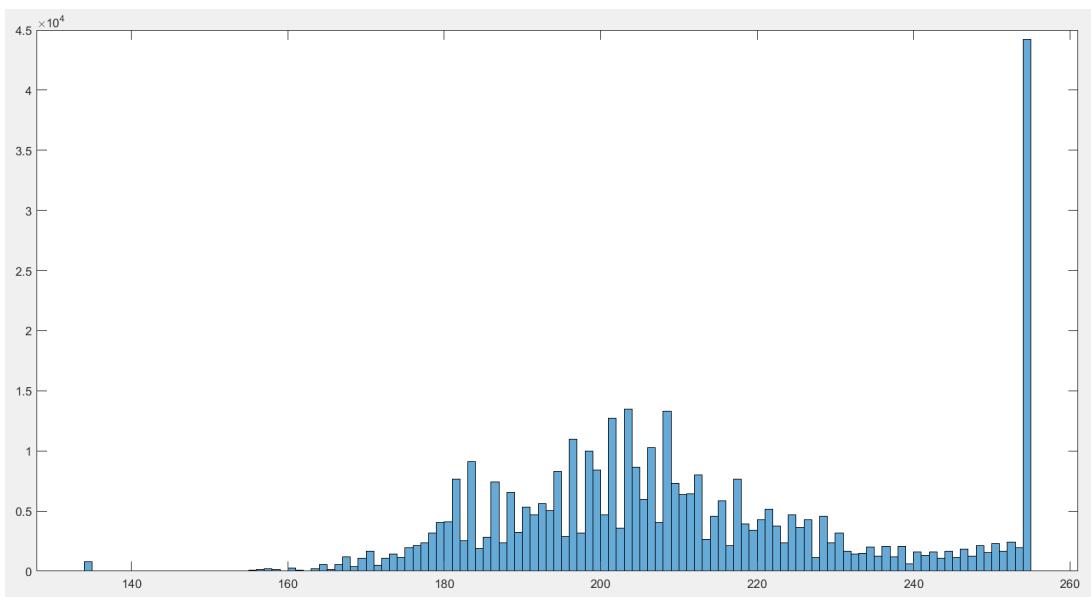


Рис. 123. Гистограмма исходного засвеченного изображения при $\gamma = 0,5$.

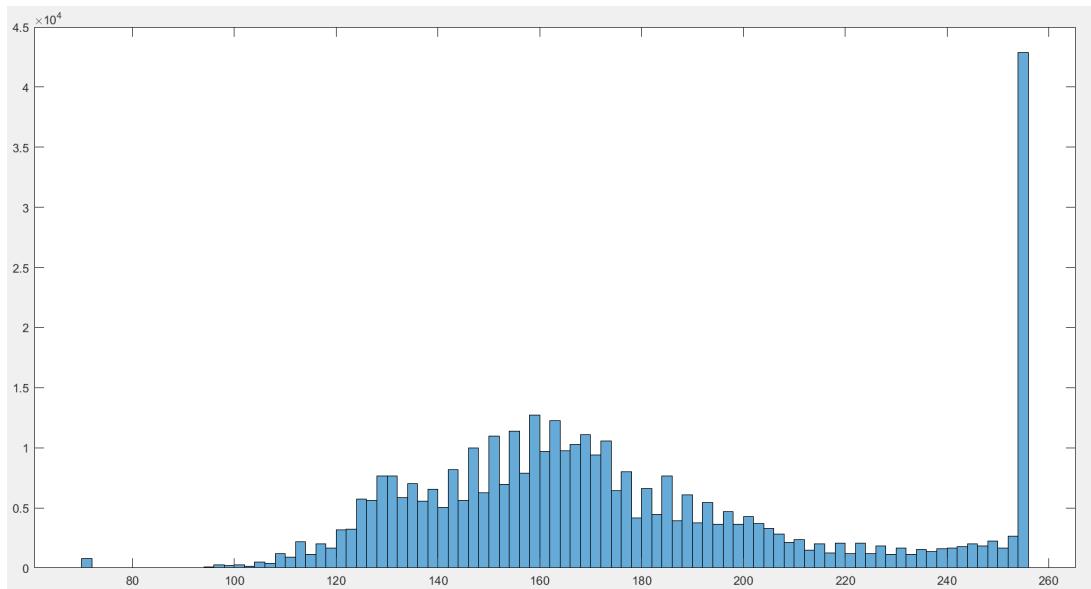


Рис. 124. Гистограмма исходного засвеченного изображения при $\gamma = 1$.

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

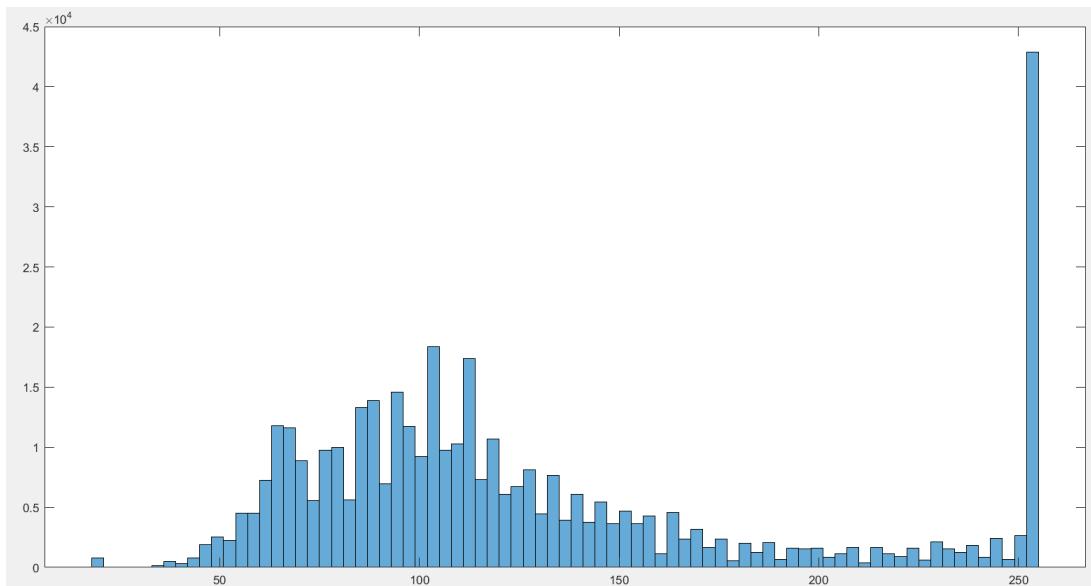


Рис. 125. Гистограмма исходного засвеченного изображения при $\gamma = 2$.

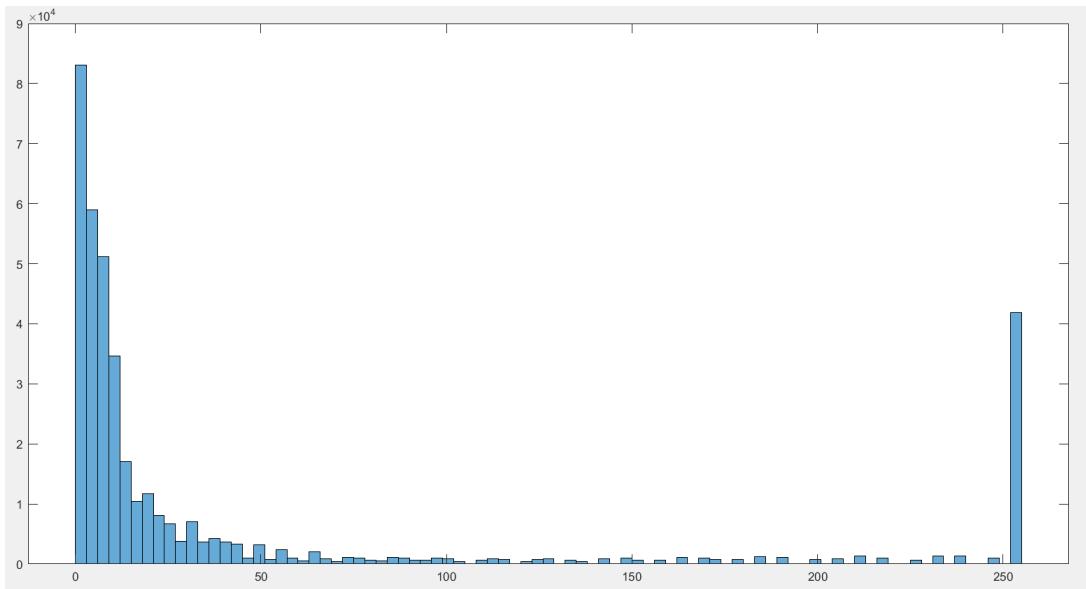


Рис. 126. Гистограмма исходного засвеченного изображения при $\gamma = 8$.

7.7-7.8. Методы выравнивания гистограмм:

Выравнивание гистограммы может быть использовано в задаче улучшения качества изображений. Данный подход может увеличить контрастность обрабатываемого изображения. Для выполнения операции выравнивания гистограмм следует воспользоваться формулой на базе эмпирической функции распределения:

$$s_k = \frac{2^L - 1}{N} \sum_{j=0}^k n_j,$$

где $2^L - 1$ – максимально допустимое число градаций интенсивности на изображении, s_k – значение интенсивности в выходном изображении, в которое будет преобразовываться r_k , N – общее число пикселей и n_j – число пикселей, имеющих яркость. Необходимо синтезировать засвеченное, затемненное, а также сбалансированное изображения и применить к ним алгоритм выравнивания гистограмм.

Результат:



Рис. 127. Исходное сбалансированное изображение.



Рис. 128. Обработанное сбалансированное изображение.



Рис. 129. Исходное засветленное изображение.



Рис. 130. Обработанное засветленное изображение.



Рис. 131. Исходное затемненное изображение.



Рис. 132. Обработанное затемненное изображение.

Результаты работы алгоритма совпадают на сбалансированном, засвеченном и затемненном изображениях (рис. 128, 130, 132).

8.9. Формирование гистограмм:

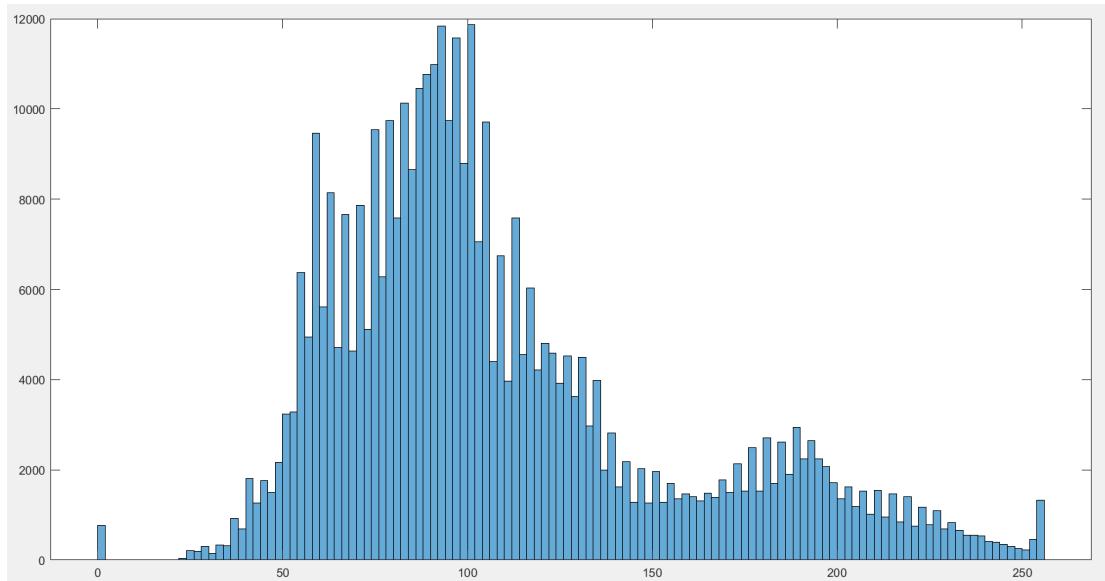


Рис. 133. Гистограмма исходного сбалансированного изображения.

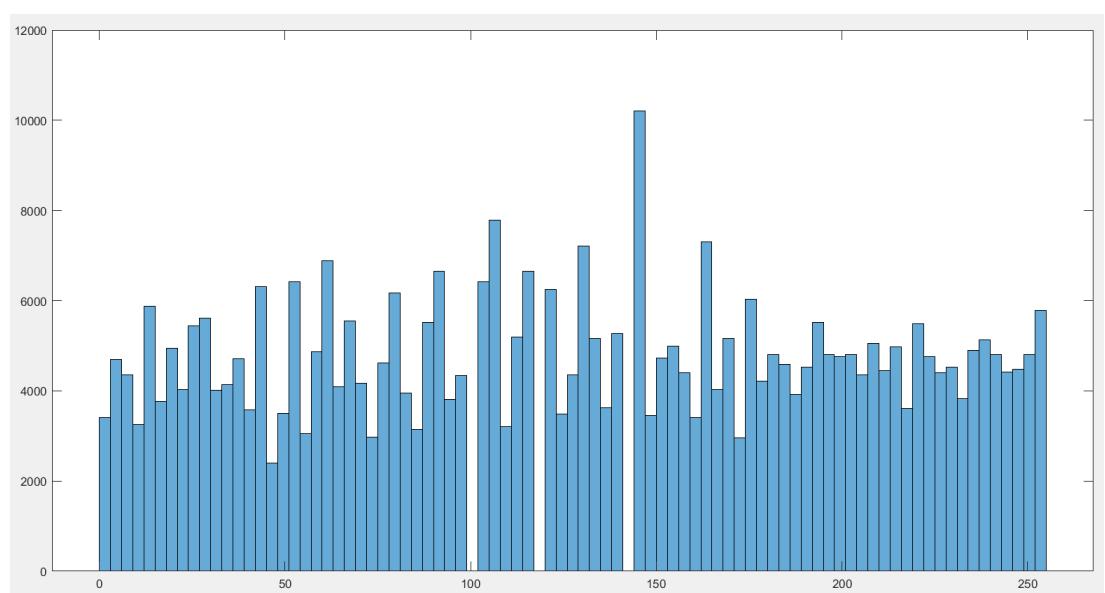


Рис. 134. Гистограмма исходного сбалансированного изображения после применения метода.

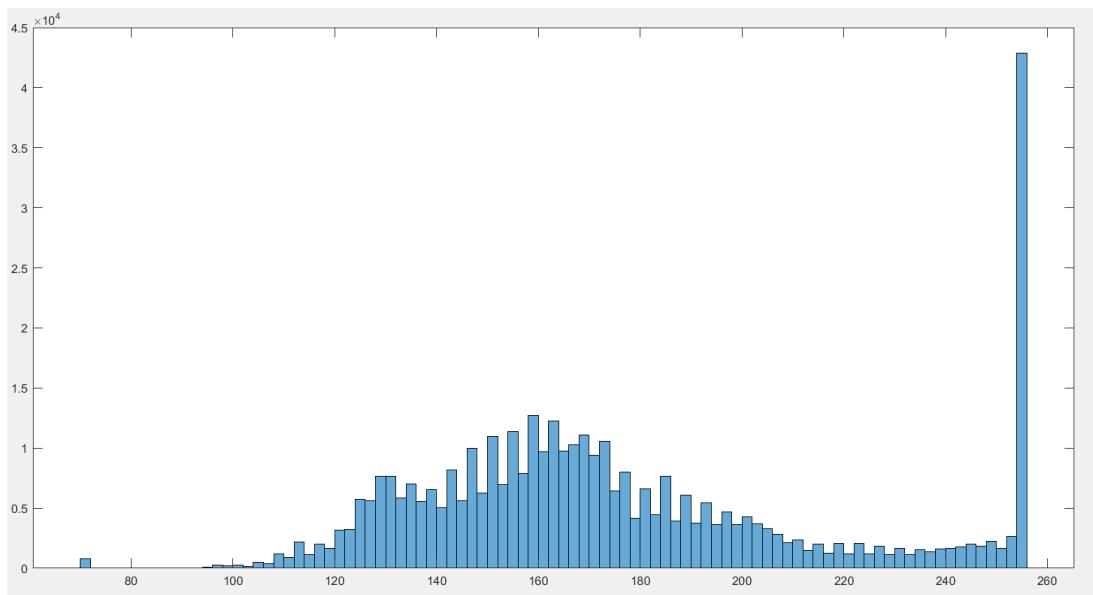


Рис. 135. Гистограмма исходного засветленного изображения.

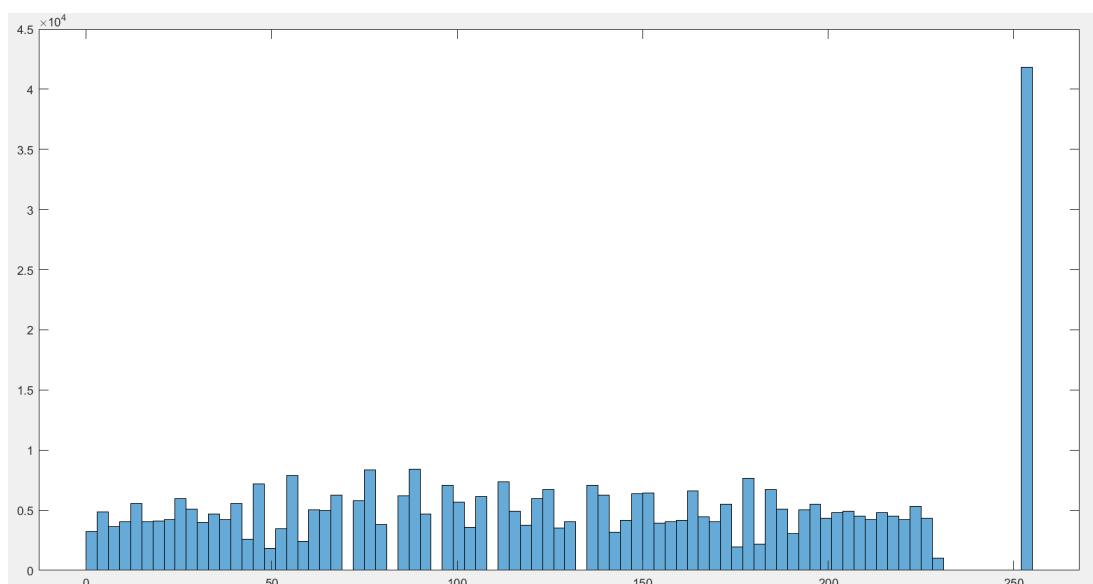


Рис. 136. Гистограмма исходного засветленного изображения после применения метода.

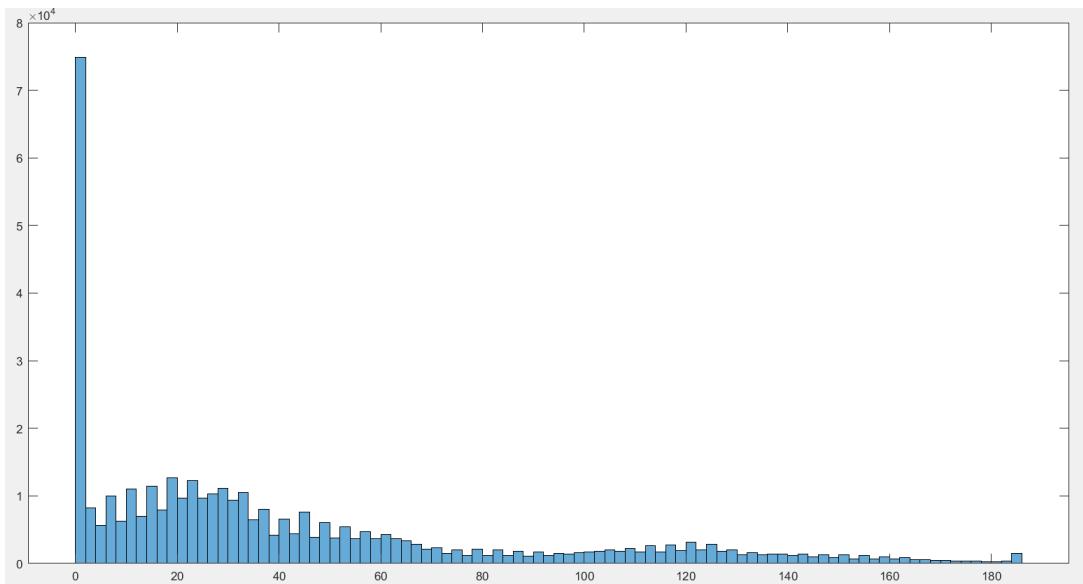


Рис. 137. Гистограмма исходного затемненного изображения.

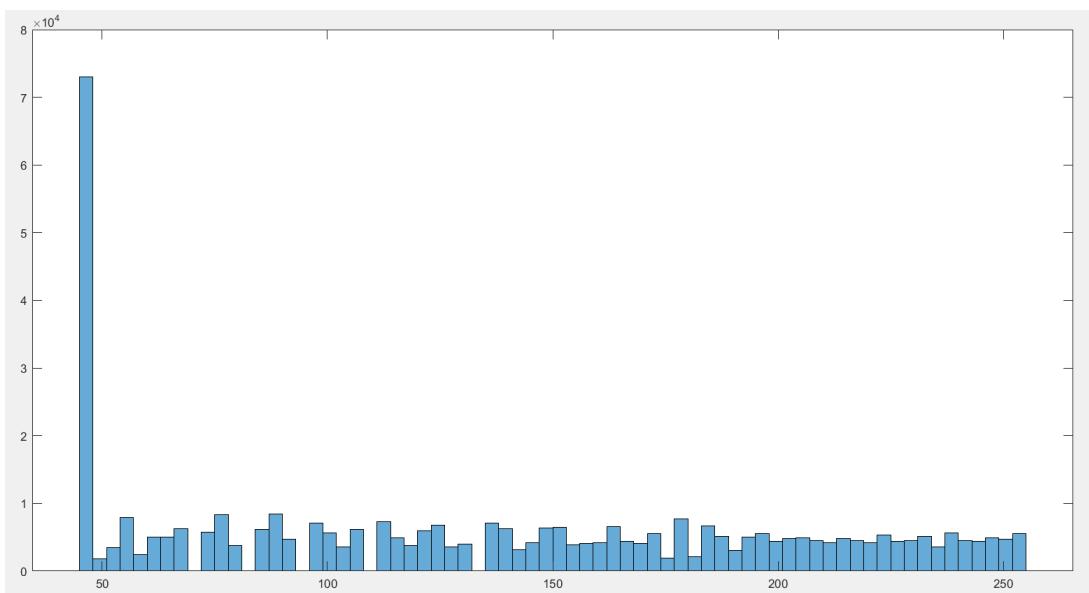


Рис. 138. Гистограмма исходного затемненного изображения после применения метода.

У затемненного и засвеченного изображений даже после применения метода выравнивания гистограмм крайние “пики” уходят неэффективно.

8.10. Методы построения карты контуров на основе градационных преобразований:

Метод построения 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 32.

Результат синтеза:



Рис. 139. Исходное сбалансированное изображение.

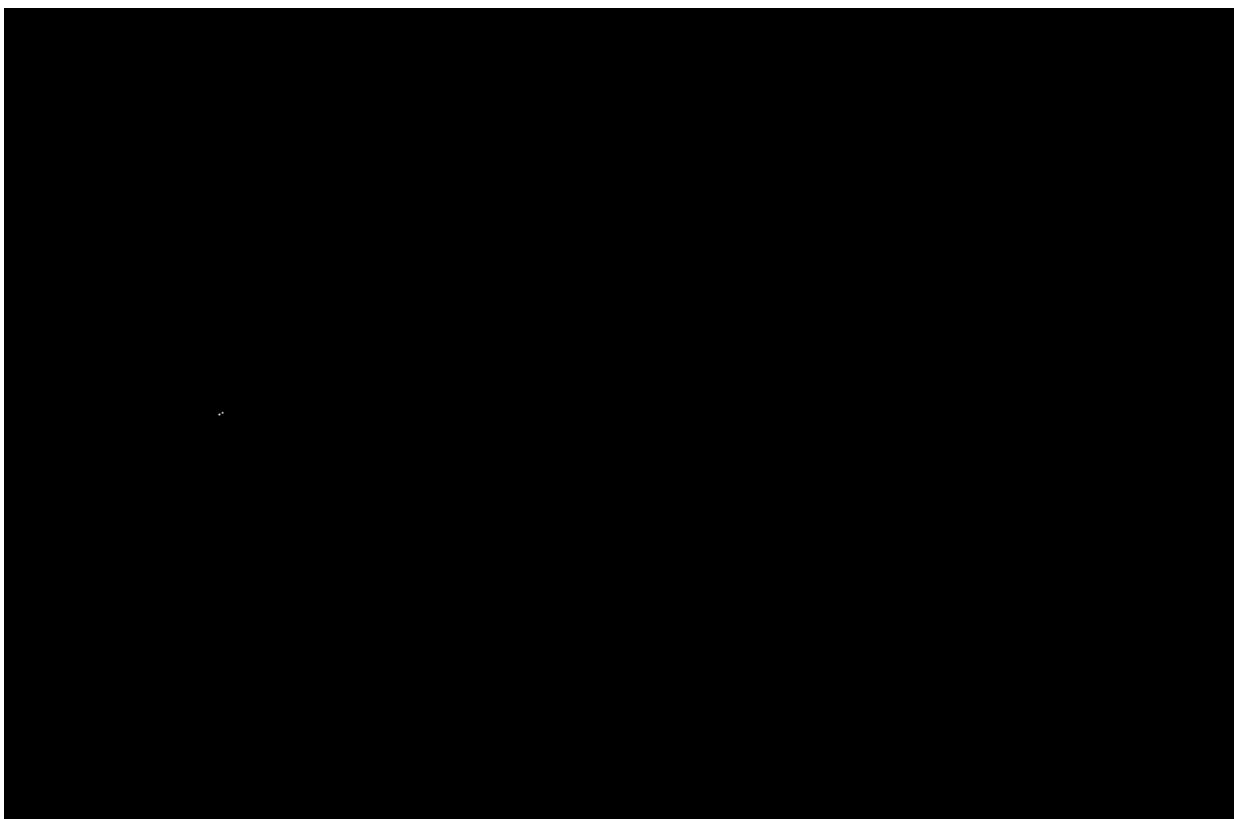


Рис. 140. Бинарное изображение при $T=16$.



Рис. 141. Бинарное изображение при $T=48$.



Рис. 142. Бинарное изображение при $T=80$.



Рис. 143. Бинарное изображение при $T=112$.



Рис. 144. Бинарное изображение при $T=144$.



Рис. 145. Бинарное изображение при $T=176$.



Рис. 146. Бинарное изображение при $T=208$.



Рис. 147. Бинарное изображение при $T=240$.

Изображение, полученное при $T=80$, вышло наиболее наглядным, так как достаточно хорошо видны мелкие детали и общий контур различим.

8. Выводы:

В ходе лабораторной работы, определили, что импульсный шум сильнее ухудшает изображение в сравнении с аддитивным, так как пиксели изображения принимают значения либо 0, либо 255 с заданной вероятностью. Благодаря дополнительному входному параметру лучшим фильтром для аддитивного шума является фильтр Гаусса, но только при небольших значениях сигмы шума. При высоких значениях фильтр Гаусса справляется плохо, как и остальные фильтры (даже для наилучших значений). Медианный фильтр дает наихудший визуальный результат, размывая изображения, но эта особенность помогает ему лучше справляться с импульсным шумом.

Оператор Лапласа при увеличении параметра альфа увеличивает общую яркость изображения. В результате применения оператора Лапласа и маски при $\alpha=1$ можно заметить, что контуры стали четче, а результат совпадает со второй производной Γ'' .

Оператор Собеля выдает наиболее точную бинарную карту при $thr=120$ и $thr=90$. При меньших значениях thr изображение в некоторых местах перегружено белыми пикселями, а при больших происходит ухудшение четкости контуров.

Методы опорных точек и гамма-преобразования помогают изменить оттенки серого, не затрагивая крайних значений пикселя. В методе опорных точек необходим человек, который будет подбирать оптимальные значения. Метод выравнивания гистограмм для засвеченного и затемнённого изображений приближают картинки к сбалансированному и выравнивают пиксели по всему диапазону значений. Метод на основе градационных преобразований дает наиболее точную карту контуров при значении порога $T = 80$.

9. Листинг программы:

```
#define _CRT_SECURE_NO_WARNINGS
#define _USE_MATH_DEFINES
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <fstream>
#include <vector>
#include <complex>
#include <map>
#include <math.h>
#include <algorithm>
#include <time.h>
#include <time.h>
using namespace std;

typedef struct BFH
{
    short bfType;
    int bfSize;
    short bfReserved1;
    short bfOffBits;;
    int bfReserved2;
} MBITMAPFILEHEADER;

typedef struct BIH
{
    int biSize;
    int biWidth;
    int biHeight;
    short int biPlanes;
    short int biBitCount;
    int biCompression;
    int biSizeImage;
    int biXPelsPerMeter;
    int biYPelsPerMeter;
    int biClrUsed;
    int biClrImportant;
} MBITMAPINFOHEADER;

typedef struct RGB
{
    unsigned char rgbBlue;
    unsigned char rgbGreen;
    unsigned char rgbRed;
} MRGBQUAD;

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;
```

```

MRGBQUAD** readBmp(FILE* f, MBITMAPFILEHEADER* bfh, MBITMAPINFO-
FOHEADER* bih)
{
    int k = 0;
    k = fread(bfh, sizeof(*bfh) - 2, 1, f);
    if (k == 0) {
        cout << "Error!" << endl;
        return 0;
    }

    k = fread(bih, sizeof(*bih), 1, f);
    if (k == NULL) {
        cout << "Error!" << endl;
        return 0;
    }

    int height = abs(bih->biHeight);
    int width = abs(bih->biWidth);
    MRGBQUAD** rgb = new MRGBQUAD * [height];
    for (int i = 0; i < height; i++) {
        rgb[i] = new MRGBQUAD[width];
    }
    int pad = 4 - (width * 3) % 4;
    for (int i = 0; i < height; i++) {
        fread(rgb[i], sizeof(MRGBQUAD), width, f);
        if (pad != 4) {
            fseek(f, pad, SEEK_CUR);
        }
    }
    return rgb;
}

void writeBMP(FILE* f, MRGBQUAD** rgbb, MBITMAPFILEHEADER* bfh, MBIT-
MAPINFOHEADER* 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((rgbb[i]), sizeof(MRGBQUAD), width, f);
        if (pad != 4) {
            fwrite(&buf, 1, pad, f);
        }
    }
}

void findMinMax(MRGBQUAD** rgbb, int height, int width) {
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            if (rgbb[i][j].rgbRed > max_R) max_R =
rgbb[i][j].rgbRed;
        }
    }
}

```

```

        if (rgb[i][j].rgbGreen > max_G)      max_G =
rgb[i][j].rgbGreen;
        if (rgb[i][j].rgbBlue > max_B)      max_B =
rgb[i][j].rgbBlue;
        if (rgb[i][j].rgbRed < min_R)  min_R =
rgb[i][j].rgbRed;
        if (rgb[i][j].rgbGreen < min_G)  min_G =
rgb[i][j].rgbGreen;
        if (rgb[i][j].rgbBlue < min_B)  min_B =
rgb[i][j].rgbBlue;
    }
}
}

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

MRGBQUAD** getRGBfromY(vector<vector<double>> Y, int height, int
width) {
    MRGBQUAD** rgb = new MRGBQUAD * [height];
    for (int i = 0; i < height; i++) {
        rgb[i] = new MRGBQUAD[width];
    }
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            rgb[i][j].rgbGreen = Y[i][j];
            rgb[i][j].rgbBlue = Y[i][j];
            rgb[i][j].rgbRed = Y[i][j];
        }
    }
    return rgb;
}

double calculteRandomDouble() {
    return (double)(rand() % 1000) / 1000.0;
}

double calculteRandomDoubleV2() {
    return (double)(rand() % 2000 - 1000) / 1000;
}

vector<vector<double>> generateImpulseNoise(vector<vector<double>>&
rgb, int height, int width, double probabilityA, double probabilityB)
{
    probabilityB = probabilityB + probabilityA;
    vector<vector<double>> result(height);
    for (int i = 0; i < height; i++) {

```

```

        for (int j = 0; j < width; j++) {
            double tmp = calculteRandomDouble();
            if (tmp <= probabilityA) {
                result[i].push_back(0);
                continue;
            }
            if (tmp <= probabilityB) {
                result[i].push_back(255);
                continue;
            }
            result[i].push_back(rgb[i][j]);
        }
    }
    return result;
}

vector<vector<double>> generateGaussianNoise(vector<vector<double>>& rgb, int height, int width, double sigma) {
    vector<vector<double>> result(height);
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j += 2) {
            double r = calculteRandomDouble();
            double phi = calculteRandomDouble();
            result[i].push_back(sigma * sqrt((-2) * log(r)) * cos(2 * M_PI * phi));
            result[i].push_back(sigma * sqrt((-2) * log(r)) * sin(2 * M_PI * phi));
        }
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                result[i][j] = clipping(rgb[i][j] + result[i][j]);
            }
        }
    }
    return result;
}

vector<vector<double>> generateGaussianNoiseV2(vector<vector<double>>& rgb, int height, int width, double sigma) {
    vector<vector<double>> result(height);
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j += 2) {
            double x = calculteRandomDoubleV2();
            double y = calculteRandomDoubleV2();
            double s = (x * x) + (y * y);
            while (s > 1 || s == 0)
            {
                x = calculteRandomDoubleV2();
                y = calculteRandomDoubleV2();
                s = (x * x) + (y * y);
            }
            result[i].push_back(sigma * x * sqrt(-2 * log(s) / s));
            result[i].push_back(sigma * y * sqrt(-2 * log(s) / s));
        }
    }
}

```

```

        }
    }
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            result[i][j] = clipping(rgb[i][j] + result[i][j]);
        }
    }
    return result;
}

void writeFile(string filename, vector<double>& array) {
    ofstream fout;
    fout.open(filename);
    for (int i = 0; i < array.size(); ++i) {
        fout << array[i] << "\n";
    }
    fout.close();
}

void writeFile(string filename, vector<double>& x, vector<double>& y)
{
    ofstream fout;
    fout.open(filename);
    for (int i = 0; i < x.size(); ++i) {
        fout << x[i] << " " << y[i] << "\n";
    }
    fout.close();
}

void writeFile(string filename, vector<vector<double>>& v, int
height, int width) {
    ofstream fout;
    fout.open(filename);
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            fout << v[i][j] << " ";
        }
        fout << "\n";
    }
    fout.close();
}

double calculateSumSquareDifferense(vector<vector<double>> firstAr-
ray, vector<vector<double>> secondArray) {
    double result = 0;
    for (int i = 0; i < firstArray.size(); i++) {
        for (int j = 0; j < firstArray[0].size(); j++) {
            result += pow((firstArray[i][j] - secondArray[i][j]), 2);
        }
    }
    return result;
}

```

```

double calculatePSNR(vector<vector<double>> firstArray, vector<vector<double>> secondArray) {
    return 10 * log10(firstArray.size() * firstArray[0].size() * pow((pow(2, 8) - 1), 2) / calculateSumSquareDifferense(firstArray, secondArray));
}

void buildGraphicsPSNR(vector<vector<double>>& array) {
    vector<double> result;
    vector<double> sigma = { 1, 10, 30, 50, 80 };
    for (int i = 0; i < sigma.size(); i++) {
        vector<vector<double>> gausY = generateGaussianNoiseV2(array, array.size(), array[0].size(), sigma[i]);
        result.push_back(calculatePSNR(array, gausY));
    }
    writeFile("additivePSNR.txt", sigma, result);
    vector<double> impulseResult;
    vector<double> freq = { 0.025, 0.05, 0.125, 0.25 };
    for (int i = 0; i < freq.size(); i++) {
        vector<vector<double>> impulseY = generateImpulseNoise(array, array.size(), array[0].size(), freq[i], freq[i]);
        impulseResult.push_back(calculatePSNR(array, impulseY));
    }
    writeFile("impulsePSNR.txt", freq, impulseResult);
}

vector<vector<double>> generateMovingAverage(vector<vector<double>>& array, int R) {
    vector<vector<double>> result(array.size());
    for (int i = 0; i < array.size(); i++) {
        for (int j = 0; j < array[0].size(); j++) {
            double tmp = 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 > array.size() - 1) {
                        x = array.size() - 1;
                    }
                    if (y < 0) {
                        y = 0;
                    }
                    if (y > array[0].size() - 1) {
                        y = array[0].size() - 1;
                    }
                    tmp += array[x][y];
                }
            }
        }
    }
}

```

```

        result[i].push_back(tmp / (double)pow((2 * R + 1),
2));
    }
}
return result;
}

void printSigmaPsnrR(vector<vector<double>>& array) {
    double sigma[] = { 1, 10, 30, 50, 80 };
    for (int i = 0; i < 5; i++) {
        cout << "SIGMA = " << sigma[i] << endl;
        vector<vector<double>> Y = generateGaussianNoiseV2(array,
array.size(), array[0].size(), sigma[i]);
        for (int R = 1; R <= 5; R++) {
            vector<vector<double>> YR = generateMovingAverage(Y,
R);
            cout << "R = " << R << " PSNR = " << calculatePSNR(ar-
ray, YR) << endl;
        }
    }
}

vector<vector<double>> doGaussianFilter(vector<vector<double>>& ar-
ray, int R, double sigma) {
    vector<vector<double>> result(array.size());
    for (int i = 0; i < array.size(); i++) {
        for (int j = 0; j < array[0].size(); j++) {
            double width = 0;
            double sum = 0;
            double Z = 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 > array.size() - 1) {
                        x = array.size() - 1;
                    }
                    if (y < 0) {
                        y = 0;
                    }
                    if (y > array[0].size() - 1) {
                        y = array[0].size() - 1;
                    }
                    width = exp((-k * k + m * m)) / (2.0 *
sigma * sigma));
                    Z = Z + width;
                    sum = sum + array[x][y] * width;
                }
            }
            result[i].push_back(sum / (double)Z);
        }
    }
}

```

```

        }
    }
    return result;
}

void buildGaussianFilterGraphicsPSNR(vector<vector<double>>& array) {
    vector<double> sigmaNoise = { 1, 10, 30, 50, 80 };
    vector<double> sigmaFilter = { 0.1, 0.25, 0.5, 0.75, 1, 1.25,
1.5, 2 };

    for (int R = 1; R <= 5; R += 2) {
        for (int i = 0; i < sigmaNoise.size(); i++) {
            vector<double> result(sigmaFilter.size());
            vector<vector<double>> Y = generateGaussianNoiseV2(ar-
ray, array.size(), array[0].size(), sigmaNoise[i]);
            for (int j = 0; j < sigmaFilter.size(); j++) {
                vector<vector<double>> YR = doGaussianFilter(Y,
R, sigmaFilter[j]);
                result[j] = (calculatePSNR(array, YR));
            }
            writeFile("R" + to_string(R) + "SN" + to_string(i) +
".txt", sigmaFilter, result);
            result.clear();
        }
    }
}

vector<vector<double>> doMedianFilter(vector<vector<double>>& array,
int R) {
    vector<vector<double>> result(array.size());
    for (int i = 0; i < array.size(); i++) {
        for (int j = 0; j < array[0].size(); j++) {
            vector<double> tmp;
            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 > array.size() - 1) {
                        x = array.size() - 1;
                    }
                    if (y < 0) {
                        y = 0;
                    }
                    if (y > array[0].size() - 1) {
                        y = array[0].size() - 1;
                    }
                    tmp.push_back(array[x][y]);
                }
            }
            result[i][j] = tmp;
        }
    }
}

```

```

        }

    }

    sort(tmp.begin(), tmp.end());
    result[i].push_back(tmp[tmp.size() / 2]);
}

}

return result;
}

void printGausNoiseMedianFilterPSNR(vector<vector<double>>& array) {
    cout << "Median Filter\n";
    double sigma[] = { 1, 10, 30, 50, 80 };
    for (int i = 0; i < 5; i++) {
        cout << "SIGMA = " << sigma[i] << endl;
        vector<vector<double>> Y = generateGaussianNoiseV2(array,
array.size(), array[0].size(), sigma[i]);
        for (int R = 1; R <= 5; R++) {
            vector<vector<double>> YR = doMedianFilter(Y, R);
            cout << "R = " << R << " PSNR = " << calculatePSNR(ar-
ray, YR) << endl;
        }
    }
}

void printImpulseNoiseMedianFilterPSNR(vector<vector<double>>& array) {
    cout << "Median Filter\n";
    vector<double> p = { 0.025, 0.05, 0.125, 0.25 };
    vector<double> R = { 1, 2, 3, 4, 5 };
    for (int i = 0; i < p.size(); i++) {
        cout << "a = b = " << p[i] << endl;
        vector<vector<double>> Y = generateImpulseNoise(array, ar-
ray.size(), array[0].size(), p[i], p[i]);
        vector<double> PSNR(R.size());
        for (int j = 0; j < R.size(); j++) {
            vector<vector<double>> YR = doMedianFilter(Y,
(int)R[j]);
            PSNR[j] = calculatePSNR(array, YR);
            cout << "R = " << (int)R[j] << " PSNR = " << PSNR[j]
<< endl;
        }
        writeFile("MFilterP" + to_string(p[i]) + ".txt", R, PSNR);
        PSNR.clear();
    }
}

void writeImage(string filename, MRGBQUAD** rgb, MBITMAPFILEHEADER* bfh, MBITMAPINFOHEADER* bih, int height, int width) {
    FILE* f = fopen(filename.c_str(), "wb");
    writeBMP(f, rgb, bfh, bih, height, width);
    fclose(f);
}

```

```

void printComparisonFilters(vector<vector<double>>& Y, MBITMAPFILE-
HEADER* bfh, MBITMAPINFOHEADER* bih, double sigma,
    double filterR, double filterSigma, double movingAverageR, dou-
ble medianFilterR, size_t i, vector<double>& resGausNoise,
    vector<double>& resGausFilter, vector<double>& resMovingAverage,
vector<double>& resMedianFilter) {
    int height = Y.size();
    int width = Y[0].size();
    vector<vector<double>> gausY = generateGaussianNoiseV2(Y,
Y.size(), Y[0].size(), sigma);
    vector<vector<double>> gausFilterY = doGaussianFilter(gausY,
filterR, filterSigma);
    MRGBQUAD** restoredRGB1 = getRGBfromY(gausY, height, width);
    MRGBQUAD** restoredRGB2 = getRGBfromY(gausFilterY, height,
width);

    double psnrNoise = calculatePSNR(Y, gausY);
    resGausNoise[i] = psnrNoise;
    double psnrGauseFilter = calculatePSNR(Y, gausFilterY);
    resGausFilter[i] = psnrGauseFilter;
    writeImage("gaussianNoise" + to_string((int)sigma) + ".bmp", re-
storedRGB1, bfh, bih, height, width);
    writeImage("gaussianFilter" + to_string((int)sigma) + ".bmp",
restoredRGB2, bfh, bih, height, width);
    cout << "sigma = " << sigma << "; gaussian noise PSNR = " <<
psnrNoise << "\n";
    cout << "sigma = " << sigma << "; gaussian filter PSNR = " <<
psnrGauseFilter << "\n";

    vector<vector<double>> movingAverageY = generateMovingAver-
age(gausY, movingAverageR);
    MRGBQUAD** restoredRGB3 = getRGBfromY(movingAverageY, height,
width);
    double psnrMovingAverage = calculatePSNR(Y, movingAverageY);
    resMovingAverage[i] = psnrMovingAverage;
    cout << "sigma = " << sigma << "; moving average PSNR = " <<
psnrMovingAverage << "\n";
    writeImage("movingAverage" + to_string((int)sigma) + ".bmp", re-
storedRGB3, bfh, bih, height, width);

    vector<vector<double>> medianFilterY = doMedianFilter(gausY, me-
dianFilterR);
    MRGBQUAD** restoredRGB4 = getRGBfromY(medianFilterY, height,
width);
    double psnrMedianFilter = calculatePSNR(Y, medianFilterY);
    resMedianFilter[i] = psnrMedianFilter;
    cout << "sigma = " << sigma << "; median filter PSNR = " <<
psnrMedianFilter << "\n";
    writeImage("medianFilter" + to_string((int)sigma) + ".bmp", re-
storedRGB4, bfh, bih, height, width);
    cout << endl;
}

```

```

void compareFilters(vector<vector<double>>& Y, MBITMAPFILEHEADER*
bfh, MBITMAPINFOHEADER* bih) {
    vector<double> sigma = { 1, 10, 30, 50, 80 };
    vector<double> filterR = { 1, 1, 1, 3, 5 };
    vector<double> filterSigma = { 0.25, 0.25, 0.75, 0.75, 1.25 };
    vector<double> movingAverageR = { 1, 1, 1, 1, 1 };
    vector<double> medianFilterR = { 1, 1, 1, 1, 2 };
    vector<double> resultGausNoise(sigma.size());
    vector<double> resultGausFilter(sigma.size());
    vector<double> resultMovingAverage(sigma.size());
    vector<double> resultMedianFilter(sigma.size());
    for (size_t i = 0; i < sigma.size(); i++) {
        printComparisonFilters(Y, bfh, bih, sigma[i], filterR[i],
filterSigma[i], movingAverageR[i], medianFilterR[i], i, re-
sultGausNoise, resultGausFilter, resultMovingAverage, resultMedian-
Filter);
    }
    writeFile("resultGausNoise.txt", sigma, resultGausNoise);
    writeFile("resultGausFilter.txt", sigma, resultGausFilter);
    writeFile("resultMovingAverage.txt", sigma, resultMovingAver-
age);
    writeFile("resultMedianFilter.txt", sigma, resultMedianFilter);
}

vector<vector<double>> laplasian(vector<vector<double>>& rgb, int
height, int width, double pl) {
    const int R = 1;
    vector<vector<double>> n(rgb.size());
    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) {
                n[i].push_back(clipping((rgb[i][j + 1] + rgb[i][j
- 1] + rgb[i + 1][j] + rgb[i - 1][j] - (4 * rgb[i][j])) + pl));
            }
            else
            {
                n[i].push_back(rgb[i][j]);
            }
        }
    }
    return n;
}

vector<vector<double>> changeBrightness(vector<vector<double>>& rgb,
unsigned char val, bool isDarked, int height, int width) {
    vector<vector<double>> n(height);
    for (size_t i = 0; i < height; i++) {
        for (size_t j = 0; j < width; j++) {
            if (isDarked) {
                double res = rgb[i][j] - val;
                n[i].push_back(clipping(res));
            }
            else {

```

```

        double res = rgb[i][j] + val;
        n[i].push_back(clipping(res));
    }
}
return n;
}

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);
}

vector<vector<double>> two_points(vector<vector<double>>& rgb, un-
signed char x0, unsigned char y0, unsigned char x1, unsigned char y1,
int height, int width) {
    vector<vector<double>> tmp(height);
    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 = rgb[i][j];
            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);
            tmp[i].push_back(res);
        }
    }
    return tmp;
}

vector<vector<double>> laplasianAlpha(vector<vector<double>>& rgb,
int height, int width) {
    const int R = 1;
    double weight_matrix[3][3] = { { 0, -1, 0 } , { -1, 4, -1 } , { 0,
-1, 0 } };

    vector<vector<double>> n(rgb.size());
    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 > rgb.size() - 1)
            x = rgb.size() - 1;
        if (y < 0)
            y = 0;
        if (y > rgb[0].size() - 1)
            y = rgb[0].size() - 1;
        double tmp = rgb[x][y];
        result += tmp * weight_matrix[k + 1][m + 1];
    }
}
n[i].push_back(clipping(rgb[i][j] + result));
}
return n;
}

vector<vector<double>> laplasian2(vector<vector<double>>& rgb, int height, int width, double pl) {
    const int R = 1;
    vector<vector<double>> n(rgb.size());
    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) {
                n[i].push_back(clipping((-rgb[i][j + 1] - rgb[i][j - 1] - rgb[i + 1][j] - rgb[i - 1][j] + (4 * rgb[i][j])) + pl));
            }
            else
            {
                n[i].push_back(rgb[i][j]);
            }
        }
    }
    return n;
}

vector<vector<double>> laplasianAlpha2(vector<vector<double>>& rgb, double alpha, int height, int width, double pl) {
    const int R = 1;
    vector<vector<double>> n(rgb.size());
    double weight_matrix[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 > rgb.size() - 1)
            x = rgb.size() - 1;
        if (y < 0)
            y = 0;
        if (y > rgb[0].size() - 1)
            y = rgb[0].size() - 1;
        double tmp = rgb[x][y];
        result += tmp * weight_matrix[k + 1][m + 1];
    }
}
n[i].push_back(clipping(rgb[i][j] * (alpha - 1) + result));
}
return n;
}

double calculateAverageBright(vector<vector<double>>& rgb) {
    double result = 0;
    for (int i = 0; i < rgb.size(); i++) {
        for (int j = 0; j < rgb[0].size(); j++) {
            result += rgb[i][j];
        }
    }
    return result / (rgb.size() * rgb[0].size());
}

typedef struct Sobel {
    double Gh;
    double Gv;
    double I;
    double theta;
} Sobel;

vector<vector<double>> doSobel(vector<vector<double>> rgb, int thr,
int height, int width) {
    vector<vector<double>> mas(height);
    const int R = 1;
    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++) {
        vector<Sobel> vec;
        for (int j = 0; j < width; j++) {
            double Gh = 0, Gv = 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 = rgb[x][y];
        Gh += cur * mask_h[k + 1][m + 1];
        Gv += cur * mask_v[k + 1][m + 1];
    }
}
double I = sqrt(pow(Gh, 2) + pow(Gv, 2));
if (I > thr) I = 255;
else I = 0;
double theta = atan2(Gv, Gh);
mas[i].push_back(I);
}
}
return mas;
}

vector<vector<Sobel>> doSobelParametr(vector<vector<double>> rgb, int
thr, int height, int width) {
    vector<vector<Sobel>> sobelData;
    vector<vector<double>> mas(height);
    const int R = 1;
    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++) {
        vector<Sobel> vec;
        for (int j = 0; j < width; j++) {
            Sobel tmp;
            double Gh = 0, Gv = 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 = rgb[x][y];
                    Gh += cur * mask_h[k + 1][m + 1];
                    Gv += cur * mask_v[k + 1][m + 1];
                }
            }
            tmp.Gh = Gh;
            tmp.Gv = Gv;
            tmp.I = sqrt(pow(Gh, 2) + pow(Gv, 2));
            if (tmp.I > thr) tmp.I = 255;
            else tmp.I = 0;
            tmp.theta = atan2(Gv, Gh);
            mas[i].push_back(tmp.I);
            vec.push_back(tmp);
        }
        sobelData.push_back(vec);
    }
}

```

```

        return sobelData;
    }

MRGBQUAD** calculateGradient(vector<vector<Sobel>>& sobel_data, int
height, int width) {
    MRGBQUAD** rgb;
    rgb = new MRGBQUAD * [height];
    for (int i = 0; i < height; i++) {
        rgb[i] = new MRGBQUAD[width];
    }
    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 (sobel_data[i][j].Gh > 0 && sobel_data[i][j].Gv >
0) rgb[i][j] =
                red;
            if (sobel_data[i][j].Gh < 0 && sobel_data[i][j].Gv >
0) rgb[i][j] =
                green;
            if (sobel_data[i][j].Gh < 0 && sobel_data[i][j].Gv <
0) rgb[i][j] =
                blue;
            if (sobel_data[i][j].Gh > 0 && sobel_data[i][j].Gv <
0) rgb[i][j] =
                white;
        }
    }
    return rgb;
}

vector<vector<double>> doGammaConversion(vector<vector<double>>& rgb,
double c, double g, int height, int width) {
    vector<vector<double>> tmp(height);
    for (size_t i = 0; i < height; i++) {
        for (size_t j = 0; j < width; j++) {
            double res = rgb[i][j];
            res /= 255;
            res = c * pow(res, g);
            res *= 255;
            tmp[i].push_back(clipping(res));
        }
    }
    return tmp;
}

vector<vector<double>> doHistogramAlignment(vector<vector<double>>&
rgb, int height, int width) {
    vector<vector<double>> mas(height);
    const size_t N = height * width;
    vector<pair<unsigned char, size_t>> freq(256);
    for (size_t i = 0; i < 256; i++) {

```

```

        pair<unsigned char, size_t> tmp(i, 0);
        freq.push_back(tmp);
    }
    for (size_t i = 0; i < height; i++) {
        for (size_t j = 0; j < width; j++) {
            freq[rgb[i][j]].second++;
        }
    }
    vector<unsigned char> lookup_table;
    for (size_t i = 0; i < 256; i++) {
        double tmp = 0;
        for (size_t j = 0; j <= i; j++) {
            tmp += freq[j].second;
        }
        tmp = tmp * 255 / N;
        lookup_table.push_back(clipping(tmp));
    }
    for (size_t i = 0; i < height; i++) {
        for (size_t j = 0; j < width; j++) {
            mas[i].push_back(lookup_table[rgb[i][j]]);
        }
    }
    return mas;
}

vector<vector<double>> doBinaryImage(vector<vector<double>>& rgb, unsigned char T, int h, int w) {
    vector<vector<double>> tmp(h);
    for (size_t i = 0; i < h; i++) {
        for (size_t j = 0; j < w; j++) {
            if (rgb[i][j] < T) tmp[i].push_back(255);
            else tmp[i].push_back(0);
        }
    }
    return tmp;
}

int main() {
    MBITMAPFILEHEADER bfh;
    MBITMAPINFOHEADER bih;
    FILE* f = fopen("myImage.bmp", "rb");
    if (f == NULL) {
        cout << "reading error";
        return 0;
    }
    MRGBQUAD** rgb = readBmp(f, &bfh, &bih);
    fclose(f);
    int height = abs(bih.biHeight);
    int width = abs(bih.biWidth);

    vector<vector<double>> Y(height);
    vector<vector<double>> Cb(height);
    vector<vector<double>> Cr(height);

```

```

        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                Y[i].push_back((double)rgb[i][j].rgbRed * 0.299 +
(double)rgb[i][j].rgbGreen * 0.587 + (double)rgb[i][j].rgbBlue * 0.114);
                Cb[i].push_back(0.5643 * ((double)rgb[i][j].rgbBlue - Y[i][j]) + 128);
                Cr[i].push_back(0.7132 * ((double)rgb[i][j].rgbRed - Y[i][j]) + 128);
            }
        }

        findMinMax(rgb, height, width);
        srand(time(NULL));

        vector<vector<double>> gausY = generateGaussianNoiseV2(Y,
height, width, 30); // 1
        rgb = getRGBfromY(gausY, height, width);
        f = fopen("additiveNoise.bmp", "wb");
        writeBMP(f, rgb, &bfh, &bih, height, width);
        fclose(f);

        vector<vector<double>> impulseY = generateImpulseNoise(Y,
height, width, 0.25, 0.25); // 2
        rgb = getRGBfromY(impulseY, height, width);
        f = fopen("impulseNoise.bmp", "wb");
        writeBMP(f, rgb, &bfh, &bih, height, width);
        fclose(f);

        //buildGraphicsPSNR(Y); //3

        //vector<vector<double>> movingAverageY = generateMovingAver-
age(gausY, 1); // 4.1, 4.2
        //rgb = getRGBfromY(movingAverageY, height, width);
        //cout << "\n" << "old PSNR: " << calculatePSNR(Y, gausY);
        //cout << "\n" << "new PSNR: " << calculatePSNR(Y, movingAver-
ageY) << endl;
        //f = fopen("gaussianNoiseMovingAverage.bmp", "wb");
        //writeBMP(f, rgb, &bfh, &bih, height, width);
        //fclose(f);
        //printSigmaPsnrR(Y);

        //vector<vector<double>> gausFilterY = doGaussianFilter(gausY,
3, 1); // 4.3, 4.4
        //rgb = getRGBfromY(gausFilterY, height, width);
        //cout << "\n" << "old PSNR: " << calculatePSNR(Y, gausY);
        //cout << "\n" << "new PSNR: " << calculatePSNR(Y, gausFilterY)
<< endl;
        //f = fopen("gaussianFilter.bmp", "wb");
        //writeBMP(f, rgb, &bfh, &bih, height, width);
        //fclose(f);

        //buildGaussianFilterGraphicsPSNR(Y); // 4.5

```

```

    //vector<vector<double>> medianFilterY = doMedianFilter(gausY,
2); //4.7
    //rgb = getRGBfromY(medianFilterY, height, width);
    //cout << "\n" << "old PSNR: " << calculatePSNR(Y, gausY);
    //cout << "\n" << "new PSNR: " << calculatePSNR(Y, medianFil-
terY) << endl;
    //f = fopen("medianFilter.bmp", "wb");
    //writeBMP(f, rgb, &bfh, &bih, height, width);
    //fclose(f);

    //printGausNoiseMedianFilterPSNR(Y); // 4.8

    compareFilters(Y, &bfh, &bih); // 4.6 и 4.9

    //vector<vector<double>> impulseY1 = generateImpulseNoise(Y,
height, width, 0.025, 0.025); //5.1
    //rgb = getRGBfromY(impulseY1, height, width);
    //f = fopen("impulseNoise0.025.bmp", "wb");
    //writeBMP(f, rgb, &bfh, &bih, height, width);
    //fclose(f);

    //vector<vector<double>> impulseY2 = generateImpulseNoise(Y,
height, width, 0.05, 0.05);
    //rgb = getRGBfromY(impulseY2, height, width);
    //f = fopen("impulseNoise0.05.bmp", "wb");
    //writeBMP(f, rgb, &bfh, &bih, height, width);
    //fclose(f);

    //vector<vector<double>> impulseY3 = generateImpulseNoise(Y,
height, width, 0.125, 0.125);
    //rgb = getRGBfromY(impulseY3, height, width);
    //f = fopen("impulseNoise0.125.bmp", "wb");
    //writeBMP(f, rgb, &bfh, &bih, height, width);
    //fclose(f);

    //vector<vector<double>> impulseY4 = generateImpulseNoise(Y,
height, width, 0.25, 0.25);
    //rgb = getRGBfromY(impulseY4, height, width);
    //f = fopen("impulseNoise0.25.bmp", "wb");
    //writeBMP(f, rgb, &bfh, &bih, height, width);
    //fclose(f);

    //cout << "\n" << "5% PSNR = " << calculatePSNR(Y, impulseY1);
// 5.2
    //cout << "\n" << "10% PSNR = " << calculatePSNR(Y, impulseY2);
    //cout << "\n" << "25% PSNR = " << calculatePSNR(Y, impulseY3);
    //cout << "\n" << "50% PSNR = " << calculatePSNR(Y, impulseY4)
<< "\n";

    //rgb = getRGBfromY(doMedianFilter(impulseY1, 1), height,
width); //5. 3

```

```

//f = fopen("impulseNoiseMedianFilterPSNR5.bmp", "wb");
//writeBMP(f, rgb, &bfh, &bih, height, width);
//fclose(f);

//rgb = getRGBfromY(doMedianFilter(impulseY2, 1), height,
width);
//f = fopen("impulseNoiseMedianFilterPSNR10.bmp", "wb");
//writeBMP(f, rgb, &bfh, &bih, height, width);
//fclose(f);

//rgb = getRGBfromY(doMedianFilter(impulseY3, 2), height,
width);
//f = fopen("impulseNoiseMedianFilterPSNR25.bmp", "wb");
//writeBMP(f, rgb, &bfh, &bih, height, width);
//fclose(f);

//rgb = getRGBfromY(doMedianFilter(impulseY4, 3), height,
width);
//f = fopen("impulseNoiseMedianFilterPSNR50.bmp", "wb");
//writeBMP(f, rgb, &bfh, &bih, height, width);
//fclose(f);
//



//printImpulseNoiseMedianFilterPSNR(Y);

//2.2

//f = fopen("laplasianImage1.bmp", "wb");
//writeBMP(f, getRGBfromY(laplasian(Y, height, width, 0),
height, width), &bfh, &bih, height, width);
//fclose(f);

//f = fopen("laplasianImage2.bmp", "wb");
//writeBMP(f, getRGBfromY(laplasian(Y, height, width, 128),
height, width), &bfh, &bih, height, width);
//fclose(f);

//f = fopen("laplasianImage3.bmp", "wb");
//writeBMP(f, getRGBfromY(laplasian2(Y, height, width, 0),
height, width), &bfh, &bih, height, width);
//fclose(f);

//f = fopen("laplasianImage4.bmp", "wb");
//writeBMP(f, getRGBfromY(laplasian2(Y, height, width, 128),
height, width), &bfh, &bih, height, width);
//fclose(f);

////1.3-1.5
//f = fopen("laplasianImage1,3.bmp", "wb");
//writeBMP(f, getRGBfromY(laplasianAlpha( Y, height, width),
height, width), &bfh, &bih, height, width);
//fclose(f);

```

```

    //vector<vector<double>> Ylap1 = laplasianAlpha2(Y, 1, height,
width, 0);
    //f = fopen("laplasianImageAlpha1,0.bmp", "wb");
    //writeBMP(f, getRGBfromY(Ylap1, height, width), &bfh, &bih,
height, width);
    //fclose(f);

    //vector<vector<double>> Ylap2 = laplasianAlpha2(Y, 1.1, height,
width, 0);
    //f = fopen("laplasianImageAlpha1,1.bmp", "wb");
    //writeBMP(f, getRGBfromY(Ylap2, height, width), &bfh, &bih,
height, width);
    //fclose(f);

    //vector<vector<double>> Ylap3 = laplasianAlpha2(Y, 1.2, height,
width, 0);
    //f = fopen("laplasianImageAlpha1,2.bmp", "wb");
    //writeBMP(f, getRGBfromY(Ylap3, height, width), &bfh, &bih,
height, width);
    //fclose(f);

    //vector<vector<double>> Ylap4 = laplasianAlpha2(Y, 1.3, height,
width, 0);
    //f = fopen("laplasianImageAlpha1,3.bmp", "wb");
    //writeBMP(f, getRGBfromY(Ylap4, height, width), &bfh, &bih,
height, width);
    //fclose(f);

    //vector<vector<double>> Ylap5 = laplasianAlpha2(Y, 1.4, height,
width, 0);
    //f = fopen("laplasianImageAlpha1,4.bmp", "wb");
    //writeBMP(f, getRGBfromY(Ylap5, height, width), &bfh, &bih,
height, width);
    //fclose(f);

    //vector<vector<double>> Ylap6 = laplasianAlpha2(Y, 1.5, height,
width, 0);
    //f = fopen("laplasianImageAlpha1,5.bmp", "wb");
    //writeBMP(f, getRGBfromY(Ylap6, height, width), &bfh, &bih,
height, width);
    //fclose(f);

    //writeBMP(fopen("laplasiankk.bmp", "wb"), getRGBfromY(lapla-
sian2(Y, 0, height, width), height, width), &bfh, &bih, height,
width);

    //cout << "Alpha = 1.0: " << calculateAverageBright(Ylap1) <<
endl;
    //cout << "Alpha = 1.1: " << calculateAverageBright(Ylap2) <<
endl;
    //cout << "Alpha = 1.2: " << calculateAverageBright(Ylap3) <<
endl;
    //cout << "Alpha = 1.3: " << calculateAverageBright(Ylap4) <<
endl;

```

```

    //cout << "Alpha = 1.4: " << calculateAverageBright(Ylap5) <<
endl;
    //cout << "Alpha = 1.5: " << calculateAverageBright(Ylap6) <<
endl;
    //
    //writeFile("l/laplasicOriginal.txt", Y, height, width);
    //writeFile("l/laplasicAlpha1,0.txt", Ylap1, height, width);
    //writeFile("l/laplasicAlpha1,1.txt", Ylap2, height, width);
    //writeFile("l/laplasicAlpha1,2.txt", Ylap3, height, width);
    //writeFile("l/laplasicAlpha1,3.txt", Ylap4, height, width);
    //writeFile("l/laplasicAlpha1,4.txt", Ylap5, height, width);
    //writeFile("l/laplasicAlpha1,5.txt", Ylap6, height, width);

    //doSobel
    //writeBMP(fopen("s/sobel.bmp", "wb"), getRGBfromY(doSobel(Y,
127, height, width), height, width), &bfh, &bih, height, width);

    //writeBMP(fopen("s/sobel30.bmp", "wb"), getRGBfromY(doSobel(Y,
30, height, width), height, width), &bfh, &bih, height, width);
    //
    //writeBMP(fopen("s/sobel60.bmp", "wb"), getRGBfromY(doSobel(Y,
60, height, width), height, width), &bfh, &bih, height, width);

    //writeBMP(fopen("s/sobel90.bmp", "wb"), getRGBfromY(doSobel(Y,
90, height, width), height, width), &bfh, &bih, height, width);

    //writeBMP(fopen("s/sobel120.bmp", "wb"), getRGBfromY(doSobel(Y,
120, height, width), height, width), &bfh, &bih, height, width);

    //writeBMP(fopen("s/sobel150.bmp", "wb"), getRGBfromY(doSobel(Y,
150, height, width), height, width), &bfh, &bih, height, width);

    //writeBMP(fopen("s/sobel180.bmp", "wb"), getRGBfromY(doSobel(Y,
180, height, width), height, width), &bfh, &bih, height, width);
    //
    vector<vector<Sobel>> sobelData = doSobelParametr(Y, 120,
height, width);
    writeBMP(fopen("s/sobelMap.bmp", "wb"), calculateGradient(so-
belData, height, width), &bfh, &bih, height, width);

    vector<vector<double>> Ydark = changeBrightness(Y, 70, 1,
height, width);
    writeBMP(fopen("gradation/dark.bmp", "wb"), getRGBfromY(Ydark,
height, width), &bfh, &bih, height, width);

    vector<vector<double>> Ylight = changeBrightness(Y, 70, 0,
height, width);
    writeBMP(fopen("gradation/light.bmp", "wb"), getRGBfromY(Ylight,
height, width), &bfh, &bih, height, width);

    vector<vector<double>> YtwoPoints = two_points(Y, 65, 40, 180,
210, height, width);
    writeBMP(fopen("gradation/2points.bmp", "wb"), getRGBfromY(Ytwo-
Points, height, width), &bfh, &bih, height, width);

```

```

    vector<vector<double>> YtwoPoints1 = two_points(Ylight, 120, 20,
210, 180, height, width);
    writeBMP(fopen("gradation/2pointsLight.bmp", "wb"),
getRGBfromY(YtwoPoints1, height, width), &bfh, &bih, height, width);

    vector<vector<double>> YtwoPoints2 = two_points(Ydark, 100, 160,
140, 220, height, width);
    writeBMP(fopen("gradation/2pointsDark.bmp", "wb"),
getRGBfromY(YtwoPoints2, height, width), &bfh, &bih, height, width);

    writeFile("gradation/isx.txt", Y, height, width);
    writeFile("gradation/dark.txt", Ydark, height, width);
    writeFile("gradation/light.txt", Ylight, height, width);
    writeFile("gradation/2pointsDark.txt", YtwoPoints2, height,
width);
    writeFile("gradation/2pointsLight.txt", YtwoPoints1, height,
width);
    writeFile("gradation/2pointsIsx.txt", YtwoPoints, height,
width);

    //vector<vector<double>> Ygamma = doGammaConversion(Y, 1, 0.1,
height, width);
    //writeBMP(fopen("gradation/gamma/gamma(0.1).bmp", "wb"),
getRGBfromY(Ygamma, height, width), &bfh, &bih, height, width);
    //

    //vector<vector<double>> Ygamma1 = doGammaConversion(Y, 1, 0.5,
height, width);
    //writeBMP(fopen("gradation/gamma/gamma(0.5).bmp", "wb"),
getRGBfromY(Ygamma1, height, width), &bfh, &bih, height, width);

    //vector<vector<double>> Ygamma2 = doGammaConversion(Y, 1, 1,
height, width);
    //writeBMP(fopen("gradation/gamma/gamma(1.0).bmp", "wb"),
getRGBfromY(Ygamma2, height, width), &bfh, &bih, height, width);

    //vector<vector<double>> Ygamma3 = doGammaConversion(Y, 1, 2,
height, width);
    //writeBMP(fopen("gradation/gamma/gamma(2.0).bmp", "wb"),
getRGBfromY(Ygamma3, height, width), &bfh, &bih, height, width);

    //vector<vector<double>> Ygamma4 = doGammaConversion(Y, 1, 8,
height, width);
    //writeBMP(fopen("gradation/gamma/gamma(8.0).bmp", "wb"),
getRGBfromY(Ygamma4, height, width), &bfh, &bih, height, width);

    //vector<vector<double>> YgammaLight = doGammaConversion(Ylight,
1, 0.1, height, width);
    //writeBMP(fopen("gradation/gamma/gammaLight(0.1).bmp", "wb"),
getRGBfromY(YgammaLight, height, width), &bfh, &bih, height, width);

    //vector<vector<double>> YgammaLight1 = doGammaConver-
sion(Ylight, 1, 0.5, height, width);

```

```

//writeBMP(fopen("gradation/gamma/gammaLight(0.5).bmp", "wb"),
getRGBfromY(YgammaLight1, height, width), &bfh, &bih, height, width);

//vector<vector<double>> YgammaLight2 = doGammaConversion(Ylight, 1, 1, height, width);
//writeBMP(fopen("gradation/gamma/gammaLight(1.0).bmp", "wb"),
getRGBfromY(YgammaLight2, height, width), &bfh, &bih, height, width);

//vector<vector<double>> YgammaLight3 = doGammaConversion(Ylight, 1, 2, height, width);
//writeBMP(fopen("gradation/gamma/gammaLight(2.0).bmp", "wb"),
getRGBfromY(YgammaLight3, height, width), &bfh, &bih, height, width);

//vector<vector<double>> YgammaLight4 = doGammaConversion(Ylight, 1, 8.0, height, width);
//writeBMP(fopen("gradation/gamma/gammaLight(8.0).bmp", "wb"),
getRGBfromY(YgammaLight4, height, width), &bfh, &bih, height, width);

//vector<vector<double>> YgammaDark = doGammaConversion(Ydark, 1, 0.1, height, width);
//writeBMP(fopen("gradation/gamma/gammaDark(0.1).bmp", "wb"),
getRGBfromY(YgammaDark, height, width), &bfh, &bih, height, width);

//vector<vector<double>> YgammaDark1 = doGammaConversion(Ydark, 1, 0.5, height, width);
//writeBMP(fopen("gradation/gamma/gammaDark(0.5).bmp", "wb"),
getRGBfromY(YgammaDark1, height, width), &bfh, &bih, height, width);

//vector<vector<double>> YgammaDark2 = doGammaConversion(Ydark, 1, 1, height, width);
//writeBMP(fopen("gradation/gamma/gammaDark(1.0).bmp", "wb"),
getRGBfromY(YgammaDark2, height, width), &bfh, &bih, height, width);

//vector<vector<double>> YgammaDark3 = doGammaConversion(Ydark, 1, 2, height, width);
//writeBMP(fopen("gradation/gamma/gammaDark(2.0).bmp", "wb"),
getRGBfromY(YgammaDark3, height, width), &bfh, &bih, height, width);

//vector<vector<double>> YgammaDark4 = doGammaConversion(Ydark, 1, 8, height, width);
//writeBMP(fopen("gradation/gamma/gammaDark(8.0).bmp", "wb"),
getRGBfromY(YgammaDark4, height, width), &bfh, &bih, height, width);

//writeFile("gradation/gamma/i.txt", Ygamma, height, width);
//writeFile("gradation/gamma/i1.txt", Ygamma1, height, width);
//writeFile("gradation/gamma/i2.txt", Ygamma2, height, width);
//writeFile("gradation/gamma/i3.txt", Ygamma3, height, width);
//writeFile("gradation/gamma/i4.txt", Ygamma4, height, width);

//writeFile("gradation/gamma/light.txt", YgammaLight, height, width);
//writeFile("gradation/gamma/light1.txt", YgammaLight1, height, width);

```

```

        //writeFile("gradation/gamma/light2.txt", YgammaLight2, height,
width);
        //writeFile("gradation/gamma/light3.txt", YgammaLight3, height,
width);
        //writeFile("gradation/gamma/light4.txt", YgammaLight4, height,
width);

        //writeFile("gradation/gamma/dark.txt", YgammaDark, height,
width);
        //writeFile("gradation/gamma/dark1.txt", YgammaDark1, height,
width);
        //writeFile("gradation/gamma/dark2.txt", YgammaDark2, height,
width);
        //writeFile("gradation/gamma/dark3.txt", YgammaDark3, height,
width);
        //writeFile("gradation/gamma/dark4.txt", YgammaDark4, height,
width);

        //vector<vector<double>> Ygist = doHistogramAlignment(Y, height,
width);
        //writeBMP(fopen("gradation/gistIsx.bmp", "wb"),
getRGBfromY(Ygist, height, width), &bfh, &bih, height, width);
        //
        //vector<vector<double>> YgistLight = doHistogramAlignment(Ylight, height, width);
        //writeBMP(fopen("gradation/gistLight.bmp", "wb"),
getRGBfromY(YgistLight, height, width), &bfh, &bih, height, width);

        //vector<vector<double>> YgistDark = doHistogramAlignment(Ydark, height, width);
        //writeBMP(fopen("gradation/gistDark.bmp", "wb"),
getRGBfromY(YgistDark, height, width), &bfh, &bih, height, width);

        //writeFile("gradation/gistIsx.txt", Ygist, height, width);
        //writeFile("gradation/gistDark.txt", YgistDark, height, width);
        //writeFile("gradation/gistLight.txt", YgistLight, height, width);

        //vector<vector<double>> Y_T = doBinaryImage(Y, 16, height,
width);
        //writeBMP(fopen("gradation/binary/binaryT16.bmp", "wb"),
getRGBfromY(Y_T, height, width), &bfh, &bih, height, width);

        //vector<vector<double>> Y_T1 = doBinaryImage(Y, 48, height,
width);
        //writeBMP(fopen("gradation/binary/binaryT48.bmp", "wb"),
getRGBfromY(Y_T1, height, width), &bfh, &bih, height, width);

        //vector<vector<double>> Y_T2 = doBinaryImage(Y, 80, height,
width);
        //writeBMP(fopen("gradation/binary/binaryT80.bmp", "wb"),
getRGBfromY(Y_T2, height, width), &bfh, &bih, height, width);
        //

```

```

    //vector<vector<double>> Y_T3 = doBinaryImage(Y, 112, height,
width);
    //writeBMP(fopen("gradation/binary/binaryT112.bmp", "wb"),
getRGBfromY(Y_T3, height, width), &bfh, &bih, height, width);

    //vector<vector<double>> Y_T4 = doBinaryImage(Y, 144, height,
width);
    //writeBMP(fopen("gradation/binary/binaryT144.bmp", "wb"),
getRGBfromY(Y_T4, height, width), &bfh, &bih, height, width);

    //vector<vector<double>> Y_T5 = doBinaryImage(Y, 176, height,
width);
    //writeBMP(fopen("gradation/binary/binaryT176.bmp", "wb"),
getRGBfromY(Y_T5, height, width), &bfh, &bih, height, width);

    //vector<vector<double>> Y_T6 = doBinaryImage(Y, 208, height,
width);
    //writeBMP(fopen("gradation/binary/binaryT208.bmp", "wb"),
getRGBfromY(Y_T6, height, width), &bfh, &bih, height, width);

    //vector<vector<double>> Y_T7 = doBinaryImage(Y, 240, height,
width);
    //writeBMP(fopen("gradation/binary/binaryT240.bmp", "wb"),
getRGBfromY(Y_T7, height, width), &bfh, &bih, height, width);

    return 0;
}

```