Министерство образования и науки РФ
ФГБОУ ВО «Тверской государственный университет»
Факультет прикладной математики и кибернетики
Направление «Фундаментальная информатика и информационные технологии»
Профиль «Инженерия программного обеспечения»

ВЫПУСКНАЯ РАБОТА БАКАЛАВРА

"Метод восстановления пространственного разрешения изображений"

Автор:

Логинов Денис Александрович

Научный руководитель:

кандидат физико-математических наук, доцент

Семёнов Андрей Борисович

Допущен (а) к защите:

Руководитель ООП:

_____

(подпись, дата)

Заведующий кафедрой: _____

(наименование)

_____

(подпись, дата)

Тверь 2018

Оглавление

Введение

Актуальность

Темой данной выпускной работы (сокращенно ВР) является метод восстановления пространственного разрешения изображения. Эта тема актуальна во многих аспектах фото- и видеосъёмки, начиная от старых камер мобильных телефонов, имеющих низкое разрешение изображения, и заканчивая космическими снимками со спутников.

Остановимся немного на спутниковой съёмке. В текущих реалиях максимальный аппаратный предел коммерческой спутниковой съёмки составляет 25 сантиметров на пиксель. Для сравнения, на изображении размером 7x4 пикселя при данном масштабе можно поместить взрослый горный велосипед, положенный на бок, а на изображении размером 17x7 при всё том же масштабе - автомобиль средних размеров. Экспертная оценка качества изображения в данной ситуации - это глаза человека, но при таком масштабе изображение будет слишком маленьким для того, чтобы глаз человека воспринял его полностью. Логичным действием в данном случае будет увеличение разрешения изображения (ресемплинга, передискретизации) для лучшего восприятия. Тут мы и подходим к сути ВР. При стандартном способе увеличения разрешения изображение получается слишком "квадратизированным" или "пикселизированным", то есть состоящим из больших одноцветных квадратов, что сильно сказывается на его качестве.

Цель работы

Цель данной ВКР заключается в создании инструмента, позволяющего изменять размер изображения с минимальными потерями качества. Для более точного описания ВКР необходимо уточнить, что в рамках данной ВКР планируется реализовать не один метод восстановления пространственного разрешения изображения, а разработать целое приложение с графическим

интерфейсом и набором инструментов для вышеописанных целей, в частности, для обработки спутниковых изображений. В качестве инструментов преимущественно будут выступать фильтры свёртки. Также в приложении будут присутствовать иные фильтры и способы обработки изображений.

Задачи работы

В ходе составления ВКР были поставлены следующие задачи:

- Изучение предметной области;
- Анализ литературы по теме исследования;
- Разработка алгоритмов пространственного разрешения изображения;
- Разработка и реализация графического интерфейса приложения;
- Программная реализация алгоритмов;
- Проведение вычислительных экспериментов;
- Анализ полученных результатов;
- Составление текста выпускной курсовой работы.

Структура основной части работы

Основная часть выпускной курсовой работы содержит:

- Список направлений деятельности;
- Краткий обзор основных терминов и методов, используемых в работе;
- Описание терминов, используемых в работе;
- Описание методов, реализованных в программной части работы;
- Изложение практических результатов применения реализованных методов;
- Пояснение о пределе качества интерполяции;

- Рекомендации по модификации программной части работы;

- Памятку по применению разработанных методов.

В описании терминов, используемых в работе, для каждого термина содержится определение данного термина, а также наглядная иллюстрация данного термина на примере.

В описании методов содержится для каждого метода:

- Описание элемента на графическом интерфейсе, за которым закреплён метод;

- Ссылка на использованную для этого метода литературу;

- Описание используемых методом параметров;

- Описание принципа работы метода;

- Описание результата работы метода;

- Вспомогательные функции, используемые методом;

- Изображение с результатом работы метода.

Если один из вышеприведённых пунктов в конкретном описании метода отсутствует, то его не существует (например, отсутствие ссылки на литературу для разработанных автором методов, которых нет в литературе)

1. Средства разработки

В программной части работы графический интерфейс пользователя выполнен на интерфейсе программирования приложений Windows Forms и языке программирования C#. Были использованы пространства имён: System, System.Collections.Generic, System.ComponentModel, System.Data, System.Drawing, System.Linq, System.Text, System.Threading.Tasks, System.Windows.Forms.

Данные средства разработки были ипользованы по причине их свободного распространения, а также по причине простоты и ясности при работе с ними.

2. Изучение предметной области и анализ литературы по теме исследования

В ходе изучения предметной области были выявлены несколько основных направлений деятельности:

- Реализация методов интерполяции изображения;

- Реализация методов увеличения чёткости (уменьшения смазанности) изображения;

- Разработка и программная реализация эвристических алгоритмов изменения изображения;

- Реализация иных алгоритмов изменения изображения.

Для увеличения изображения с минимальной потерей качества необходимо понять, какие существуют способы для реализации данной цели, а также способы которые не влияют напрямую на качество изображения. Для этого обратимся к литературе. Следующий список показывает основные понятия, которые будут использоваться в данной ВКР при разработке алгоритмов: интерполяция, масштабирование, ресемплинг (англ. resampling), передискретизация, размытие, нерезкое маскирование, резкость, эрозия, наращивание, контраст, фильтрация, маска изображения, математическая морфология, интенсивность; а также основные алгоритмы: unsharp mask, unsharp filter, sharpen, high pass filter, nearest neighbour, bilinear, bicubic, Lanczos interpolation, median filter, erosion, build-up, smoothing, color filter, step filter, Kirsch, Sobel, Prewitt, Roberts filters.

Рассмотрим каждый элемент списка по отдельности.

Интенсивность. В обработке изображений интенсивность задаётся пикселям и обозначает яркость каждого из трёх каналов: красного, зелёного и синего. Интенсивность меняется в диапазоне от 0 до 255.

Фильтрация. В обработке изображений фильтрация - это способ изменения изображения по определённому алгоритму. Опорными данными в этом случае является исходное изображение, а выходными - изменённое изображение того же размера. Самая простая фильтрация - это свёртка по ядру. В этом случае алгоритм для каждого значения пикселя результирующего изображения, исходя из его (пикселя) координатного положения, берёт значения яркости пикселей исходного изображения в области, заданной ядром (фильтром), с заданными им же весами.

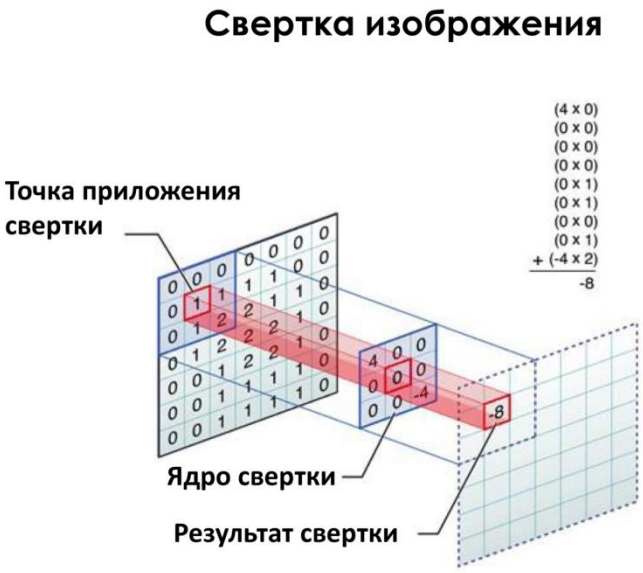

Рис. 1. Наглядный пример свёртки изображения по ядру.

Размытие изображения - смягчение границ переходов цвета за счёт усреднения соседних пикселов. Как следствие, происходит уменьшение резкости изображения.

Рис. 2. Слева - оригинальное изображение, справа - размытое.

Резкость изображения - характеристика изображения, показывающая степень резкости границ переходов цвета.

Рис. 3. Слева - оригинальное изображение, справа - с увеличенной резкостью.

Контрастность - близкое к резкости. Это степень разности в интенсивности, цвете и т. д. различных областей изображения относительно друг друга.

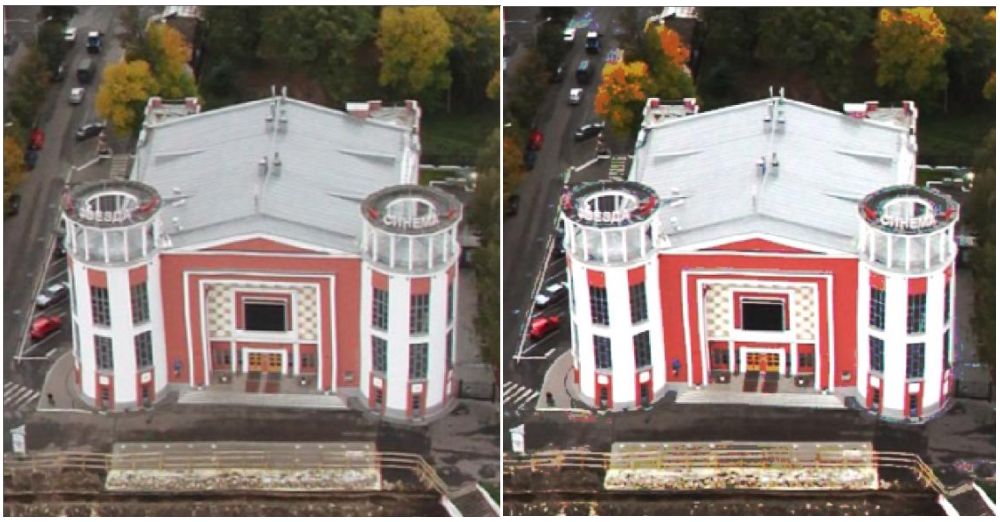

Рис. 4. Слева - оригинальное изображение, справа - с увеличенным контрастом.

Математическая морфология - теория и техника анализа и обработки геометрических структур, основанная на теории множеств, топологии и случайных функциях[11, стр. 31].

Эрозия - базовая операция математической морфологии. С точки зрения обработки изображений это фильтрация изображения по заданному булеву фильтру, при которой выбирается пиксель с наименьшей интенсивностью в области, ограниченной фильтром. После фильтрации в изображении подчёркиваются более тёмные элементы.

Рис. 5. Слева - оригинальное изображение, справа - с применением фильтра эрозия.

Наращивание - также, как и эрозия, это базовая операция математической морфологии. С точки зрения обработки изображений это фильтрация изображения по заданному булеву фильтру, при которой выбирается пиксель с наибольшей интенсивностью в области, ограниченной фильтром. После фильтрации в изображении подчёркиваются более светлые элементы.

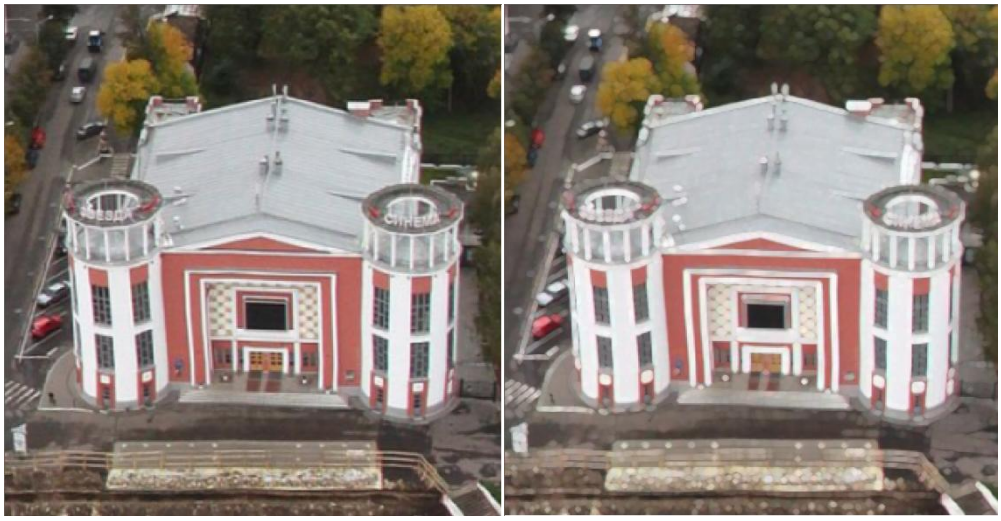

Рис. 6. Слева - оригинальное изображение, справа - с применением фильтра наращивание.

Маска изображения. Под маской изображения подразумевается изображение того же размера, что и оригинал, но при этом хранящее все границы

исходного изображения. Места с малыми перепадами цветов или их отсутствием более тёмные, нежели чёткие границы.

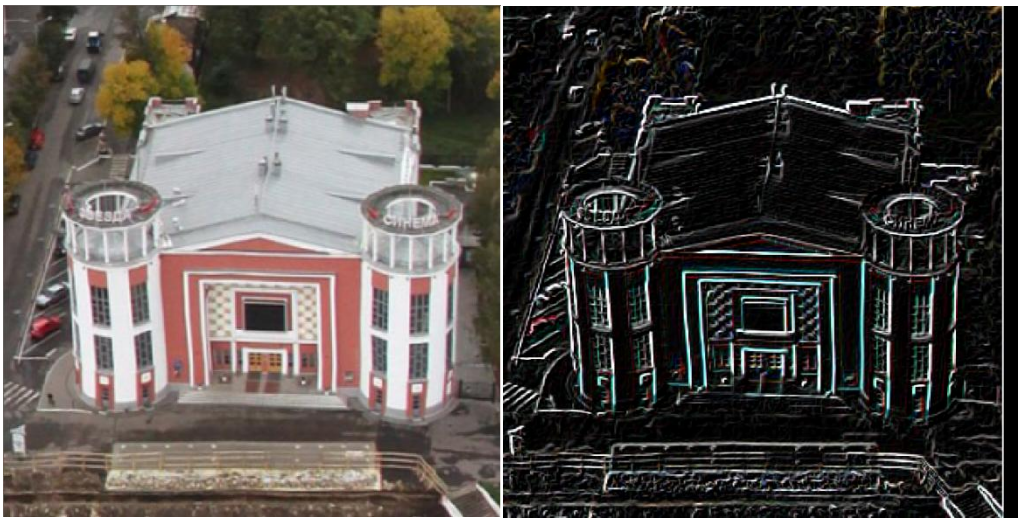

Рис. 7. Слева - оригинальное изображение, справа - маска оригинального изображения, полученная с помощью оператора Собеля.

Нерезкое маскирование - приём в обработки изображений, позволяющий увеличить резкость изображения за счёт усиления контраста тональных переходов.

Масштабирование - изменение размера изображения при сохранении его пропорций.

Рис. 8. Слева - оригинальное изображение, справа - масштабированное с помощью интерполяции: интерполированное.

Передискретизация, ресемплинг (англ. resampling) - то же, что и масштабирование за исключением того, что при передискретизации лишь изменяется размер исходного изображения без учёта пропорций. То есть масштабирование есть частный случай передискретизации.

Интерполяция есть способ нахождения промежуточных значений функции по уже имеющемуся набору значений. В обработке изображений интерполяция - это применение двумерной функции (фильтра) к исходному изображению. В качестве двумерной функции в данном случае выступают методы ближайшего соседа (nearest neighbour), билинейный (bilinear), бикубический (bicubic) и метод Ланцоша (Lanczos). Исходный (имеющийся) набор значений - это изображение с низким разрешением, а результат - изображение с высоким разрешением.

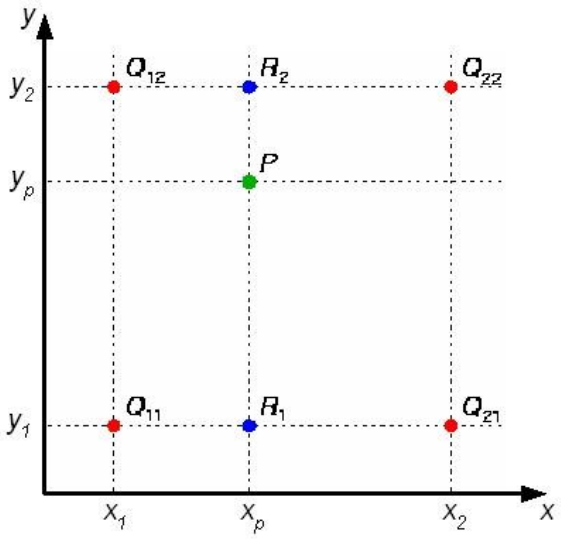

Рис. 9. Пример интерполирования по четырём точкам исходного изображения (метод билинейной интерполяции).

Зашумлённость изображения. Под зашумлённостью изображения подразумевается наличие искажений, либо неразличимых деталей, которые придают самому изображению зернистый вид. Выбор правильных

параметров для удаления шума очень важен, ведь большинство фильтров, избавляющих изображение от шумов, могут также избавить его и от необходимых границ.

Рис. 10. Слева - оригинальное изображение, справа - после избавления от шумов.

3. Структура программы

Программная реализация описанных ниже методов представляет из себя пространство имён Diplom с одним единственным классом Form1 и графический интерфейс - окно приложения с элементами управления на нём. Каждый элемент управления привязан к одному из методов класса Form1. В классе Form1 преимущественно содержатся методы обработки изображений и вспомогательные функции для подсчётов, но также в классе присутствуют и специальные структуры, такие, как массив, хранящий историю фильтрования. Если говорить об интерполяции, то интерполировать изображения можно на произвольный размер. Структура программы позволяет установить размер желаемого изображения.

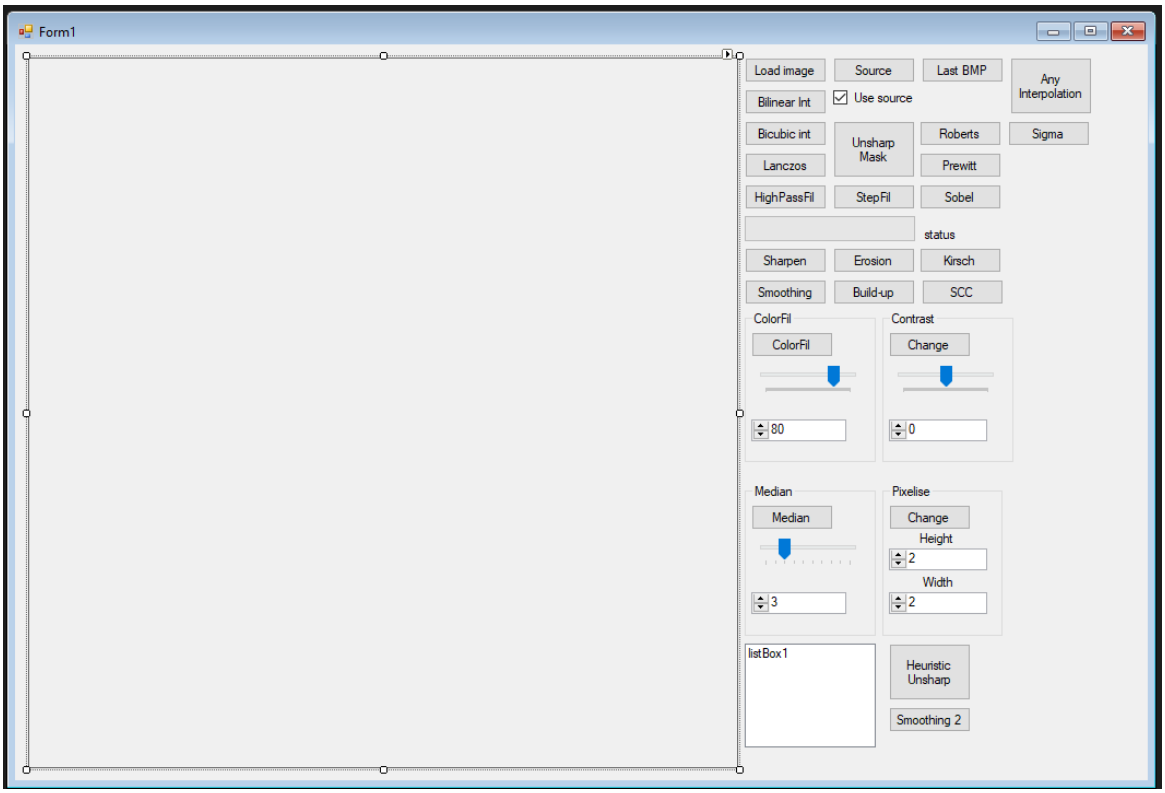

Рис. 11. Внешний вид графического интерфейса.

Слева располагается холст отрисовки входного изображения. Справа находятся кнопки, по нажатию которых выполняется алгоритм, указанный в их названии. Справа сверху имеется кнопка загрузки изображения с компьютера пользователя, ниже, шкала выполнения того или иного фильтра,

а также статус, указывающий, какой фильтр применяется в данный момент. В самом низу находится окно, указывающее, какие фильтры были применены. Пользователь может в любой момент откатить изменения на необходимое число шагов назад.

4. Реализованные методы

4.1. Метод resizenumber

Вспомогательный метод, высчитывающий коэффициент увеличения для последующего применения данного коэффициента в методах изменения размера изображений.

4.2. Метод Get_BMP

Данный метод представлен на форме кнопкой с названием source и используется при начальной загрузке изображения на холст отрисовки. Фактически, это стандартный метод nearest neighbour (метод ближайшего соседа), который берёт в качестве промежуточных пикселей ближайших к ним "соседей", см. [7, стр. 30]. С помощью этого метода изображение при увеличении становится "пикселизированным". Среди всех методов интерполяции этот - самый быстрый.

Рис. 12. Слева - оригинальное изображение, справа - увеличенное с помощью метода ближайшего соседа.

4.3. Метод BiLinear

Метод вызывается при нажатии на кнопку с названием Bilinear Int. Данный метод представляет собой метод билинейной интерполяции, см. [6, стр. 30]. Каждый промежуточный пиксель формируется исходя из четырех

ближайших из исходного изображения. По сравнению с методом ближайшего соседа данный даёт более плавные переходы в результирующем изображении, хотя "пикселизированность" всё равно остаётся. Особенно это заметно при увеличении изображений с низким разрешением.

Рис. 13. Слева - оригинальное изображение, справа - увеличенное с помощью метода билинейной интерполяции.

4.4. Метод BiCubic

Вызов метода осуществляется нажатием на кнопку с названием Bicubic int. Представляет собой метод бикубической интерполяции, см. [5, стр. 30]. В методе бикубической интерполяции промежуточные пиксели высчитываются по 16 ближайшим пикселам исходного изображения. Данный метод один из лучших методов для масштабирования изображения, хотя при увеличении резкости всё равно заметна сетка, оставшаяся от исходного изображения.

Рис. 14. Слева - оригинальное изображение, справа - увеличенное с помощью метода бикубической интерполяции с увеличением резкости.

4.5. Метод Lanczos

Данный метод вызывается нажатием на кнопку с названием этого же метода. Вспомогательные методы: sinc_x, impulse_characteristics_lanczos. Один из самых лучших и оптимальных методов, основанный на функции sinc_x, см. [19, стр. 31]. Параметр ***a***, задаваемый методу Lanczos, представляет собой размер окна, в пределах которого метод будет брать соседей искомого промежуточного пикселя из исходного изображения.

Применение этого фильтра позволяет добиться высокой чёткости изображения, но при обработке возможно появление нежелательных артефактов типа звона. Это искажение заключается в появлении вокруг контрастных переходов яркости узких контрастных ореолов, что позволяет сохранить резкость контрастных линий при сохранении достаточной гладкости тональных переходов.

При практической обработке изображений удовлетворительное качество достигается при значении параметра ***a*** равному 2 или 3 [19, стр. 31].

Рис. 15. Слева - оригинальное изображение, справа - увеличенное с помощью метода Ланцоша.

4.6. Метод Filter_Image

Общий метод для свёртки изображения по ядру, см. [13, стр. 31]. Метод получает в качестве параметров ядро свёртки и коэффициент нормирования. На форме он не представлен, но с его помощью реализованы другие методы, описанные ниже.

4.7. Метод HighPassFil

Вызывается нажатием на кнопку с названием HighPassFil. Метод представляет собой обычную свёртку

$$\text{изображения по ядру:} \quad \begin{matrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{matrix}$$

Частный случай общего метода Filter_Image. Результатом работы этого метода будет изображение повышенной чёткости, но при этом сильно зашумлённое, см. [13, стр. 31]. Это обусловлено тем, что метод увеличивает не только интенсивность пикселей на границах перехода цвета, но и все остальные пиксели. Также при обработке изображения этим методом могу появиться "битые" пиксели: пиксели на изображении, которые неестественно отличаются от пикселей исходного изображения в тех же координатах.

Рис. 16. Слева - оригинальное изображение, справа - обработанное с помощью метода HighPassFil.

4.8. Метод Sharpen

Кнопка с названием Sharpen. При её нажатии вызывается данный метод. Метод представляет собой обычную свёртку

$$\text{изображения по ядру: } \begin{matrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{matrix}$$

Также, как и high pass filter, частный случай общего метода Filter_Image. Этот метод позволяет повысить чёткость изображения но при этом количество шумов значительно ниже, нежели в high pass filter, см. [13, стр. 31].

4.9. Методы Smoothing и Gaussian_Blur

Для вызова методов используются кнопки с названиями Smoothing и Smoothing 2 соответственно. Данные методы позволяют сделать изображение более размытым, что часто бывает полезно при обработке изображений, см. [1, стр. 30]. Различия методов состоят в том, что smoothing использует для размытия метод Filter_Image с ядром свёртки 5x5, содержащем уже известные коэффициенты, найденные с помощью гауссовского распределения, а Gaussian_Blur предварительно подсчитывает матрицу коэффициентов произвольного размера на основе треугольника Паскаля.

Получившаяся матрица является приближением к матрице той же размерности, посчитанной с помощью гауссовского распределения.

Пример размытия с помощью Gaussian_Blur представлен в разделе 2 (*Рис. 2*).

4.10. Метод Median_Filter

Представлен кнопкой Median на форме и вызывается по её нажатию. В качестве параметра методу передаётся размер окна, по которому будет производиться фильтрация. Суть фильтрации заключается в следующем: для каждого пикселя в одномерный массив добавляются интенсивности пикселей вокруг него из заданного окна, см. [14, стр. 31]. После полученный массив сортируется и в качестве интенсивности текущего пикселя выбирается интенсивность из середины массива. Данная особенность алгоритма позволяет убрать шумы из изображения, но при неправильном подборе величины окна шумы будут убираться в ущерб важным деталям.

Рис. 17. Слева - оригинальное изображение, справа - обработанное с помощью метода Median_Filter.

4.11. Метод Sigma_Filter

Кнопка с названием Sigma на форме вызывает данный метод. Как и Median_Filter, Sigma_Filter позволяет избавиться от шумов на изображении, но в отличии от median он с большей долей вероятности сохраняет границы

объектов, см. [20, стр. 31]. На вход методу также, как в и median, подаётся размер окна, а также параметр сигма. Идея Sigma_Filter состоит в усреднении только тех яркостей в окне, которые отличаются от яркости центрального пикселя не более, чем на величину параметра сигма.

Рис. 18. Слева - оригинальное изображение, справа - обработанное с помощью метода Sigma_Filter. Здесь можно заметить, что границы на изображении менее размыты, а значит, сохранено больше деталей.

4.12. Методы Kirsch_Edging и SPR_Edging

Данные методы представляют собой четыре метода, вызываемых нажатием на кнопки формы с названиями Kirsch, Sobel, Prewitt и Roberts и объединённых одной идеей: выделение маски изображения и увеличение интенсивности пикселей на границах изображения, см. [2, 17, 16, 18, стр. 30-31]. У каждого метода примерно одинаковый метод выделения маски - фильтрация изображения через специальное(-ые) ядро(-а) свёртки:

Фильтр Кирша:

$$\mathbf{g}^{(1)} = \begin{bmatrix} +5 & +5 & +5 \\ -3 & 0 & -3 \\ -3 & -3 & -3 \end{bmatrix}, \mathbf{g}^{(2)} = \begin{bmatrix} +5 & +5 & -3 \\ +5 & 0 & -3 \\ -3 & -3 & -3 \end{bmatrix}, \mathbf{g}^{(3)} = \begin{bmatrix} +5 & -3 & -3 \\ +5 & 0 & -3 \\ +5 & -3 & -3 \end{bmatrix}, \mathbf{g}^{(4)} = \begin{bmatrix} -3 & -3 & -3 \\ +5 & 0 & -3 \\ +5 & +5 & -3 \end{bmatrix} \text{ and so on.}$$

Фильтр Собеля:

$$\mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A}$$

Фильтр Робертса:

$$\begin{bmatrix} +1 & 0 \\ 0 & -1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 0 & +1 \\ -1 & 0 \end{bmatrix}$$

Фильтр Превитта:

$$\mathbf{G_x} = \begin{bmatrix} -1 & 0 & +1 \\ -1 & 0 & +1 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G_y} = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ +1 & +1 & +1 \end{bmatrix} * \mathbf{A}$$

Маски изображений по Собелю, Превитту и Робертсу вычисляются по формуле:

$$\mathbf{G} = \sqrt{\mathbf{G}_x^{\,2} + \mathbf{G}_y^{\,2}}$$

Маска изображения по Киршу, напротив, вычисляется по все 8 маскам, высчитанным по данным ядрам свёртки. Из всех масок берётся пиксель с максимальной интенсивностью. В отличие от High Pass Filter и Sharpen данные методы позволяют увеличить интенсивность пикселей на границах, не сильно увеличивая при этом общую шумность изображения. Вспомогательные методы: Kirsch_Preparing, Sobel_Preparing, Prewitt_Preparing, Roberts_Preparing.

Рис. 19. Слева - оригинальное изображение, справа - обработанное с помощью метода Kirsch_Edging.

4.13. Методы Change_Contrast и Smart_Change_Contrast

Два метода увеличения или уменьшения контраста, которые представлены на форме как Change в рамке Contrast и SCC соответственно. Первый метод изменяет интенсивность пикселя по каждому из каналов на заданный параметр step так, что если интенсивность меньше средней по изображению, то она уменьшается на step, иначе увеличивается.

Второй метод растягивает гистограмму интенсивностей пикселей по следующему правилу: $\frac{Q-Q_1}{Q_2-Q_1}$, где Q_1 - левая граница задаваемого диапазона, Q_2 - правая.

Вспомогательные методы: Calculate_Intence, Normalize_Intence, SCC_Prepare.

Пример увеличения контраста методом Change_Contrast представлен в разделе 2 (*Рис. 4*).

4.14. Методы Erosion и Build-up

Данные методы вызываются кнопками с названиями Erosion и Build-up соответственно и программно объединены в один метод Erosion. Суть алгоритмов: для каждого пикселя изображения в окрестности, заданной булевой матрицей, выбирается пиксель с наибольшей (для Build-up) или

наименьшей (Erosion) интенсивностями и данная интенсивность присваивается центральному пикселю изображения, см. [13, стр. 31]. Как результат, изображение получается более тёмное или светлое на границах изображения. Такое свойство может быть полезно при комбинации со следующими фильтрами.

Пример работы методов Erosion и Build-up представлен в разделе 2 (*Рис. 5* и *Рис. 6*).

4.15. Методы Unsharp_Mask и Unsharp_Filter

Метод Unsharp Mask или "Нерезкое Маскирование" представлен на форме в виде двух разных кнопок с названиями Unsharp Mask и Heuristic Unsharp. Первый метод стандартный, а второй был написан в виде эвристики. Общая суть двух методов: Создаётся размытая копия исходного изображения (степень размытия задаётся в параметрах метода), см. [3, стр. 30]. После из исходного изображения вычитается размытое и получается маска. Создаётся копия исходного изображения с повышенным контрастом (коэффициент повышения контраста задаётся в параметрах метода). Если интенсивность маски превышает заданный порог, то в результирующее изображение добавляется пиксель из контрастного изображения, иначе - из исходного. Такой подход позволяет увеличивать интенсивность на границах изображения независимо от его чёткости, чего нельзя сказать о методах Кирша, Превитта, Собеля и Робертса: на нечётких изображениях они работают некорректно. Эвристический алгоритм Unsharp_Filter работает так же, как оригинальный Unsharp_Mask, до тех пор, пока метод не дойдёт до применения маски. В данном варианте маска увеличивается в интенсивности на коэффициент, заданный в параметрах, а после берётся интерполяция между интенсивностью пикселя исходного изображения и контрастного исходя из интенсивности маски.

Рис. 20. Слева - оригинальное нечёткое изображение, справа - обработанное с помощью метода Heuristic Unsharp.

4.16. Метод Color_Filter

Вызывается кнопкой с названием ColorFil. Данный эвристический метод подойдёт не для всех изображений, так как имеет узкую направленность.

Суть метода заключается в том, что для каждого пикселя результирующего изображения выбирается интенсивность самого близкого по интенсивности к исходному пиксела из набора самых популярных цветов изображения. Такой подход даёт более-менее чёткую границу в местах сильного размытия. В качестве параметра методу передаётся максимальное количество цветов в процентном соотношении.

Если говорить о применимости данного метода, то в основном его следует использовать на изображениях с низким разрешением, либо на малых частях больших изображений. Если применять данный метод к большим изображениям, то возможна потеря деталей на изображении. Параметр метода следует выбирать по следующим правилам: чем больше деталей на изображении - тем больше параметр. Чем меньше изображение - тем больше параметр. Чем больше изображение - тем меньше параметр. Вспомогательные методы: Statistics и Pixelize.

Рис. 21. Слева - оригинальное изображение, справа - обработанное с помощью метода Color_Filter с параметром 5. Можно заметить, что обработанное изображение потеряло много цветов.

4.17. Метод Pixelize

Вызывается кнопкой на форме с названием Pixelize. Позволяет получить меньшее по сравнению с исходным изображение. Используемый метод уменьшения - ближайший сосед.

4.18. Метод Step_Fil

Представлен на форме как StepFil. Параметр dispersion задаёт размер окна, а параметр step задаёт максимальную разницу в интенсивности, на которую может измениться пиксель. По методу каждый пиксель пытается найти в области, заданной окном, такой пиксел, разница с которым была бы максимальной. Данный метод является эвристическим и в теории позволяет несильно выделять границы, однако мелкие детали на изображении могут быть утрачены, а также могут появиться шумы.

Рис. 22. Слева - оригинальное изображение, справа - обработанное с помощью метода Step_Fil с параметром 10.

4.19 Метод Any_Interpolation_Heuristic

Данный метод вызывается нажатием на кнопку формы с названием AIH. Параметр dispersion задаёт размеры окна. Сам же метод представляет собой разработанный автором способ интерполяции изображения. При вычислении очередного промежуточного пикселя результирующего изображения вычисляются расстояния между положением промежуточного пикселя и положением всех пикселей исходного изображения в указанном окне и записываются в матрицу с обратным значением (единица, делёная на расстояние). То есть, чем дальше от промежуточного пикселя находится пиксель исходного изображения, тем меньшее значение будет записано в матрицу. После проводится нормировка всех элементов полученного массива: каждый элемент массива делится на сумму всех его элементов. Интенсивность промежуточного пикселя считается исходя из интенсивности пикселя исходного изображения в окне указанного размера, умноженного на соответствующее ему значение из рассчитанной матрицы. Итоговое изображение получается размытым, по сравнению с изображениями, интерполированными более эффективными методами (например, методом Ланцоша), следовательно, не является эффективным с точки зрения метода интерполяции. Но как вспомогательный метод может быть очень полезным

при восстановлении пространственного разрешения изображений с малым разрешением в случаях, когда требуется размытое изображение, например в методе Unsharp_Filter или методе Unsharp_Mask.

Рис. 23. Слева - оригинальное изображение, увеличенное методом ближайшего соседа, справа - обработанное с помощью метода Any_Interpolation_Heuristic с размером окна 2x2.

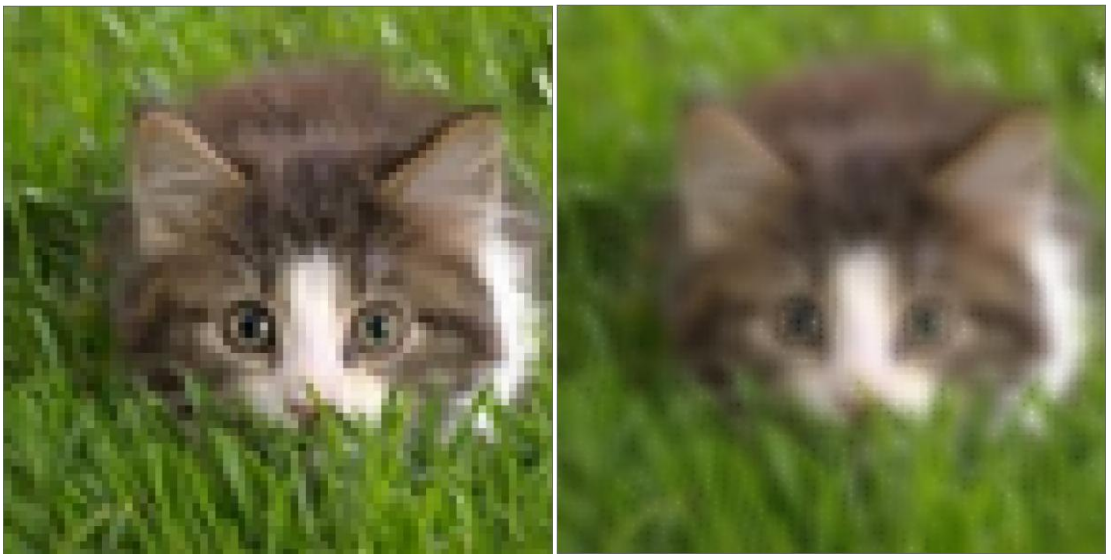

Рис. 24. Слева - оригинальное изображение, увеличенное методом ближайшего соседа, справа - обработанное с помощью метода Any_Interpolation_Heuristic с размером окна 4x4.

Рис. 25. Слева - оригинальное изображение, увеличенное методом ближайшего соседа, справа - обработанное с помощью метода Any_Interpolation_Heuristic с размером окна 6x6.

Рис. 26. Слева - оригинальное изображение, увеличенное методом Lanczos с параметром 3, справа - оно же, но с применением метода Unsharp_Filter. Размытое изображение в Unsharp_Filter получается с помощью метода Any_Interpolation_Heuristic с размером окна 6x6.

5. Результаты

5.1. Практическое применение

В ходе проведения вычислительных экспериментов разработанный комплекс инструментов был протестирован на реальных изображениях, в том числе и на спутниковых. Ниже приведён пример обработки одного из имеющихся спутниковых изображений. К исходному изображению были применены по порядку следующие методы с указанными параметрами: Lanczos(3), Sigma_Filter(3,10), Unsharp_Filter(100, 150, 7, 5, 3.5), Kirsch_Edging(30, 0.1), Sigma_Filter(3,10), Gaussian_Blur(3), Sharpen().

Рис. 22. Сверху - оригинальное изображение, увеличенное с помощью метода ближайшего соседа, Снизу - обработанное с помощью разработанного комплекса инструментов.

Рис. 23. Слева - часть крыла самолёта на оригинальном изображении, справа - часть крыла самолёта на обработанном изображении. На данных изображениях хорошо видна разница в качестве.

5.2. Ограничения интерполяции

Также в ходе проведения вычислительных экспериментов было установлено, что при интерполяции для достижения хорошего качества изображения необходимо устанавливать не слишком большой размер окна, по которому интерполируется пиксель (экспериментально - не более 7x7). Это обусловлено тем, что при нахождении очередного промежуточного пикселя при большом размере окна учитываются интенсивности пикселей во всём окне. Это приводит к размытию результирующего изображения, что негативно сказывается на его качестве.

5.3. Возможности по модификации

Данный вариант программы не конечный и может дополняться и модифицироваться. Вот краткий список того, что может быть привнесено в программу:

- реализация многопоточности;

- оптимизация структуры программы (рефакторинг кода, использование паттернов, особых структур и т. д.);

- автоматизация подбора фильтров и параметров к ним (например, с помощью нейронных сетей или частотного анализа).

5.4. Рекомендации к применению

В ходе проведения вычислительных экспериментов было установлено, что:

- методы Sharpen и HighPassFil следует применять лишь в том случае, если изображение несильно зашумлено или не зашумлено вовсе;

- для достаточно чёткого изображения эффективнее всего применять методы Roberts, Kirsch, Sobel и Prewitt;

- для недостаточно чёткого изображения необходимо применять методы Unsharp_Mask и Unsharp_Filter, причём Unsharp_Filter является более гибким для увеличения чёткости изображения, нежели Unsharp_Mask;

- для произвольного размытия эффективнее применять Gaussian_Blur;

- не допускать многократного применения фильтров Sharpen и High Pass Filter;

- для увеличения изображения с минимальными потерями качества и деталей использовать метод Lanczos с параметром 3;

- для удаления шумов на изображении и минимальной потерей деталей необходимо использовать метод Sigma_Filter;

- для более плавного увеличения контраста лучше использовать метод Smart_Change_Contrast;

- для изображений с низким разрешением, а также для смазанных изображений с низким количеством деталей можно использовать Color_Filter;

- для максимально возможного увеличения качества увеличенного изображения комбинировать все представленные фильтры.

Заключение

Целью данной ВКР являлось создание комплекса инструментов, позволяющего изменять размер изображения при минимальных потерях качества. В ходе выполнения ВКР был разработан необходимый комплекс инструментов, позволяющий обрабатывать изображения и восстанавливать их пространственное разрешение. Экспериментально было установлено, что интерполирование изображения не даёт максимально возможного качества изображения. На основании этого в программу были добавлены иные методы улучшения качества изображений, дающие изображению различные эффекты. Все поставленные задачи были выполнены в полном объёме.

Литература

1. Gaussian blur. – Режим доступа: https://en.wikipedia.org/wiki/Gaussian_blur;

2. Kirsch operator. – Режим доступа: https://en.wikipedia.org/wiki/Kirsch_operator;

3. Sharpening: unsharp mask. – Режим доступа: https://www.cambridgeincolour.com/tutorials/unsharp-mask.htm;

4. Алгоритмы масштабирования пиксельной графики. – Режим доступа: https://ru.wikipedia.org/wiki/Алгоритмы_масштабирования_пиксельной_графики;

5. Бикубическая интерполяция. – Режим доступа: https://ru.wikipedia.org/wiki/Бикубическая_интерполяция;

6. Билинейная интерполяция. – Режим доступа: https://ru.wikipedia.org/wiki/Билинейная_интерполяция;

7. Интерполяция цифрового изображения. – Режим доступа: https://www.cambridgeincolour.com/ru/tutorials-ru/image-interpolation.htm;

8. Никулин Е. А. – Компьютерная геометрия и алгоритмы машинной графики. – М.:БХВ-Петербург, 2003. – 560 стр.;

9. Шикин Е.В., Боресков А.В. Компьютерная графика. Динамика, реалистические изображения. – М.: Диалог-МИФИ, 1995. – 288 стр.;

10. Локальное улучшение контраста. – Режим доступа: https://www.cambridgeincolour.com/ru/tutorials-ru/local-contrast-enhancement.htm;

11. Математическая морфология. – Режим доступа: https://ru.wikipedia.org/wiki/Математическая_морфология;

12. Роджерс Д., Адамс Дж. Математические основы машинной графики. – М.: Мир, 2001. – 604 стр.;

13. Матричные фильтры обработки изображений. – Режим доступа: https://habr.com/post/142818;

14. Медианный фильтр. – Режим доступа: https://ru.wikipedia.org/wiki/Медианный_фильтр;

15. Нерезкое маскирование. – Режим доступа: https://ru.wikipedia.org/wiki/Нерезкое_маскирование;

16. Оператор Прюитт. – Режим доступа: https://ru.wikipedia.org/wiki/Оператор_Прюитт;

17. Оператор Собеля. – Режим доступа: https://ru.wikipedia.org/wiki/Оператор_Собеля;

18. Перекрёстный оператор Робертса. – Режим доступа: https://ru.wikipedia.org/wiki/Перекрёстный_оператор_Робертса;

19. Фильтр Ланцоша. – Режим доступа: https://ru.wikipedia.org/wiki/Фильтр_Ланцоша;

20. Эффективная фильтрация и выделение границ. – Режим доступа: http://www.irtc.org.ua/image/app/webroot/Files/presentations/Kovalevskiy/Kovalevski_Effiziente_Filterung_und_Kantendetektion_Kurz.pdf.

ПРИЛОЖЕНИЕ 1

Код программы

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Diplom
{
    public partial class Form1 : Form
    {
        OpenFileDialog open_dialog = new OpenFileDialog();
        Bitmap Load_bmp;
        Bitmap bmp;
        Graphics g, g1;
        Color col;
        float resize_number, resize_number_x, resize_number_y;
        bool mod_done, was_calc_intence, was_calc_cs;
        List<Bitmap> story;
        double mean_intence_R, mean_intence_G, mean_intence_B, mean_intence;
        List<Point> change_stat;
        Point work_point1, work_point2;
        bool l_m_d, r_m_d;
        Point move_point, last_move_point;
        int p_w, p_h;
        int pos_x, pos_y;

        public Form1()
        {
            InitializeComponent();

            bmp = new Bitmap(650, 650);
            pictureBox1.Image = new Bitmap(bmp.Width, bmp.Height);
            g = Graphics.FromImage(pictureBox1.Image);
            g1 = Graphics.FromImage(bmp);
            resize_number = 1;
            mod_done = true;
            story = new List<Bitmap>();
            was_calc_intence = false;
            p_w = pictureBox1.Width;
            p_h = pictureBox1.Height;
        }

        public void resizenumber(Bitmap In_BMP)
        {
            resize_number_x = (float)(In_BMP.Width) / (float)(bmp.Width);
            resize_number_y = (float)(In_BMP.Height) / (float)(bmp.Height);

            resize_number = Math.Max(resize_number_x, resize_number_y);
        }
        public void GetBmp(Bitmap In_BMP)    // обычный алгоритм масштабирования методом
ближайшего соседа
        {
            Graphics g1 = Graphics.FromImage(bmp);
```

```
            g1.Clear(Color.White);
            int w = (int)(In_BMP.Width / resize_number);
            int h = (int)(In_BMP.Height / resize_number);
            for (int x = 0; x < w; x++)
            {
                for (int y = 0; y < h; y++)
                {
                    col = In_BMP.GetPixel((int)(x * resize_number), (int)(y * resize_number));
                    bmp.SetPixel(x, y, Color.FromArgb(col.R, col.G, col.B));
                }
            }

        }
        public void BiLinear(Bitmap In_BMP)  // билинейная интреполяция входного изображения
        {
            int w = (int)(In_BMP.Width / resize_number);
            int h = (int)(In_BMP.Height / resize_number);
            int work_x, work_y;
            double lambda, myu;
            double w_x, w_y;
            Color w_c1, w_c2, w_c3, w_c4;
            int R, G, B;
            mod_done = false;

            for (int x = 0; x < w; x++)
            {
                for (int y = 0; y < h; y++)
                {
                    w_x = x * resize_number;
                    w_y = y * resize_number;
                    work_x = (int)Math.Floor(w_x);
                    work_y = (int)Math.Floor(w_y);
                    lambda = w_x - work_x;
                    myu = w_y - work_y;
                    R = 0;
                    G = 0;
                    B = 0;

                    if (work_x + 1 < In_BMP.Width && work_y + 1 < In_BMP.Height)
                    {
                        w_c1 = In_BMP.GetPixel(work_x, work_y);
                        w_c2 = In_BMP.GetPixel(work_x, work_y + 1);
                        w_c3 = In_BMP.GetPixel(work_x + 1, work_y);
                        w_c4 = In_BMP.GetPixel(work_x + 1, work_y + 1);

                        R = (int)(((1 - lambda) * ((1 - myu) * w_c1.R + myu * w_c2.R) + lambda
* ((1 - myu) * w_c3.R + myu * w_c4.R)) % 256);
                        G = (int)(((1 - lambda) * ((1 - myu) * w_c1.G + myu * w_c2.G) + lambda
* ((1 - myu) * w_c3.G + myu * w_c4.G)) % 256);
                        B = (int)(((1 - lambda) * ((1 - myu) * w_c1.B + myu * w_c2.B) + lambda
* ((1 - myu) * w_c3.B + myu * w_c4.B)) % 256);
                    }
                    if (work_x + 1 >= In_BMP.Width && work_y + 1 < In_BMP.Height)
                    {
                        w_c1 = In_BMP.GetPixel(work_x, work_y);
                        w_c2 = In_BMP.GetPixel(work_x, work_y + 1);
                        R = (int)((1 - myu) * w_c1.R + myu * w_c2.R);
                        G = (int)((1 - myu) * w_c1.G + myu * w_c2.G);
                        B = (int)((1 - myu) * w_c1.B + myu * w_c2.B);
                    }
                    if (work_x + 1 >= In_BMP.Width && work_y + 1 >= In_BMP.Height)
                    {
                        w_c1 = In_BMP.GetPixel(work_x, work_y);
                        R = w_c1.R;
                        G = w_c1.G;
                        B = w_c1.B;
                    }
                    if (work_x + 1 < In_BMP.Width && work_y + 1 >= In_BMP.Height)
```

```
                    {
                        w_c3 = In_BMP.GetPixel(work_x + 1, work_y);
                        w_c1 = In_BMP.GetPixel(work_x, work_y);
                        R = (int)(((1 - lambda) * (w_c1.R) + lambda * (w_c3.R)));
                        G = (int)(((1 - lambda) * (w_c1.G) + lambda * (w_c3.G)));
                        B = (int)(((1 - lambda) * (w_c1.B) + lambda * (w_c3.B)));
                    }

                    if (R > 255)
                    {
                        R = 255;
                    }
                    if (G > 255)
                    {
                        G = 255;
                    }
                    if (B > 255)
                    {
                        B = 255;
                    }


                    bmp.SetPixel(x, y, Color.FromArgb(R, G, B));
                }
                Progress_Func(x);
            }
            mod_done = true;
            Progress_Func(0);
        }

        public void BiCubic(Bitmap In_BMP)  // бикубическая интерполяция входного изображения
        {
            int w = (int)(In_BMP.Width / resize_number);
            int h = (int)(In_BMP.Height / resize_number);
            int work_x, work_y;
            double b1, b2, b3, b4, b5, b6, b7, b8, b9, b10, b11, b12, b13, b14, b15, b16;
            double lambda, myu;
            double w_x, w_y;
            Color w_c1, w_c2, w_c3, w_c4, w_c5, w_c6, w_c7, w_c8, w_c9, w_c10, w_c11, w_c12,
w_c13, w_c14, w_c15, w_c16;
            int R, G, B;
            mod_done = false;

            for (int x = 0; x < w; x++)
            {
                for (int y = 0; y < h; y++)
                {
                    w_x = x * resize_number;
                    w_y = y * resize_number;
                    work_x = (int)Math.Floor(w_x);
                    work_y = (int)Math.Floor(w_y);
                    lambda = w_x - work_x;
                    myu = w_y - work_y;

                    if (work_x + 2 < In_BMP.Width && work_y + 2 < In_BMP.Height && work_x > 0
&& work_y > 0)
                    {
                        b1 = 0.25 * (lambda - 1) * (lambda - 2) * (lambda + 1) * (myu - 1) *
(myu - 2) * (myu + 1);
                        b2 = -0.25 * lambda * (lambda - 2) * (lambda + 1) * (myu - 1) * (myu -
2) * (myu + 1);
                        b3 = -0.25 * myu * (lambda - 1) * (lambda - 2) * (lambda + 1) * (myu -
2) * (myu + 1);
                        b4 = 0.25 * lambda * myu * (lambda - 2) * (lambda + 1) * (myu - 2) *
(myu + 1);
                        b5 = ((double)-1 / 12) * lambda * (lambda - 1) * (lambda - 2) * (myu -
1) * (myu - 2) * (myu + 1);
```

```
                        b6 = ((double)-1 / 12) * (lambda - 1) * (lambda - 2) * (lambda + 1) *
(myu - 1) * (myu - 2) * myu;
                        b7 = ((double)1 / 12) * (lambda - 1) * (lambda - 2) * lambda * myu *
(myu - 2) * (myu + 1);
                        b8 = ((double)1 / 12) * lambda * (lambda - 2) * (lambda + 1) * (myu -
1) * (myu - 2) * myu;
                        b9 = ((double)1 / 12) * (lambda - 1) * lambda * (lambda + 1) * (myu -
1) * (myu - 2) * (myu + 1);
                        b10 = ((double)1 / 12) * (lambda - 1) * (lambda - 2) * (lambda + 1) *
(myu - 1) * myu * (myu + 1);
                        b11 = ((double)1 / 36) * (lambda - 1) * (lambda - 2) * lambda * (myu -
1) * (myu - 2) * myu;
                        b12 = ((double)-1 / 12) * (lambda - 1) * lambda * (lambda + 1) * myu *
(myu - 2) * (myu + 1);
                        b13 = ((double)-1 / 12) * lambda * (lambda - 2) * (lambda + 1) * (myu
- 1) * myu * (myu + 1);
                        b14 = ((double)-1 / 36) * (lambda - 1) * lambda * (lambda + 1) * (myu
- 1) * (myu - 2) * myu;
                        b15 = ((double)-1 / 36) * (lambda - 1) * (lambda - 2) * lambda * (myu
- 1) * myu * (myu + 1);
                        b16 = ((double)1 / 36) * (lambda - 1) * lambda * (lambda + 1) * (myu -
1) * myu * (myu + 1);

                        w_c1 = In_BMP.GetPixel(work_x, work_y);
                        w_c2 = In_BMP.GetPixel(work_x + 1, work_y);
                        w_c3 = In_BMP.GetPixel(work_x, work_y + 1);
                        w_c4 = In_BMP.GetPixel(work_x + 1, work_y + 1);
                        w_c5 = In_BMP.GetPixel(work_x - 1, work_y);
                        w_c6 = In_BMP.GetPixel(work_x, work_y - 1);
                        w_c7 = In_BMP.GetPixel(work_x - 1, work_y + 1);
                        w_c8 = In_BMP.GetPixel(work_x + 1, work_y - 1);
                        w_c9 = In_BMP.GetPixel(work_x + 2, work_y);
                        w_c10 = In_BMP.GetPixel(work_x, work_y + 2);
                        w_c11 = In_BMP.GetPixel(work_x - 1, work_y - 1);
                        w_c12 = In_BMP.GetPixel(work_x + 2, work_y + 1);
                        w_c13 = In_BMP.GetPixel(work_x + 1, work_y + 2);
                        w_c14 = In_BMP.GetPixel(work_x + 2, work_y - 1);
                        w_c15 = In_BMP.GetPixel(work_x - 1, work_y + 2);
                        w_c16 = In_BMP.GetPixel(work_x + 2, work_y + 2);

                        R = (int)((b1 * w_c1.R + b2 * w_c2.R + b3 * w_c3.R + b4 * w_c4.R + b5
* w_c5.R + b6 * w_c6.R + b7 * w_c7.R + b8 * w_c8.R + b9 * w_c9.R + b10 * w_c10.R + b11 *
w_c11.R + b12 * w_c12.R + b13 * w_c13.R + b14 * w_c14.R + b15 * w_c15.R + b16 * w_c16.R));
                        G = (int)((b1 * w_c1.G + b2 * w_c2.G + b3 * w_c3.G + b4 * w_c4.G + b5
* w_c5.G + b6 * w_c6.G + b7 * w_c7.G + b8 * w_c8.G + b9 * w_c9.G + b10 * w_c10.G + b11 *
w_c11.G + b12 * w_c12.G + b13 * w_c13.G + b14 * w_c14.G + b15 * w_c15.G + b16 * w_c16.G));
                        B = (int)((b1 * w_c1.B + b2 * w_c2.B + b3 * w_c3.B + b4 * w_c4.B + b5
* w_c5.B + b6 * w_c6.B + b7 * w_c7.B + b8 * w_c8.B + b9 * w_c9.B + b10 * w_c10.B + b11 *
w_c11.B + b12 * w_c12.B + b13 * w_c13.B + b14 * w_c14.B + b15 * w_c15.B + b16 * w_c16.B));
                        if (R < 0)
                        {
                            R = 0;
                        }
                        if (G < 0)
                        {
                            G = 0;
                        }
                        if (B < 0)
                        {
                            B = 0;
                        }
                        if (R > 255)
                        {
                            R = 255;
                        }
                        if (G > 255)
                        {
                            G = 255;
```

```
                    }
                    if (B > 255)
                    {
                        B = 255;
                    }
                    bmp.SetPixel(x, y, Color.FromArgb(R, G, B));
                }
            }
            Progress_Func(x);
        }

        mod_done = true;
        Progress_Func(0);
    }

    public double sinc_x(double x)
    {
        if (x == 0)
        {
            return 1;
        }
        return (Math.Sin(Math.PI * x)) / (Math.PI * x);
    }

    public double impulse_characteristics_lanczos(double t, int a)  // импульсная
характеристика пикселя для a пикселей вокруг
    {
        if (t == 0)
        {
            return 1;
        }
        if (-a < t && t < a)
        {
            return sinc_x(t) * sinc_x(t / (double)a);
        }

        return 0;
    }

    public void Lanczos(Bitmap In_BMP, int dispersion)
    {
        int w = (int)(In_BMP.Width / resize_number);
        int h = (int)(In_BMP.Height / resize_number);
        int i_width = In_BMP.Width;
        int i_height = In_BMP.Height;
        int ker_size = 2 * dispersion + 1;
        int work_x, work_y;
        int pos_x, pos_y;
        double w_x, w_y;
        double weight_x, weight_y;
        double R, G, B;
        Color work_c;
        mod_done = false;

        for (int y = 0; y < h; y++)
        {
            w_y = (double)y * resize_number;
            work_y = (int)Math.Floor(w_y);
            for (int x = 0; x < w; x++)
            {
                R = 0;
                G = 0;
                B = 0;
                w_x = (double)x * resize_number;
                work_x = (int)Math.Floor(w_x);

                for (int i = 0; i < ker_size; i++)
                {
```

```
                        for (int j = 0; j < ker_size; j++)
                        {
                            pos_x = work_x - dispersion + j;
                            pos_y = work_y - dispersion + i;
                            if (pos_x >= 0 && pos_y >= 0 && pos_x < i_width && pos_y <
i_height)
                            {
                                work_c = In_BMP.GetPixel(pos_x, pos_y);
                                weight_x = impulse_characteristics_lanczos(pos_x - w_x,
dispersion);
                                weight_y = impulse_characteristics_lanczos(pos_y - w_y,
dispersion);
                                R += work_c.R * weight_x * weight_y;
                                G += work_c.G * weight_x * weight_y;
                                B += work_c.B * weight_x * weight_y;
                            }
                        }
                    }
                    if (R < 0)
                    {
                        R = 0;
                    }
                    if (G < 0)
                    {
                        G = 0;
                    }
                    if (B < 0)
                    {
                        B = 0;
                    }
                    if (R > 255)
                    {
                        R = 255;
                    }
                    if (G > 255)
                    {
                        G = 255;
                    }
                    if (B > 255)
                    {
                        B = 255;
                    }
                    bmp.SetPixel(x, y, Color.FromArgb((int)R, (int)G, (int)B));
                }
                Progress_Func(y);
            }

            mod_done = true;
            Progress_Func(0);
        }

        public Bitmap Filter_Image(Bitmap in_bmp, List<List<double>> kernel, int ker_power)
        {
            int k_height = kernel.Count;
            int k_width = kernel[0].Count;
            int convolution_x = kernel[0].Count / 2;
            int convolution_y = kernel.Count / 2;
            int i_height = in_bmp.Height;
            int i_width = in_bmp.Width;
            double col_R = 0;
            double col_G = 0;
            double col_B = 0;
            int pos_x, pos_y;
            Bitmap bmp_copy = new Bitmap(in_bmp);
            mod_done = false;

            for (int i = 0; i < i_height; i++)
            {
```

```
                for (int j = 0; j < i_width; j++)
                {
                    for (int k = 0; k < k_height; k++)
                    {
                        for (int m = 0; m < k_width; m++)
                        {
                            pos_x = j - convolution_x + m;
                            pos_y = i - convolution_y + k;
                            if (pos_x >= 0 && pos_y >= 0 && pos_x < i_width && pos_y <
i_height)
                            {
                                col_R += kernel[k][m] * in_bmp.GetPixel(pos_x, pos_y).R;
                                col_G += kernel[k][m] * in_bmp.GetPixel(pos_x, pos_y).G;
                                col_B += kernel[k][m] * in_bmp.GetPixel(pos_x, pos_y).B;
                            }
                        }
                    }
                    col_R = col_R / ker_power;
                    col_G = col_G / ker_power;
                    col_B = col_B / ker_power;
                    if (col_R < 0)
                    {
                        col_R = 0;
                    }
                    if (col_G < 0)
                    {
                        col_G = 0;
                    }
                    if (col_B < 0)
                    {
                        col_B = 0;
                    }

                    if (col_R > 255)
                    {
                        col_R = 255;
                    }
                    if (col_G > 255)
                    {
                        col_G = 255;
                    }
                    if (col_B > 255)
                    {
                        col_B = 255;
                    }

                    bmp_copy.SetPixel(j, i, Color.FromArgb((int)col_R, (int)col_G,
(int)col_B));

                    col_R = 0;
                    col_G = 0;
                    col_B = 0;
                }
                Progress_Func(i);
            }

            mod_done = true;
            Progress_Func(0);
            return bmp_copy;
        }
        public List<Color> Statistics(Bitmap in_bmp, int color_num)
        {
            List<Color> colors = new List<Color>();
            List<Color> all_colors = new List<Color>();
            List<int> all_colors_num = new List<int>();
            Color work_col = Color.FromArgb(0, 0, 0);

            bool to_add = false;
```

```
            for (int i = 0; i < in_bmp.Height; i++)
            {
                for (int j = 0; j < in_bmp.Width; j++)
                {
                    to_add = true;
                    work_col = in_bmp.GetPixel(j, i);
                    if (all_colors.Count > 0)
                    {
                        for (int k = 0; k < all_colors.Count; k++)
                        {
                            if (all_colors[k].R == work_col.R && all_colors[k].G == work_col.G
&& all_colors[k].B == work_col.B)
                            {
                                all_colors_num[k]++;
                                to_add = false;
                            }
                        }
                    }
                    else
                    {
                        to_add = true;
                    }
                    if (to_add == true)
                    {
                        all_colors.Add(Color.FromArgb(work_col.R, work_col.G, work_col.B));
                        all_colors_num.Add(1);
                    }
                }
            }

            int max = 0;
            int idx = 0;

            color_num = (int)((double)all_colors.Count * ((double)color_num / (double)100));


            for (int k = 0; k < color_num; k++)
            {
                max = 0;
                idx = 0;
                for (int i = 0; i < all_colors_num.Count; i++)
                {
                    if (all_colors_num[i] > max)
                    {
                        idx = i;
                        max = all_colors_num[i];
                    }
                }
                all_colors_num[idx] = 0;
                colors.Add(Color.FromArgb(all_colors[idx].R, all_colors[idx].G,
all_colors[idx].B));

            }
            return colors;
        }
        public void Color_Filter(Bitmap in_bmp, List<Color> colors)
        {
            Color work_col;
            Bitmap bmp_copy = new Bitmap(in_bmp);

            mod_done = false;

            int min = 0;
            int num = 0;
            int idx = 0;
            for (int i = 0; i < in_bmp.Height; i++)
            {
```

```
                for (int j = 0; j < in_bmp.Width; j++)
                {
                    min = 765;
                    idx = 0;
                    num = 0;
                    work_col = in_bmp.GetPixel(j, i);
                    for (int k = 0; k < colors.Count; k++)
                    {
                        num = Math.Abs(colors[k].R - work_col.R) + Math.Abs(colors[k].G -
work_col.G) + Math.Abs(colors[k].B - work_col.B);
                        if (num < min)
                        {
                            min = num;
                            idx = k;
                        }
                    }
                    bmp_copy.SetPixel(j, i, colors[idx]);
                }
                Progress_Func(i);
            }
            mod_done = true;
            Progress_Func(0);
            bmp = bmp_copy;
        }

        public void Median_Filter(Bitmap in_bmp, int dispersion)
        {
            int ker_size = 2 * dispersion + 1;
            List<double> kernel;
            List<int> pos;
            int pos_x = 0;
            int pos_y = 0;
            int i_width = in_bmp.Width;
            int i_height = in_bmp.Height;
            int col_idx = 0;
            Color work_col;
            mod_done = false;
            int count = 0;
            double var;
            Bitmap bmp_copy = new Bitmap(in_bmp);

            for (int i = 0; i < i_height; i++)
            {
                for (int j = 0; j < i_width; j++)
                {
                    count = 0;
                    kernel = new List<double>();
                    pos = new List<int>();
                    for (int k = 0; k < ker_size; k++)
                    {
                        for (int m = 0; m < ker_size; m++)
                        {
                            pos_x = j - dispersion + m;
                            pos_y = i - dispersion + k;
                            if (pos_x >= 0 && pos_y >= 0 && pos_x < i_width && pos_y <
i_height)
                            {
                                work_col = bmp_copy.GetPixel(pos_x, pos_y);
                                kernel.Add((double)(work_col.R + work_col.G + work_col.B) /
(double)3);
                                pos.Add(count);

                            }
                            count++;
                        }
                    }
                    for (int k = 0; k < kernel.Count; k++)
                    {
```

```
                for (int m = 1; m < kernel.Count; m++)
                {
                    if (kernel[m] < kernel[m - 1])
                    {
                        var = kernel[m];
                        kernel[m] = kernel[m - 1];
                        kernel[m - 1] = var;
                        count = pos[m];
                        pos[m] = pos[m - 1];
                        pos[m - 1] = count;
                    }
                }
            }
            col_idx = kernel.Count / 2;
            if (kernel.Count % 2 == 1)
            {
                col_idx++;
            }
            pos_x = j - dispersion + pos[col_idx] % ker_size;
            pos_y = i - dispersion + pos[col_idx] / ker_size;

            work_col = bmp_copy.GetPixel(j, i);
            var = (double)(work_col.R + work_col.G + work_col.B) / (double)3;
            work_col = bmp_copy.GetPixel(pos_x, pos_y);

            if (var == (double)(work_col.R + work_col.G + work_col.B) / (double)3)
            {
                work_col = bmp_copy.GetPixel(j, i);
            }

            in_bmp.SetPixel(j, i, work_col);
        }
        Progress_Func(i);
    }
    mod_done = true;
    Progress_Func(0);
}

public void Sigma_Filter(Bitmap in_bmp, int dispersion, int sigma)
{
    int ker_size = 2 * dispersion + 1;
    int pos_x = 0;
    int pos_y = 0;
    int i_width = in_bmp.Width;
    int i_height = in_bmp.Height;

    Color work_col, work_col2;
    mod_done = false;

    Bitmap bmp_copy = new Bitmap(in_bmp);

    double sum_r = 0, sum_g = 0, sum_b = 0;
    int number_r = 0, number_g = 0, number_b = 0;
    int R = 0, G = 0, B = 0;
    for (int i = 0; i < i_height; i++)
    {
        for (int j = 0; j < i_width; j++)
        {
            sum_r = 0;
            sum_g = 0;
            sum_b = 0;
            number_r = 0;
            number_g = 0;
            number_b = 0;
            for (int k = 0; k < ker_size; k++)
            {
                for (int m = 0; m < ker_size; m++)
                {
```

```
                            pos_x = j - dispersion + m;
                            pos_y = i - dispersion + k;
                            if (pos_x >= 0 && pos_y >= 0 && pos_x < i_width && pos_y <
i_height)
                            {
                                work_col = bmp_copy.GetPixel(j, i);
                                work_col2 = bmp_copy.GetPixel(pos_x, pos_y);
                                R = Math.Abs(work_col.R - work_col2.R);
                                G = Math.Abs(work_col.G - work_col2.G);
                                B = Math.Abs(work_col.B - work_col2.B);
                                if (R < sigma)
                                {
                                    sum_r += work_col2.R;
                                    number_r++;
                                }
                                if(G < sigma)
                                {
                                    sum_g += work_col2.G;
                                    number_g++;
                                }
                                if(B < sigma)
                                {
                                    sum_b += work_col2.B;
                                    number_b++;
                                }
                            }
                        }
                    }

                    in_bmp.SetPixel(j, i, Color.FromArgb((int)(sum_r/number_r), (int)(sum_g /
number_g), (int)(sum_b / number_b)));
                }
                Progress_Func(i);
            }
            mod_done = true;
            Progress_Func(0);
        }

        public void Step_Fil(Bitmap in_bmp, int dispersion, int step)
        {
            Color work_col_1, work_col_2;
            Bitmap bmp_copy = new Bitmap(in_bmp);
            mod_done = false;
            int ker_size = dispersion * 2 + 1;
            int pos_x, pos_y;
            int i_w = in_bmp.Width;
            int i_h = in_bmp.Height;
            double max = 0, num = 0;
            Color exit_col;

            for (int i = 0; i < i_h; i++)
            {
                for (int j = 0; j < i_w; j++)
                {
                    work_col_1 = bmp_copy.GetPixel(j, i);
                    exit_col = work_col_1;
                    max = 0;
                    for (int k = 0; k < ker_size; k++)
                    {
                        for (int m = 0; m < ker_size; m++)
                        {
                            pos_x = j - dispersion + m;
                            pos_y = i - dispersion + k;
                            if (pos_x >= 0 && pos_x < i_w && pos_y >= 0 && pos_y < i_h)
                            {
                                work_col_2 = bmp_copy.GetPixel(pos_x, pos_y);
                                num = Math.Abs(work_col_1.R - work_col_2.R) +
Math.Abs(work_col_1.G - work_col_2.G) + Math.Abs(work_col_1.B - work_col_2.B);
```

```
                                if (Math.Abs(work_col_1.R - work_col_2.R) <= step &&
Math.Abs(work_col_1.G - work_col_2.G) <= step && Math.Abs(work_col_1.B - work_col_2.B) <= step
&& num > max)
                                {
                                    max = num;
                                    exit_col = work_col_2;
                                }
                            }
                        }
                    }
                    in_bmp.SetPixel(j, i, exit_col);
                }
                Progress_Func(i);
            }
            mod_done = true;
            Progress_Func(0);
        }

        public void Erosion(Bitmap in_bmp, List<List<bool>> kernel, int pos)
        {
            Bitmap bmp_copy = new Bitmap(in_bmp);
            mod_done = false;
            int pos_x, pos_y;
            int i_w = in_bmp.Width;
            int i_h = in_bmp.Height;
            int w_dispersion = kernel.Count / 2;
            int h_dispersion = kernel[0].Count / 2;
            int k_w = kernel[0].Count;
            int k_h = kernel.Count;
            int min = 0;
            int num;
            Color work_col_1, exit_col;

            for (int i = 0; i < i_h; i++)
            {
                for (int j = 0; j < i_w; j++)
                {
                    min = 255;
                    exit_col = bmp_copy.GetPixel(j, i);
                    for (int k = 0; k < k_h; k++)
                    {
                        for (int m = 0; m < k_w; m++)
                        {
                            pos_x = j - w_dispersion + m;
                            pos_y = i - h_dispersion + k;
                            if (pos_x >= 0 && pos_x < i_w && pos_y >= 0 && pos_y < i_h &&
kernel[k][m])
                            {
                                work_col_1 = bmp_copy.GetPixel(pos_x, pos_y);
                                num = (work_col_1.R + work_col_1.G + work_col_1.B) / 3;
                                if (Math.Abs(pos - num) < min)
                                {
                                    min = Math.Abs(pos - num);
                                    exit_col = work_col_1;
                                }
                            }
                        }
                    }
                    in_bmp.SetPixel(j, i, exit_col);
                }
                Progress_Func(i);
            }
            mod_done = true;
            Progress_Func(0);
        }

        public void Calculate_Intence()
        {
```

```
            double mean = 0;
            double mean_R = 0, mean_G = 0, mean_B = 0;
            Color work_c1;
            double res = (double)1 / resize_number;
            int i_w = (int)(Load_bmp.Width * res);
            int i_h = (int)(Load_bmp.Height * res);
            for (int i = 0; i < i_h; i++)
            {
                for (int j = 0; j < i_w; j++)
                {
                    if (j < bmp.Width && i < bmp.Height)
                    {
                        work_c1 = bmp.GetPixel(j, i);
                        mean += (double)(work_c1.R + work_c1.G + work_c1.B) / 3;
                        mean_R += work_c1.R;
                        mean_G += work_c1.G;
                        mean_B += work_c1.B;
                    }
                }
            }
            mean_R /= (i_h * i_w);
            mean_G /= (i_h * i_w);
            mean_B /= (i_h * i_w);
            mean /= (i_h * i_w);
            mean_intence = mean;
            mean_intence_R = mean_R;
            mean_intence_G = mean_G;
            mean_intence_B = mean_B;
            was_calc_intence = true;
        }

        public int Normalize_Intence(int color_channel, double mean, double step)
        {
            if (color_channel > mean)
            {
                color_channel = (int)(color_channel + (double)color_channel * step);
            }
            else if (color_channel < mean)
            {
                color_channel = (int)(color_channel - (double)color_channel * step);
            }
            return color_channel;
        }
        public void Change_Contrast(Bitmap in_bmp, double step, double mean_R, double mean_G,
double mean_B)
        {
            mod_done = false;
            Color work_c;
            int R = 0, G = 0, B = 0;
            for (int i = 0; i < bmp.Height; i++)
            {
                for (int j = 0; j < bmp.Width; j++)
                {
                    work_c = in_bmp.GetPixel(j, i);

                    R = Normalize_Intence(work_c.R, mean_R, step);
                    G = Normalize_Intence(work_c.G, mean_G, step);
                    B = Normalize_Intence(work_c.B, mean_B, step);

                    if (R < 0)
                    {
                        R = 0;
                    }
                    if (G < 0)
                    {
                        G = 0;
                    }
                    if (B < 0)
```

```
                {
                    B = 0;
                }

                if (R > 255)
                {
                    R = 255;
                }
                if (G > 255)
                {
                    G = 255;
                }
                if (B > 255)
                {
                    B = 255;
                }
                in_bmp.SetPixel(j, i, Color.FromArgb(R, G, B));
            }
            Progress_Func(i);
        }
        mod_done = true;
        Progress_Func(0);
    }

    public int SCC_Prepare(int channel_intence, int left, int right)
    {
        if (channel_intence >= left && channel_intence <= right)
        {
            channel_intence = (int)(((channel_intence - left) / (double)(right - left)) *
255);
        }
        else if (channel_intence < left)
        {
            channel_intence = 0;
        }
        else if (channel_intence > right)
        {
            channel_intence = 255;
        }
        return channel_intence;
    }
    public void Smart_Change_Contrast(Bitmap in_bmp, int left_border, int right_border)
    {
        int R = 0, G = 0, B = 0;
        Color work_c;
        for (int i = 0; i < in_bmp.Height; i++)
        {
            for (int j = 0; j < in_bmp.Width; j++)
            {
                work_c = in_bmp.GetPixel(j, i);

                R = SCC_Prepare(work_c.R, left_border, right_border);
                G = SCC_Prepare(work_c.G, left_border, right_border);
                B = SCC_Prepare(work_c.B, left_border, right_border);

                if (R < 0)
                {
                    R = 0;
                }
                if (G < 0)
                {
                    G = 0;
                }
                if (B < 0)
                {
                    B = 0;
                }
```

```
            if (R > 255)
            {
                R = 255;
            }
            if (G > 255)
            {
                G = 255;
            }
            if (B > 255)
            {
                B = 255;
            }
            in_bmp.SetPixel(j, i, Color.FromArgb(R, G, B));
        }
    }
}

public Bitmap Gaussian_Blur(Bitmap in_bmp, int dispersion)
{
    int matrix_size = dispersion * 2 + 1;
    List<List<double>> gauss_matrix = new List<List<double>>(matrix_size);

    for (int i = 0; i < matrix_size; i++)
    {
        gauss_matrix.Add(new List<double>(matrix_size));
    }

    for (int i = 0; i < matrix_size; i++)
    {
        for (int j = 0; j < matrix_size; j++)
        {
            if (i == 0 || j == 0)
            {
                gauss_matrix[i].Add(1);
            }
            else
            {
                gauss_matrix[i].Add(gauss_matrix[i - 1][j] + gauss_matrix[i][j - 1]);
            }
        }
    }

    List<double> g1 = new List<double>(matrix_size);

    for (int i = 0; i < matrix_size; i++)
    {
        g1.Add(gauss_matrix[matrix_size - 1 - i][i]);
    }

    for (int i = 0; i < matrix_size; i++)
    {
        for (int j = 0; j < matrix_size; j++)
        {
            gauss_matrix[i][j] = g1[i] * g1[j];
        }
    }

    double koef = 0;
    for (int i = 0; i < matrix_size; i++)
    {
        for (int j = 0; j < matrix_size; j++)
        {
            koef += gauss_matrix[i][j];
        }
    }

    for (int i = 0; i < matrix_size; i++)
    {
```

```
                for (int j = 0; j < matrix_size; j++)
                {
                    gauss_matrix[i][j] /= koef;
                }
            }

            return Filter_Image(in_bmp, gauss_matrix, 1);
        }

        public void Unsharp_Mask(Bitmap in_bmp, int left_border, int right_border, int
gau_radius, int threshold)
        {
            mod_done = false;

            int i_w = in_bmp.Width;
            int i_h = in_bmp.Height;

            Bitmap bmp_copy_1 = new Bitmap(in_bmp);
            Bitmap bmp_copy_2 = new Bitmap(in_bmp);
            Color work_c;
            int R = 0, G = 0, B = 0;

            label3.Text = "smoothing";
            label3.Refresh();

            bmp_copy_1 = Gaussian_Blur(bmp_copy_1, gau_radius);

            label3.Text = "calculating mask";
            label3.Refresh();

            for (int i = 0; i < i_h; i++)
            {
                for (int j = 0; j < i_w; j++)
                {
                    work_c = in_bmp.GetPixel(j, i);
                    R = work_c.R;
                    G = work_c.G;
                    B = work_c.B;
                    work_c = bmp_copy_1.GetPixel(j, i);

                    R = Math.Abs(R - work_c.R);
                    G = Math.Abs(G - work_c.G);
                    B = Math.Abs(B - work_c.B);

                    if (R > 255)
                    {
                        R = 255;
                    }
                    if (G > 255)
                    {
                        G = 255;
                    }
                    if (B > 255)
                    {
                        B = 255;
                    }
                    bmp_copy_1.SetPixel(j, i, Color.FromArgb(R, G, B));
                }
            }

            label3.Text = "changing contrast";
            label3.Refresh();

            Smart_Change_Contrast(bmp_copy_2, left_border, right_border);

            label3.Text = "mask accepting";
            label3.Refresh();
```

```
    Color work_c2, work_c3;

    for (int i = 0; i < i_h; i++)
    {
        for (int j = 0; j < i_w; j++)
        {
            work_c2 = in_bmp.GetPixel(j, i);
            work_c = bmp_copy_1.GetPixel(j, i);
            work_c3 = bmp_copy_2.GetPixel(j, i);

            if ((work_c.R + work_c.G + work_c.B) / 3 >= threshold)
            {
                R = work_c3.R;
                G = work_c3.G;
                B = work_c3.B;
            }
            else
            {
                R = work_c2.R;
                G = work_c2.G;
                B = work_c2.B;
            }
            in_bmp.SetPixel(j, i, Color.FromArgb(R, G, B));
        }
        Progress_Func(i);
    }
    mod_done = true;
    Progress_Func(0);
    label3.Text = "status";
    label3.Refresh();
}

public Bitmap Increase_Mask(Bitmap mask, double edge_intence, int threshold)
{
    label3.Text = "increasing mask intence";
    label3.Refresh();

    mod_done = false;
    Color work_c;
    int R = 0, G = 0, B = 0;
    for (int i = 0; i < mask.Height; i++)
    {
        for (int j = 0; j < mask.Width; j++)
        {
            work_c = mask.GetPixel(j, i);
            R = (int)(work_c.R * edge_intence);
            G = (int)(work_c.G * edge_intence);
            B = (int)(work_c.B * edge_intence);

            if (R < threshold)
            {
                R = 0;
            }
            if (G < threshold)
            {
                G = 0;
            }
            if (B < threshold)
            {
                B = 0;
            }

            if (R > 255)
            {
                R = 255;
            }
            if (G > 255)
            {
```

```
                    G = 255;
                }
                if (B > 255)
                {
                    B = 255;
                }

                mask.SetPixel(j, i, Color.FromArgb(R, G, B));
            }
            Progress_Func(i);
        }

        mod_done = true;
        Progress_Func(0);

        return mask;
    }

    public void Unsharp_Filter(Bitmap in_bmp, int left_border, int right_border, int
gau_radius, int threshold, double edge_intence)
    {
        mod_done = false;

        int i_w = in_bmp.Width;
        int i_h = in_bmp.Height;

        Bitmap bmp_copy_1 = new Bitmap(in_bmp);
        Bitmap bmp_copy_2 = new Bitmap(in_bmp);

        label3.Text = "smoothing";
        label3.Refresh();

        bmp_copy_1 = Gaussian_Blur(bmp_copy_1, gau_radius);
        Color work_c;
        int R = 0, G = 0, B = 0;

        label3.Text = "calculating mask";
        label3.Refresh();

        for (int i = 0; i < i_h; i++)
        {
            for (int j = 0; j < i_w; j++)
            {
                work_c = in_bmp.GetPixel(j, i);
                R = work_c.R;
                G = work_c.G;
                B = work_c.B;
                work_c = bmp_copy_1.GetPixel(j, i);
                if (Math.Abs(R - work_c.R) > threshold)
                {
                    R = Math.Abs(R - work_c.R);
                }

                else
                {
                    R = 0;
                }

                if (Math.Abs(G - work_c.G) > threshold)
                {
                    G = Math.Abs(G - work_c.G);
                }
                else
                {
                    G = 0;
                }

                if (Math.Abs(B - work_c.B) > threshold)
```

```
        {
            B = Math.Abs(B - work_c.B);
        }

        else
        {
            B = 0;
        }

        if (R > 255)
        {
            R = 255;
        }
        if (G > 255)
        {
            G = 255;
        }
        if (B > 255)
        {
            B = 255;
        }
        bmp_copy_1.SetPixel(j, i, Color.FromArgb(R, G, B));
    }
}

bmp_copy_1 = Increase_Mask(bmp_copy_1, edge_intence, threshold);

int w_r = 0, w_g = 0, w_b = 0;

mod_done = false;
label3.Text = "the highest mask intence";
label3.Refresh();

for (int i = 0; i < in_bmp.Height; i++)
{
    for (int j = 0; j < in_bmp.Width; j++)
    {
        work_c = bmp_copy_1.GetPixel(j, i);
        if (work_c.R > w_r)
        {
            w_r = work_c.R;
        }
        if (work_c.G > w_g)
        {
            w_g = work_c.G;
        }
        if (work_c.B > w_b)
        {
            w_b = work_c.B;
        }
    }
    Progress_Func(i);
}

mod_done = true;
Progress_Func(0);

label3.Text = "changing contrast";
label3.Refresh();

Smart_Change_Contrast(bmp_copy_2, left_border, right_border);

label3.Text = "mask accepting";
label3.Refresh();

Color work_c2, work_c3;

double r1 = 0, g1 = 0, b1 = 0;
```

```
    for (int i = 0; i < i_h; i++)
    {
        for (int j = 0; j < i_w; j++)
        {
            work_c2 = in_bmp.GetPixel(j, i);
            work_c = bmp_copy_1.GetPixel(j, i);
            work_c3 = bmp_copy_2.GetPixel(j, i);

            r1 = ((double)(work_c.R) / (double)w_r);
            g1 = ((double)(work_c.G) / (double)w_g);
            b1 = ((double)(work_c.B) / (double)w_b);

            R = (int)(r1 * work_c3.R + (1 - r1) * work_c2.R);
            G = (int)(g1 * work_c3.G + (1 - g1) * work_c2.G);
            B = (int)(b1 * work_c3.B + (1 - b1) * work_c2.B);
            if (R > 255)
            {
                R = 255;
            }
            if (G > 255)
            {
                G = 255;
            }
            if (B > 255)
            {
                B = 255;
            }

            in_bmp.SetPixel(j, i, Color.FromArgb(R, G, B));
        }
        Progress_Func(i);
    }
    mod_done = true;
    Progress_Func(0);
    label3.Text = "status";
    label3.Refresh();
}

public List<Bitmap> Kirsch_Preparing(Bitmap in_bmp)
{
    List<Bitmap> kirsch_bmp = new List<Bitmap>();

    label3.Text = "1 kernel";
    label3.Refresh();
    Bitmap bmp_copy_1 = new Bitmap(in_bmp);
    List<double> kirsch_filter_1 = new List<double> { 5, 5, 5 };
    List<double> kirsch_filter_2 = new List<double> { -3, 0, -3 };
    List<double> kirsch_filter_3 = new List<double> { -3, -3, -3 };
    List<List<double>> kirsch_filter = new List<List<double>>();
    kirsch_filter.Add(kirsch_filter_1);
    kirsch_filter.Add(kirsch_filter_2);
    kirsch_filter.Add(kirsch_filter_3);
    bmp_copy_1 = Filter_Image(bmp_copy_1, kirsch_filter, 1);
    kirsch_bmp.Add(bmp_copy_1);

    label3.Text = "2 kernel";
    label3.Refresh();
    Bitmap bmp_copy_2 = new Bitmap(in_bmp);
    kirsch_filter_1 = new List<double> { 5, 5, -3 };
    kirsch_filter_2 = new List<double> { 5, 0, -3 };
    kirsch_filter_3 = new List<double> { -3, -3, -3 };
    kirsch_filter = new List<List<double>>();
    kirsch_filter.Add(kirsch_filter_1);
    kirsch_filter.Add(kirsch_filter_2);
    kirsch_filter.Add(kirsch_filter_3);
    bmp_copy_2 = Filter_Image(bmp_copy_2, kirsch_filter, 1);
    kirsch_bmp.Add(bmp_copy_2);
```

```
label3.Text = "3 kernel";
label3.Refresh();
Bitmap bmp_copy_3 = new Bitmap(in_bmp);
kirsch_filter_1 = new List<double> { 5, -3, -3 };
kirsch_filter_2 = new List<double> { 5, 0, -3 };
kirsch_filter_3 = new List<double> { 5, -3, -3 };
kirsch_filter = new List<List<double>>();
kirsch_filter.Add(kirsch_filter_1);
kirsch_filter.Add(kirsch_filter_2);
kirsch_filter.Add(kirsch_filter_3);
bmp_copy_3 = Filter_Image(bmp_copy_3, kirsch_filter, 1);
kirsch_bmp.Add(bmp_copy_3);

label3.Text = "4 kernel";
label3.Refresh();
Bitmap bmp_copy_4 = new Bitmap(in_bmp);
kirsch_filter_1 = new List<double> { -3, -3, -3 };
kirsch_filter_2 = new List<double> { 5, 0, -3 };
kirsch_filter_3 = new List<double> { 5, 5, -3 };
kirsch_filter = new List<List<double>>();
kirsch_filter.Add(kirsch_filter_1);
kirsch_filter.Add(kirsch_filter_2);
kirsch_filter.Add(kirsch_filter_3);
bmp_copy_4 = Filter_Image(bmp_copy_4, kirsch_filter, 1);
kirsch_bmp.Add(bmp_copy_4);

label3.Text = "5 kernel";
label3.Refresh();
Bitmap bmp_copy_5 = new Bitmap(in_bmp);
kirsch_filter_1 = new List<double> { -3, -3, -3 };
kirsch_filter_2 = new List<double> { -3, 0, -3 };
kirsch_filter_3 = new List<double> { 5, 5, 5 };
kirsch_filter = new List<List<double>>();
kirsch_filter.Add(kirsch_filter_1);
kirsch_filter.Add(kirsch_filter_2);
kirsch_filter.Add(kirsch_filter_3);
bmp_copy_5 = Filter_Image(bmp_copy_5, kirsch_filter, 1);
kirsch_bmp.Add(bmp_copy_5);

label3.Text = "6 kernel";
label3.Refresh();
Bitmap bmp_copy_6 = new Bitmap(in_bmp);
kirsch_filter_1 = new List<double> { -3, -3, -3 };
kirsch_filter_2 = new List<double> { -3, 0, 5 };
kirsch_filter_3 = new List<double> { -3, 5, 5 };
kirsch_filter = new List<List<double>>();
kirsch_filter.Add(kirsch_filter_1);
kirsch_filter.Add(kirsch_filter_2);
kirsch_filter.Add(kirsch_filter_3);
bmp_copy_6 = Filter_Image(bmp_copy_6, kirsch_filter, 1);
kirsch_bmp.Add(bmp_copy_6);

label3.Text = "7 kernel";
label3.Refresh();
Bitmap bmp_copy_7 = new Bitmap(in_bmp);
kirsch_filter_1 = new List<double> { -3, -3, 5 };
kirsch_filter_2 = new List<double> { -3, 0, 5 };
kirsch_filter_3 = new List<double> { -3, -3, 5 };
kirsch_filter = new List<List<double>>();
kirsch_filter.Add(kirsch_filter_1);
kirsch_filter.Add(kirsch_filter_2);
kirsch_filter.Add(kirsch_filter_3);
bmp_copy_7 = Filter_Image(bmp_copy_7, kirsch_filter, 1);
kirsch_bmp.Add(bmp_copy_7);

label3.Text = "8 kernel";
label3.Refresh();
Bitmap bmp_copy_8 = new Bitmap(in_bmp);
```

```
    kirsch_filter_1 = new List<double> { -3, 5, 5 };
    kirsch_filter_2 = new List<double> { -3, 0, 5 };
    kirsch_filter_3 = new List<double> { -3, -3, -3 };
    kirsch_filter = new List<List<double>>();
    kirsch_filter.Add(kirsch_filter_1);
    kirsch_filter.Add(kirsch_filter_2);
    kirsch_filter.Add(kirsch_filter_3);
    bmp_copy_8 = Filter_Image(bmp_copy_8, kirsch_filter, 1);
    kirsch_bmp.Add(bmp_copy_8);

    return kirsch_bmp;
}
public void Kirsch_Edging(Bitmap in_bmp, int threshold, double edge_intence)
{

    Bitmap bmp_copy_0 = new Bitmap(in_bmp);
    List<Bitmap> kirsch_bmp = Kirsch_Preparing(in_bmp);

    int R = 0, G = 0, B = 0;
    Color work_c;
    work_c = kirsch_bmp[0].GetPixel(0, 0);

    mod_done = false;

    label3.Text = "calculating mask";
    label3.Refresh();
    for (int i = 0; i < in_bmp.Height; i++)
    {
        for (int j = 0; j < in_bmp.Width; j++)
        {
            R = 0;
            G = 0;
            B = 0;
            for (int k = 0; k < kirsch_bmp.Count; k++)
            {
                work_c = kirsch_bmp[k].GetPixel(j, i);
                if (work_c.R > R)
                {
                    R = work_c.R;
                }
                if (work_c.G > G)
                {
                    G = work_c.G;
                }
                if (work_c.B > B)
                {
                    B = work_c.B;
                }
            }
            bmp_copy_0.SetPixel(j, i, Color.FromArgb(R, G, B));
        }
        Progress_Func(i);
    }

    mod_done = true;
    Progress_Func(0);

    R = 0;
    G = 0;
    B = 0;

    mod_done = false;
    label3.Text = "the highest mask intence";
    label3.Refresh();
    for (int i = 0; i < in_bmp.Height; i++)
    {
        for (int j = 0; j < in_bmp.Width; j++)
        {
```

```
                    work_c = bmp_copy_0.GetPixel(j, i);
                    if (work_c.R > R)
                    {
                        R = work_c.R;
                    }
                    if (work_c.G > G)
                    {
                        G = work_c.G;
                    }
                    if (work_c.B > B)
                    {
                        B = work_c.B;
                    }
                }
                Progress_Func(i);
            }

            mod_done = true;
            Progress_Func(0);

            mod_done = false;

            label3.Text = "changing contrast";
            label3.Refresh();

            int w_r, w_g, w_b;
            Color work_c2;

            label3.Text = "mask accepting";
            label3.Refresh();
            for (int i = 0; i < in_bmp.Height; i++)
            {
                for (int j = 0; j < in_bmp.Width; j++)
                {
                    work_c = bmp_copy_0.GetPixel(j, i);
                    work_c2 = in_bmp.GetPixel(j, i);
                    if ((work_c.R + work_c.G + work_c.B) / 3 > threshold)
                    {
                        w_r = (int)(((double)(work_c.R) / (double)R) * work_c2.R *
edge_intence + work_c2.R);
                        w_g = (int)(((double)(work_c.G) / (double)G) * work_c2.G *
edge_intence + work_c2.G);
                        w_b = (int)(((double)(work_c.B) / (double)B) * work_c2.B *
edge_intence + work_c2.B);

                        if (w_r < 0)
                        {
                            w_r = 0;
                        }
                        if (w_g < 0)
                        {
                            w_g = 0;
                        }
                        if (w_b < 0)
                        {
                            w_b = 0;
                        }

                        if (w_r > 255)
                        {
                            w_r = 255;
                        }
                        if (w_g > 255)
                        {
                            w_g = 255;
                        }
                        if (w_b > 255)
                        {
```

```
                    w_b = 255;
                }
                in_bmp.SetPixel(j, i, Color.FromArgb(w_r, w_g, w_b));
            }
        }
        Progress_Func(i);
    }

    mod_done = true;
    Progress_Func(0);
    label3.Text = "status";
    label3.Refresh();

}

public List<Bitmap> Sobel_Preparing(Bitmap in_bmp)
{
    List<Bitmap> sobel_bmp = new List<Bitmap>();

    label3.Text = "1 kernel";
    label3.Refresh();
    Bitmap bmp_copy_1 = new Bitmap(in_bmp);
    List<double> sobel_filter_1 = new List<double> { -1, -2, -1 };
    List<double> sobel_filter_2 = new List<double> { 0, 0, 0 };
    List<double> sobel_filter_3 = new List<double> { 1, 2, 1 };
    List<List<double>> sobel_filter = new List<List<double>>();
    sobel_filter.Add(sobel_filter_1);
    sobel_filter.Add(sobel_filter_2);
    sobel_filter.Add(sobel_filter_3);
    bmp_copy_1 = Filter_Image(bmp_copy_1, sobel_filter, 1);
    sobel_bmp.Add(bmp_copy_1);

    label3.Text = "2 kernel";
    label3.Refresh();
    Bitmap bmp_copy_2 = new Bitmap(in_bmp);
    sobel_filter_1 = new List<double> { -1, 0, 1 };
    sobel_filter_2 = new List<double> { -2, 0, 2 };
    sobel_filter_3 = new List<double> { -1, 0, 1 };
    sobel_filter = new List<List<double>>();
    sobel_filter.Add(sobel_filter_1);
    sobel_filter.Add(sobel_filter_2);
    sobel_filter.Add(sobel_filter_3);
    bmp_copy_2 = Filter_Image(bmp_copy_2, sobel_filter, 1);
    sobel_bmp.Add(bmp_copy_2);

    return sobel_bmp;
}

public List<Bitmap> Roberts_Preparing(Bitmap in_bmp)
{
    List<Bitmap> roberts_bmp = new List<Bitmap>();

    label3.Text = "1 kernel";
    label3.Refresh();
    Bitmap bmp_copy_1 = new Bitmap(in_bmp);
    List<double> roberts_filter_1 = new List<double> { 1, 0 };
    List<double> roberts_filter_2 = new List<double> { 0, -1 };
    List<List<double>> roberts_filter = new List<List<double>>();
    roberts_filter.Add(roberts_filter_1);
    roberts_filter.Add(roberts_filter_2);
    bmp_copy_1 = Filter_Image(bmp_copy_1, roberts_filter, 1);
    roberts_bmp.Add(bmp_copy_1);

    label3.Text = "2 kernel";
    label3.Refresh();
    Bitmap bmp_copy_2 = new Bitmap(in_bmp);
    roberts_filter_1 = new List<double> { 0, 1 };
    roberts_filter_2 = new List<double> { -1, 0 };
```

```
            roberts_filter = new List<List<double>>();
            roberts_filter.Add(roberts_filter_1);
            roberts_filter.Add(roberts_filter_2);
            bmp_copy_2 = Filter_Image(bmp_copy_2, roberts_filter, 1);
            roberts_bmp.Add(bmp_copy_2);

            return roberts_bmp;
        }

        public List<Bitmap> Prewitt_Preparing(Bitmap in_bmp)
        {
            List<Bitmap> prewitt_bmp = new List<Bitmap>();

            label3.Text = "1 kernel";
            label3.Refresh();
            Bitmap bmp_copy_1 = new Bitmap(in_bmp);
            List<double> prewitt_filter_1 = new List<double> { -1, -1, -1 };
            List<double> prewitt_filter_2 = new List<double> { 0, 0, 0 };
            List<double> prewitt_filter_3 = new List<double> { 1, 1, 1 };
            List<List<double>> prewitt_filter = new List<List<double>>();
            prewitt_filter.Add(prewitt_filter_1);
            prewitt_filter.Add(prewitt_filter_2);
            prewitt_filter.Add(prewitt_filter_3);
            bmp_copy_1 = Filter_Image(bmp_copy_1, prewitt_filter, 1);
            prewitt_bmp.Add(bmp_copy_1);

            label3.Text = "2 kernel";
            label3.Refresh();
            Bitmap bmp_copy_2 = new Bitmap(in_bmp);
            prewitt_filter_1 = new List<double> { -1, 0, 1 };
            prewitt_filter_2 = new List<double> { -1, 0, 1 };
            prewitt_filter_3 = new List<double> { -1, 0, 1 };
            prewitt_filter = new List<List<double>>();
            prewitt_filter.Add(prewitt_filter_1);
            prewitt_filter.Add(prewitt_filter_2);
            prewitt_filter.Add(prewitt_filter_3);
            bmp_copy_2 = Filter_Image(bmp_copy_2, prewitt_filter, 1);
            prewitt_bmp.Add(bmp_copy_2);

            return prewitt_bmp;
        }

        public void SPR_Edging(Bitmap in_bmp, List<Bitmap> filter_bmp, int threshold, double
edge_intence)
        {
            Bitmap bmp_copy_0 = new Bitmap(in_bmp);

            int R = 0, G = 0, B = 0;
            Color work_c;
            work_c = filter_bmp[0].GetPixel(0, 0);

            mod_done = false;

            label3.Text = "calculating mask";
            label3.Refresh();

            for (int i = 0; i < in_bmp.Height; i++)
            {
                for (int j = 0; j < in_bmp.Width; j++)
                {
                    R = 0;
                    G = 0;
                    B = 0;
                    for (int k = 0; k < filter_bmp.Count; k++)
                    {
                        work_c = filter_bmp[k].GetPixel(j, i);
                        R += work_c.R * work_c.R;
                        G += work_c.G * work_c.G;
```

```
            B += work_c.B * work_c.B;
        }
        R = (int)Math.Sqrt(R);
        G = (int)Math.Sqrt(G);
        B = (int)Math.Sqrt(B);
        if (R > 255)
        {
            R = 255;
        }
        if (G > 255)
        {
            G = 255;
        }
        if (B > 255)
        {
            B = 255;
        }
        bmp_copy_0.SetPixel(j, i, Color.FromArgb(R, G, B));
        //in_bmp.SetPixel(j, i, Color.FromArgb(R, G, B));
    }
    Progress_Func(i);
}

mod_done = true;
Progress_Func(0);

R = 0;
G = 0;
B = 0;

mod_done = false;
label3.Text = "the highest mask intence";
label3.Refresh();
for (int i = 0; i < in_bmp.Height; i++)
{
    for (int j = 0; j < in_bmp.Width; j++)
    {
        work_c = bmp_copy_0.GetPixel(j, i);
        if (work_c.R > R)
        {
            R = work_c.R;
        }
        if (work_c.G > G)
        {
            G = work_c.G;
        }
        if (work_c.B > B)
        {
            B = work_c.B;
        }
    }
    Progress_Func(i);
}

mod_done = true;
Progress_Func(0);

mod_done = false;

int w_r, w_g, w_b;
Color work_c2;
label3.Text = "mask accepting";
label3.Refresh();
for (int i = 0; i < in_bmp.Height; i++)
{
    for (int j = 0; j < in_bmp.Width; j++)
    {
        work_c = bmp_copy_0.GetPixel(j, i);
```

```
                    work_c2 = in_bmp.GetPixel(j, i);
                    if ((work_c.R + work_c.G + work_c.B) / 3 > threshold)
                    {
                        w_r = (int)(((double)(work_c.R) / (double)R) * work_c2.R *
edge_intence + work_c2.R);
                        w_g = (int)(((double)(work_c.G) / (double)G) * work_c2.G *
edge_intence + work_c2.G);
                        w_b = (int)(((double)(work_c.B) / (double)B) * work_c2.B *
edge_intence + work_c2.B);

                        if (w_r < 0)
                        {
                            w_r = 0;
                        }
                        if (w_g < 0)
                        {
                            w_g = 0;
                        }
                        if (w_b < 0)
                        {
                            w_b = 0;
                        }

                        if (w_r > 255)
                        {
                            w_r = 255;
                        }
                        if (w_g > 255)
                        {
                            w_g = 255;
                        }
                        if (w_b > 255)
                        {
                            w_b = 255;
                        }
                        in_bmp.SetPixel(j, i, Color.FromArgb(w_r, w_g, w_b));
                    }
                }
                Progress_Func(i);
            }

            mod_done = true;
            Progress_Func(0);
            label3.Text = "status";
            label3.Refresh();
        }

        public Bitmap Any_Interpolation_Heuristic(Bitmap in_bmp, int dispersion)
        {
            int w = (int)(in_bmp.Width / resize_number);
            int h = (int)(in_bmp.Height / resize_number);
            int i_width = in_bmp.Width;
            int i_height = in_bmp.Height;
            int ker_size = 2 * dispersion;
            int work_x, work_y;
            int pos_x, pos_y;
            List<List<double>> kernel = new List<List<double>>(ker_size);
            Bitmap res_bmp = new Bitmap(bmp);
            bool only_orig = false;
            for(int i = 0; i < ker_size; i++)
            {
                kernel.Add(new List<double>(ker_size));
                for(int j = 0; j < ker_size; j++)
                {
                    kernel[i].Add(-1);
                }
            }
            double w_x, w_y;
```

```
            double R, G, B;
            Color work_c;
            mod_done = false;
            double sum = 0;

            for (int y = 0; y < h; y++)
            {
                w_y = (double)y * resize_number;
                work_y = (int)Math.Floor(w_y);
                for (int x = 0; x < w; x++)
                {
                    R = 0;
                    G = 0;
                    B = 0;
                    w_x = (double)x * resize_number;
                    work_x = (int)Math.Floor(w_x);
                    only_orig = false;
                    for (int i = 0; i < ker_size; i++)
                    {
                        pos_y = work_y - dispersion + i + 1;
                        for (int j = 0; j < ker_size; j++)
                        {
                            pos_x = work_x - dispersion + j + 1;
                            if (pos_x >= 0 && pos_y >= 0 && pos_x < i_width && pos_y <
i_height)
                            {
                                kernel[i][j] = Math.Sqrt(Math.Pow(Math.Abs(w_x - pos_x), 2) +
Math.Pow(Math.Abs(w_y - pos_y), 2));
                                if(kernel[i][j] > 0)
                                {
                                    kernel[i][j] = 1 / kernel[i][j];
                                }
                                if(kernel[i][j] == 0)
                                {
                                    only_orig = true;
                                }
                            }
                        }
                    }
                    if (!only_orig)
                    {
                        sum = 0;
                        for (int i = 0; i < ker_size; i++)
                        {
                            for (int j = 0; j < ker_size; j++)
                            {
                                if (kernel[i][j] > 0)
                                {
                                    sum += kernel[i][j];
                                }
                            }
                        }
                        for (int i = 0; i < ker_size; i++)
                        {
                            for (int j = 0; j < ker_size; j++)
                            {
                                if (kernel[i][j] > 0)
                                {
                                    kernel[i][j] /= sum;
                                }
                            }
                        }

                        for (int i = 0; i < ker_size; i++)
                        {
                            pos_y = work_y - dispersion + i + 1;
                            for (int j = 0; j < ker_size; j++)
                            {
```

```
                                pos_x = work_x - dispersion + j + 1;
                                if (pos_x >= 0 && pos_y >= 0 && pos_x < i_width && pos_y <
i_height)
                                {
                                    work_c = in_bmp.GetPixel(pos_x, pos_y);
                                    if (kernel[i][j] > 0)
                                    {
                                        R += work_c.R * kernel[i][j];
                                        G += work_c.G * kernel[i][j];
                                        B += work_c.B * kernel[i][j];
                                    }
                                }
                            }
                        }

                        for (int i = 0; i < ker_size; i++)
                        {
                            for (int j = 0; j < ker_size; j++)
                            {
                                kernel[i][j] = -1;
                            }
                        }
                    }
                    else
                    {
                        work_c = in_bmp.GetPixel(work_x, work_y);
                        R += work_c.R;
                        G += work_c.G;
                        B += work_c.B;
                    }
                    if (R < 0)
                    {
                        R = 0;
                    }
                    if (G < 0)
                    {
                        G = 0;
                    }
                    if (B < 0)
                    {
                        B = 0;
                    }
                    if (R > 255)
                    {
                        R = 255;
                    }
                    if (G > 255)
                    {
                        G = 255;
                    }
                    if (B > 255)
                    {
                        B = 255;
                    }
                    res_bmp.SetPixel(x, y, Color.FromArgb((int)R, (int)G, (int)B));
                }
                Progress_Func(y);
            }

            mod_done = true;
            Progress_Func(0);

            return res_bmp;
        }

        public void Progress_Func(int current_position)
        {
            if (!mod_done)
```

```
    {
        double step = bmp.Height / (double)100;
        int progress = (int)(((double)current_position) / step);
        progressBar1.Value = progress;
    }
    else
    {
        progressBar1.Value = 0;
    }
}

public void Draw_Image()
{
    g.Clear(Color.White);
    story.Add(bmp);
    bmp = new Bitmap(story[story.Count-1]);
    pictureBox1.Image = bmp;
}

private void button2_Click(object sender, EventArgs e)
{
    if (checkBox1.Checked == true)
    {
        resizenumber(Load_bmp);
        BiLinear(Load_bmp);
        Draw_Image();
    }
    else
    {
        resizenumber(bmp);
        BiLinear(bmp);
        Draw_Image();
    }
    listBox1.Items.Add("BiLinear");
}

private void button3_Click(object sender, EventArgs e)
{
        resizenumber(Load_bmp);
        GetBmp(Load_bmp);
        Draw_Image();
        listBox1.Items.Add("Neighbour");
}

private void button4_Click(object sender, EventArgs e)
{
    if (checkBox1.Checked == true)
    {
        resizenumber(Load_bmp);
        BiCubic(Load_bmp);
        Draw_Image();
    }
    else
    {
        resizenumber(bmp);
        BiCubic(bmp);
        Draw_Image();
    }
    listBox1.Items.Add("BiCubic");
}

private void button6_Click(object sender, EventArgs e)
{
    List<double> HP_filter_1 = new List<double> { -1, -1, -1 };
    List<double> HP_filter_2 = new List<double> { -1, 9, -1 };
    List<double> HP_filter_3 = new List<double> { -1, -1, -1 };
    List<List<double>> HP_filter = new List<List<double>>();
    HP_filter.Add(HP_filter_1);
```

```
            HP_filter.Add(HP_filter_2);
            HP_filter.Add(HP_filter_3);
            bmp = Filter_Image(bmp, HP_filter, 1);
            Draw_Image();
            listBox1.Items.Add("HighPassFilter");
        }

        private void button7_Click(object sender, EventArgs e)
        {
            Bitmap newbmp;
            if (Load_bmp.Width > 100||Load_bmp.Height > 100)
            {
                int h = 100;
                int w = 100;
                if(Load_bmp.Width < Load_bmp.Height)
                {
                    w = (int)(w * (Load_bmp.Width / (double)Load_bmp.Height));
                }
                if(Load_bmp.Width > Load_bmp.Height)
                {
                    h = (int)(h * (Load_bmp.Height / (double)Load_bmp.Width));
                }
                newbmp = new Bitmap(bmp, w, h);
            }
            else
            {
                newbmp = Load_bmp;
            }
            Color_Filter(bmp, Statistics(newbmp, trackBar1.Value));
            Draw_Image();
            listBox1.Items.Add("ColorFil");
        }

        private void button8_Click(object sender, EventArgs e)
        {
            List<double> smoothing_filter_1 = new List<double> { 0.000789, 0.006581, 0.013347,
0.006581, 0.000789};
            List<double> smoothing_filter_2 = new List<double> { 0.006581, 0.054901, 0.111345,
0.054901, 0.006581 };
            List<double> smoothing_filter_3 = new List<double> { 0.013347, 0.111345, 0.225821,
0.111345, 0.013347 };
            List<double> smoothing_filter_4 = new List<double> { 0.006581, 0.054901, 0.111345,
0.054901, 0.006581 };
            List<double> smoothing_filter_5 = new List<double> { 0.000789, 0.006581, 0.013347,
0.006581, 0.000789 };
            List<List<double>> smoothing_filter = new List<List<double>>();
            smoothing_filter.Add(smoothing_filter_1);
            smoothing_filter.Add(smoothing_filter_2);
            smoothing_filter.Add(smoothing_filter_3);
            smoothing_filter.Add(smoothing_filter_4);
            smoothing_filter.Add(smoothing_filter_5);

            bmp = Filter_Image(bmp, smoothing_filter, 1);

            Draw_Image();
            listBox1.Items.Add("Smoothing");
        }

        private void trackBar1_ValueChanged(object sender, EventArgs e)
        {
            numericUpDown1.Value = trackBar1.Value;
        }

        private void numericUpDown1_ValueChanged(object sender, EventArgs e)
        {
            trackBar1.Value = (int)numericUpDown1.Value;
        }
```

```
private void button9_Click(object sender, EventArgs e)
{
    Median_Filter(bmp, trackBar2.Value);
    Draw_Image();
    listBox1.Items.Add("Median");
}

private void trackBar2_ValueChanged(object sender, EventArgs e)
{
    numericUpDown2.Value = trackBar2.Value;
}

private void numericUpDown2_ValueChanged(object sender, EventArgs e)
{
    trackBar2.Value = (int)numericUpDown2.Value;
}

private void button10_Click(object sender, EventArgs e)
{

    List<double> edge_filter_1 = new List<double> { 0, -1, 0};
    List<double> edge_filter_2 = new List<double> {-1,5,-1};
    List<double> edge_filter_3 = new List<double> { 0,-1,0};
    List<List<double>> edge_filter = new List<List<double>>();
    edge_filter.Add(edge_filter_1);
    edge_filter.Add(edge_filter_2);
    edge_filter.Add(edge_filter_3);
    bmp = Filter_Image(bmp, edge_filter, 1);
    Draw_Image();
    listBox1.Items.Add("Sharpen");
}

private void button11_Click(object sender, EventArgs e)
{

    Step_Fil(bmp, 2, 5);
    Draw_Image();
    listBox1.Items.Add("StepFil");
}

private void button12_Click(object sender, EventArgs e)
{
    List<bool> erosion_1 = new List<bool> { false, false, true, false, false };
    List<bool> erosion_2 = new List<bool> { false, true, true, true, false };
    List<bool> erosion_3 = new List<bool> { true, true, true, true, true };
    List<bool> erosion_4 = new List<bool> { false, true, true, true, false };
    List<bool> erosion_5 = new List<bool> { false, false, true, false, false };
    List<List<bool>> erosion = new List<List<bool>>();
    erosion.Add(erosion_1);
    erosion.Add(erosion_2);
    erosion.Add(erosion_3);
    erosion.Add(erosion_4);
    erosion.Add(erosion_5);
    Erosion(bmp, erosion, 0);
    Draw_Image();
    listBox1.Items.Add("Erosion");
}

private void button13_Click(object sender, EventArgs e)
{
    List<bool> erosion_1 = new List<bool> { false, false, true, false, false };
    List<bool> erosion_2 = new List<bool> { false, true, true, true, false };
    List<bool> erosion_3 = new List<bool> { true, true, true, true, true };
    List<bool> erosion_4 = new List<bool> { false, true, true, true, false };
    List<bool> erosion_5 = new List<bool> { false, false, true, false, false };
    List<List<bool>> erosion = new List<List<bool>>();
    erosion.Add(erosion_1);
    erosion.Add(erosion_2);
```

```
            erosion.Add(erosion_3);
            erosion.Add(erosion_4);
            erosion.Add(erosion_5);
            Erosion(bmp, erosion, 255);
            Draw_Image();
            listBox1.Items.Add("Build-up");
        }

        private void trackBar3_ValueChanged(object sender, EventArgs e)
        {
            bmp = new Bitmap(story[story.Count - 1]);
            numericUpDown3.Value = trackBar3.Value;
            if (!was_calc_intence)
            {
                Calculate_Intence();
            }
            Change_Contrast(bmp, ((double)trackBar3.Value / (double)100), mean_intence_R,
mean_intence_G, mean_intence_B);
            g.Clear(Color.White);
            pictureBox1.Image = bmp;
        }

        private void numericUpDown3_ValueChanged(object sender, EventArgs e)
        {
            trackBar3.Value = (int)numericUpDown3.Value;
        }

        private void button14_Click(object sender, EventArgs e)
        {
            was_calc_intence = false;
            Draw_Image();
            listBox1.Items.Add("Changed contrast");
        }

        private void button15_Click(object sender, EventArgs e)
        {
            bmp = new Bitmap(story[story.Count - 1]);
            g.Clear(Color.White);
            pictureBox1.Image = bmp;
        }

        private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
        {
            if(listBox1.SelectedIndex < listBox1.Items.Count-1)
            {
                int lb_count = listBox1.Items.Count;
                for (int i = listBox1.SelectedIndex+1; i < lb_count; i++)
                {
                    listBox1.Items.RemoveAt(listBox1.SelectedIndex + 1);
                    story.RemoveAt(listBox1.SelectedIndex + 1);
                }
                bmp = new Bitmap(story[story.Count - 1]);
                g.Clear(Color.White);
                pictureBox1.Image = bmp;
                listBox1.SelectedIndex = listBox1.Items.Count - 1;
            }
        }

        private void button16_Click(object sender, EventArgs e)
        {
            Bitmap newbmp = new Bitmap(bmp, (int)numericUpDown5.Value,
(int)numericUpDown4.Value);
            bmp = newbmp;
            Draw_Image();
            listBox1.Items.Add("Pixelised");
        }

        private void button17_Click(object sender, EventArgs e)
```

```
{
    Smart_Change_Contrast(bmp, 50,200);
    Draw_Image();
    listBox1.Items.Add("SCC");
}

private void button18_Click(object sender, EventArgs e)
{
    Unsharp_Mask(bmp, 30,220, 2, 10);
    Draw_Image();
    listBox1.Items.Add("Unsharp Mask");
}

private void button19_Click(object sender, EventArgs e)
{
    Kirsch_Edging(bmp, 30, 0.1);
    Draw_Image();
    listBox1.Items.Add("Kirsch Edging");
}

 private void button27_Click(object sender, EventArgs e)
{
    resizenumber(Load_bmp);
    bmp = Any_Interpolation_Heuristic(Load_bmp, 3);
    Draw_Image();
    listBox1.Items.Add("AIH");
}

private void button20_Click(object sender, EventArgs e)
{
    List<Bitmap> filter_bmp = Sobel_Preparing(bmp);
    SPR_Edging(bmp, filter_bmp, 30, 0.5);
    Draw_Image();
    listBox1.Items.Add("Sobel Edging");
}

private void button21_Click(object sender, EventArgs e)
{
    List<Bitmap> filter_bmp = Prewitt_Preparing(bmp);
    SPR_Edging(bmp, filter_bmp, 30, 0.5);
    Draw_Image();
    listBox1.Items.Add("Prewitt Edging");
}

private void button26_Click(object sender, EventArgs e)
{
    Sigma_Filter(bmp, trackBar2.Value, 10);
    Draw_Image();
    listBox1.Items.Add("Sigma");
}

private void button22_Click(object sender, EventArgs e)
{
    List<Bitmap> filter_bmp = Prewitt_Preparing(bmp);
    SPR_Edging(bmp, filter_bmp, 30, 0.5);
    Draw_Image();
    listBox1.Items.Add("Roberts Edging");
}

private void button23_Click(object sender, EventArgs e)
{
    Unsharp_Filter(bmp, 100, 150, 7, 5, 3.5);
    Draw_Image();
    listBox1.Items.Add("Heuristic Unsharp");
}

private void button24_Click(object sender, EventArgs e)
{
```

```
            bmp = Gaussian_Blur(bmp, 3);
            Draw_Image();
            listBox1.Items.Add("Gaussian Blur");
        }

        private void button5_Click(object sender, EventArgs e)
        {
            resizenumber(Load_bmp);
            Lanczos(Load_bmp,3);
            Draw_Image();
            listBox1.Items.Add("Lanczos");
        }

        private void button1_Click(object sender, EventArgs e)
        {
            open_dialog.Filter = "Image
Files(*.BMP;*.JPG;*.GIF;*.PNG)|*.BMP;*.JPG;*.GIF;*.PNG|All files (*.*)|*.*"; //формат
загружаемого файла
            if (open_dialog.ShowDialog() == DialogResult.OK) //если в окне была нажата кнопка
"OK"
            {
                try
                {
                    Load_bmp = new Bitmap(open_dialog.FileName);
                    if(listBox1.Items.Count > 0)
                    {
                        while(listBox1.Items.Count > 0)
                        {
                            listBox1.Items.RemoveAt(0);
                            story.RemoveAt(0);
                        }
                    }
                    resizenumber(Load_bmp);
                    GetBmp(Load_bmp);
                    Draw_Image();
                    listBox1.Items.Add("Neighbour");
                    was_calc_intence = false;
                }
                catch
                {
                    DialogResult rezult = MessageBox.Show("Невозможно открыть выбранный файл",
                    "Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);
                }
            }
        }
    }
}
```