МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение высшего образования
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА 44

КУРСОВОЙ ПРОЕКТ
ЗАЩИЩЕНА С ОЦЕНКОЙ

РУКОВОДИТЕЛЬ

старший преподаватель		Е.К. Григорьев
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ОТЧЕТ О ПРОЕКТЕ НА ТЕМУ

«Моделирование и исследование алгоритмов скремблирования цифровых изображений»

по дисциплине: Моделирование

РАБОТУ ВЫПОЛНИЛИ

СТУДЕНТЫ гр. № 4142 Д.Р. Рябов
Е.В. Асламова
В. И. Вихрова
Н.С. Никифоров
подпись, дата
инициалы, фамилия

Санкт-Петербург 2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цифровые изображения играют значительную роль в современном мире, применяясь в различных областях, таких как медицина, графика, обработка видео и другие. Однако, с увеличением доступности и распространенности цифровых изображений возникает вопрос безопасности их использования. Одним из способов обеспечения безопасности является скремблирование изображений.

Целью данного проекта является оценка алгоритмов скремблирования цифрового изображения и выбор наилучшего алгоритма. Для достижения этой цели было выполнено следующее:

1. Исходное изображение в оттенках серого размером NxN, например, широко известное - Лена, было загружено и выведено в MATLAB с использованием команды imshow. Это позволило получить представление о исходном изображении и использовать его в дальнейшем моделировании.

2. Было проведено моделирование алгоритмов скремблирования, описанных в источнике (https://www.elibrary.ru/download/elibrary_35044402_93482034.pdf). Для каждого алгоритма были получены скремблированные изображения.

3. Каждый алгоритм был оценен визуально, а также при помощи метрик, представленных в работе. Это позволило провести объективную оценку эффективности каждого алгоритма скремблирования.

4. Результаты оценки были занесены в таблицу, что позволило сравнить и проанализировать эффективность каждого алгоритма.

5. На основании проведенных оценок был выбран наилучший алгоритм скремблирования, который может быть рекомендован для использования в целях защиты цифровых изображений.

Таким образом, выполнение данного проекта позволило оценить алгоритмы скремблирования цифрового изображения и выбрать наилучший алгоритм, способствующий обеспечению безопасности и защите цифровых изображений.

КОМАНДА ПРОЕКТА

Ниже представлены роли участников проекта:

1. Никифоров Никита Сергеевич: разработка и анализ программы оценки созданных алгоритмов, разработка алгоритма скремблирования Арнольда, составление формулировки проектного задания, оформление приложений проекта.
2. Рябов Даниил Романович: разработка и анализ программы оценки созданных алгоритмов, оформление изображений для проектного задания.
3. Вихрова Василина Игоревна: разработка и анализ прайм-алгоритма перетасовки, оформление текста для проектного задания.
4. Асламова Елизавета Вячеславовна: разработка и анализ алгоритма Рубика, оформление изображений для проектного задания.

1. Проектное задание

Дано исходное изображение в оттенках серого размером NxN (например, 512 или 1024), например, широко известное – Лена. (http://lenna.org/). Вывести данное изображение в MATLAB (команда imshow).

Провести моделирование алгоритмов скремблирования из источника (https://www.elibrary.ru/download/elibrary_35044402_93482034.pdf). Вывести скремблированные изображения для каждого алгоритма. Оценить каждый алгоритм визуально и при помощи метрик, представленных в работе. Занести результаты в таблицу. Выбрать лучший алгоритм.

2. Использованные критерии сравнения алгоритмов скремблирования

Для оценки эффективности алгоритмов скремблирования была использована формула 1.

$$D_s = DSF \times GSF, \quad (1)$$

где D_s – степень скремблирования;

DSF (англ. distance scrambling factor) – фактор, определяемый расстоянием между начальным и конечным положением пикселей изображения;

GSF (англ. gray scrambling factor) – фактор, определяемый величиной изменения хаотичности изображения.

Для дальнейших расчетов были использованы формулы 2-6.

Расстояние между начальным положением пикселя изображения (x, y) и конечным (x′, y′) определяется по формуле:

$$\mathrm{d}(\mathrm{x},\mathrm{y}) = \sqrt{(x - x')^2 + (y - y')^2} \quad (2)$$

Среднее значение d(x, y) при обработке изображения A размера m×n рассчитывается по формуле:

$$E(d) = \frac{1}{m \times n} \sum_{x=1}^{m} \sum_{y=1}^{n} d(x, y) \quad (3)$$

Очевидно, что если ни один пиксель не изменил своего местоположения, то есть изображение не было скремблировано, то E(d) принимает минимальное значение $E(d)_{min} = 0$, а если пиксели произвели перемещение на максимально возможное расстояние, E(d) принимает максимальное значение, вычисляемое по формуле:

$$E(d)_{max} = \sqrt{(m-1)^2 + (n-1)^2} \quad (4)$$

Так как степень скремблирования линейно зависит от расстояния, на которое перемещены пиксели, формулу для вычисления DSF можно записать следующим образом:

$$DSF(A, A') = \frac{E(d)}{E(d)_{max}}, \quad (5)$$

где A – исходное изображение;

A′ – скремблированное изображение.

GSF (фактор, определяемый величиной изменения хаотичности изображения) был найден по формуле 3.

$$GSF = \frac{\sigma^2{}_1}{\sigma^2{}_2}, \quad (6)$$

где σ_1^2– хаотичность исходного изображения;

σ_2^2– хаотичность скремблированного изображения.

В свою очередь, эти значения вычисляются по формулам 7-9.

Разделив оригинальные и скремблированное изображение на непересекающиеся блоки размера k × k пикселей, среднее значение для пикселей блока Bpl рассчитываем по формуле:

$$E(Bpl) = \frac{1}{k \times k} \sum_{i=1}^{k} \sum_{j=1}^{k} Bpl(i, j) \quad (7)$$

Так как изображение разделено на (m/k) × (n/k) блоков, существует столько же средних значений для всей совокупности блоков. Для определения среднего значения пикселей всего изображения используется формула 8, затем вычисляется квадрат среднего квадратического отклонения (дисперсия) средних значений пикселей блоков от среднего значения пикселей всего изображения по формуле 9.

$$E(E(Bpl)) = \frac{1}{(\frac{m}{k})\times(\frac{n}{k})}\sum_{p=1}^{m/k}\sum_{l=1}^{n/k}E(Bpl) \quad (8)$$

$$\sigma^2 = \frac{1}{(\frac{m}{k})\times(\frac{n}{k})}\sum_{p=1}^{m/k}\sum_{l=1}^{n/k}(E(Bpl) - E(E(Bpl)))^2 \quad (9)$$

Для дальнейших вычислений оценки для каждого из алгоритмов, был написан код, листинг которого представлен в приложении А.

3. Алгоритм Арнольда

Алгоритм Арнольда был разработан выдающимся математиком Владимиром Игоревичем Арнольдом в целях демонстрации двумерного отображения, где фазовое пространство моделируется в виде поверхности тора [2]. В настоящее время этот алгоритм активно применяется не только в математических исследованиях, но и в различных практических областях, включая защиту цифровых данных, обработку изображений, создание криптографических алгоритмов, а также в компьютерной графике и генерации псевдослучайных чисел.

Алгоритм Арнольда задает взаимно-однозначное отображение и описывается следующим соотношением 10-11.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} \ mod\ N, \tag{10}$$

где x, y – координаты элемента в исходной квадратной матрице;

x′, y′ – координаты элемента в новой матрице;

N – порядок матрицы;

Mod N – остаток от деления на порядок матрицы.

Обратное преобразование Арнольда выполняется по следующей формуле:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix} \times \begin{bmatrix} x' \\ y' \end{bmatrix} \ mod\ N, \tag{11}$$

На рисунке 1 изображено блочное представление работы алгоритма Арнольда для итераций изображения.

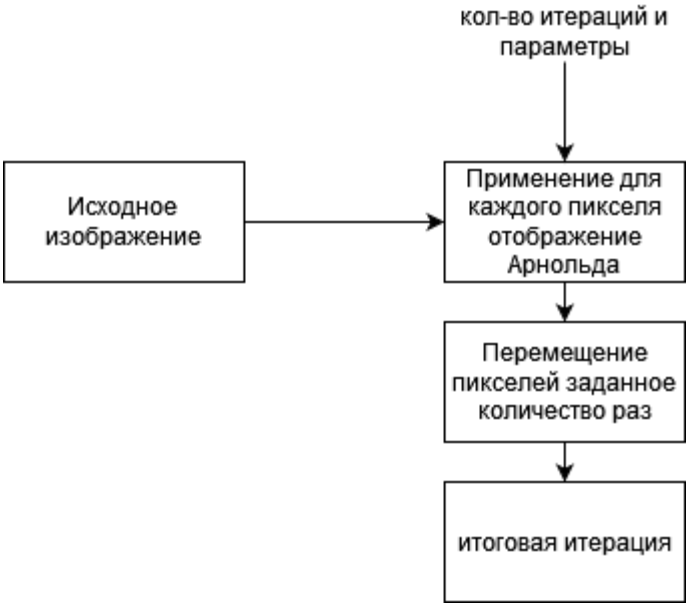

Рисунок 1 – Схема процесса итерации алгоритма Арнольда

На рисунке 2 показано стандартное изображение «Lena».

Рисунок 2 – Стандартного тестового изображения «Лена»

Итерации изображения «Лена» показаны на рисунке 3.

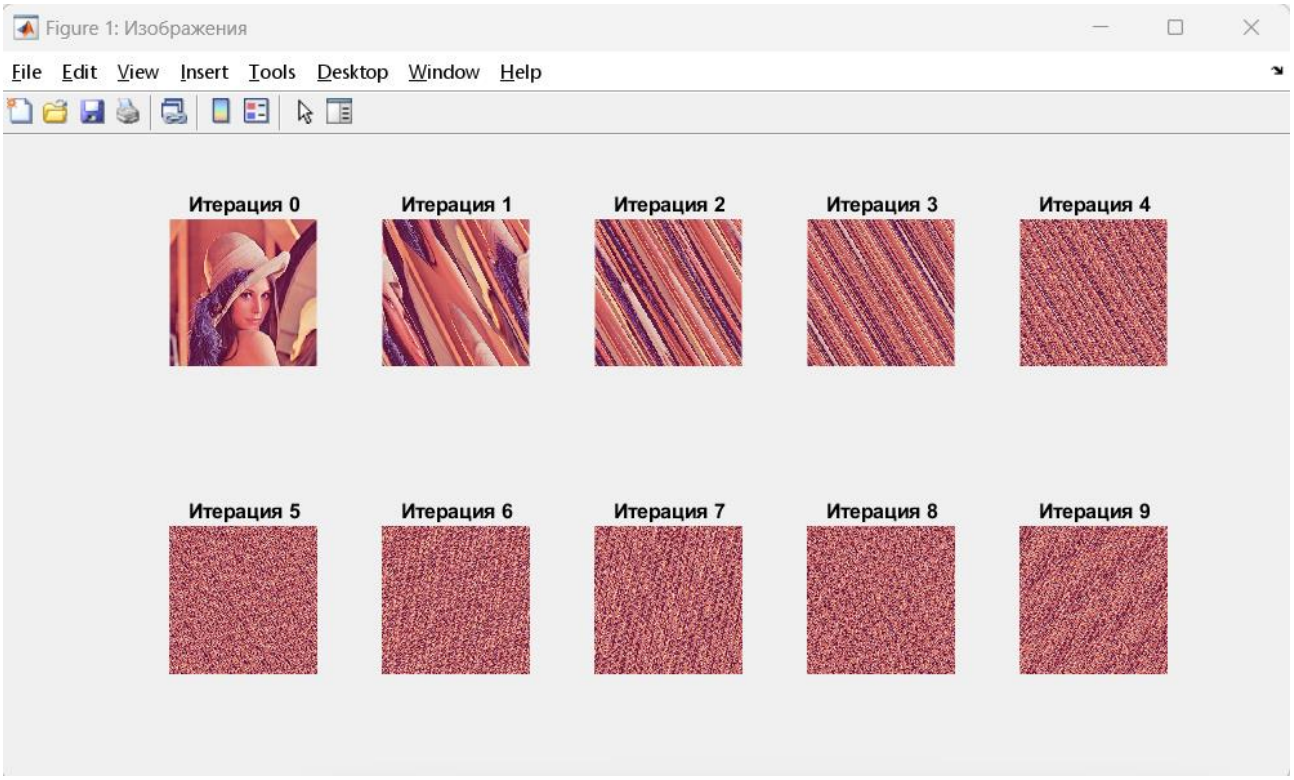

Рисунок 3 – Результат итераций с помощью алгоритма Арнольда изображения «Лена»

Результат итерации изображения «Лена» показан на рисунке 4.

Рисунок 4 – Последний этап итераций с помощью алгоритма Арнольда изображения «Лена»

Далее для объективной оценки качества обработки изображения были простроены графики зависимости энтропии от количества итераций для изображения. Этот показатель позволяет определить степень насыщенности и контрастности изображения, то есть насколько равномерно распределены тона и оттенки изображения, и помогает настроить параметры алгоритма для достижения оптимальных результатов.

Для вычисления энтропии была использована следующая формула:

$$E_r = \frac{S_s}{S_i}, \tag{12}$$

где E_r – энтропия;

S_s – размер скремблированного изображения;

S_i размер оригинального изображения.

Дальнейшее вычисление энтропии в проекте будет происходить аналогично.

График зависимости энтропии от количества итераций для изображения «Лена» показан на рисунке 5.

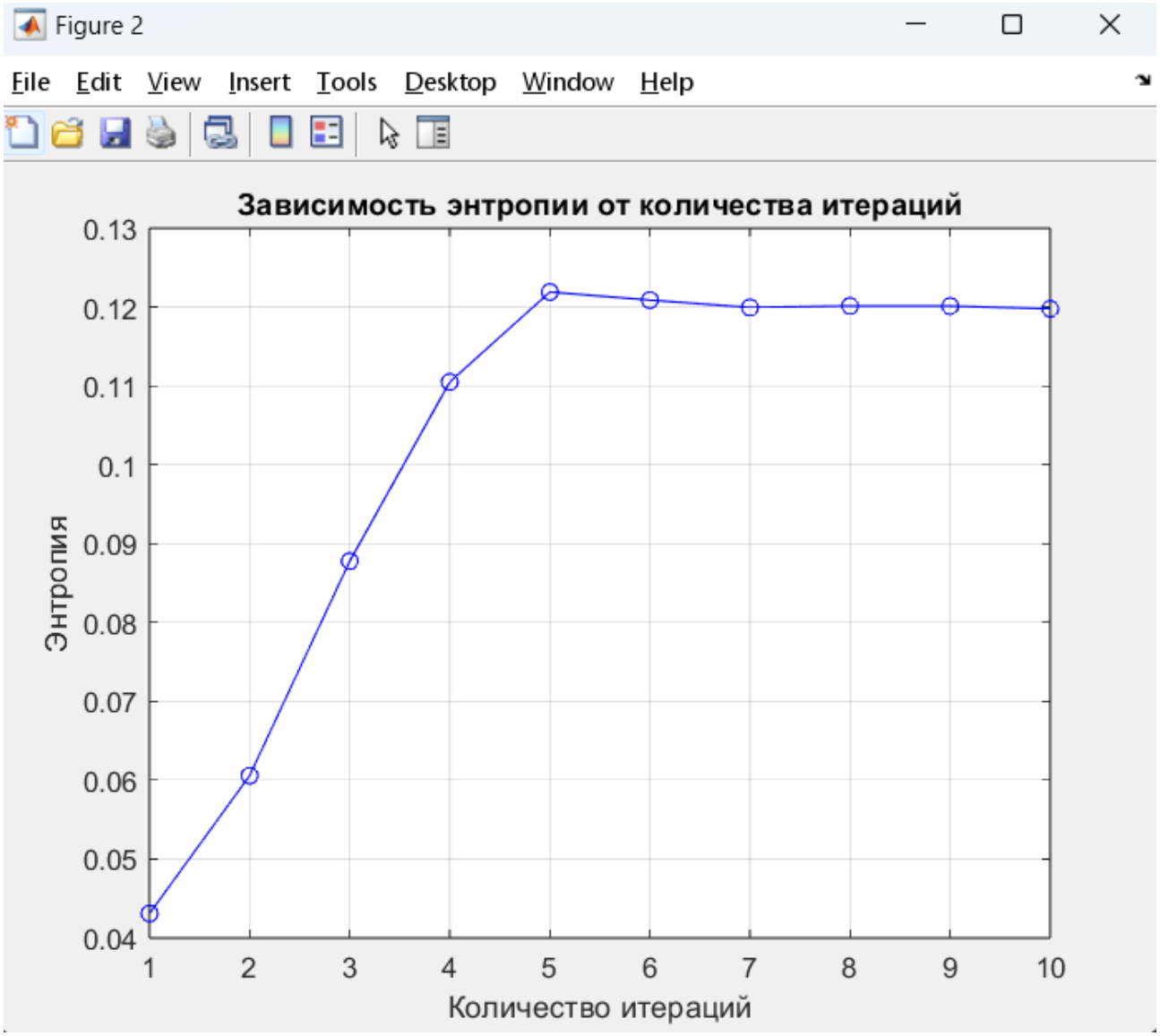

Рисунок 5 – Зависимость энтропии от количества итераций для изображения «Лена»

В результате построения графика, можно увидеть, что постепенное увеличение количества итераций увеличивает значение энтропии, что говорит о более высоком качестве скремблирования изображения, но после определенного количества итераций значение энтропии доходит до плато, что означает, что последующие итерации не имеют смысла.

На рисунке 6 показано стандартное изображение «Cameraman».

Рисунок 6 – Стандартного тестового изображения «Камерамэн»

Итерации изображения «Камерамэн» показаны на рисунке 7.

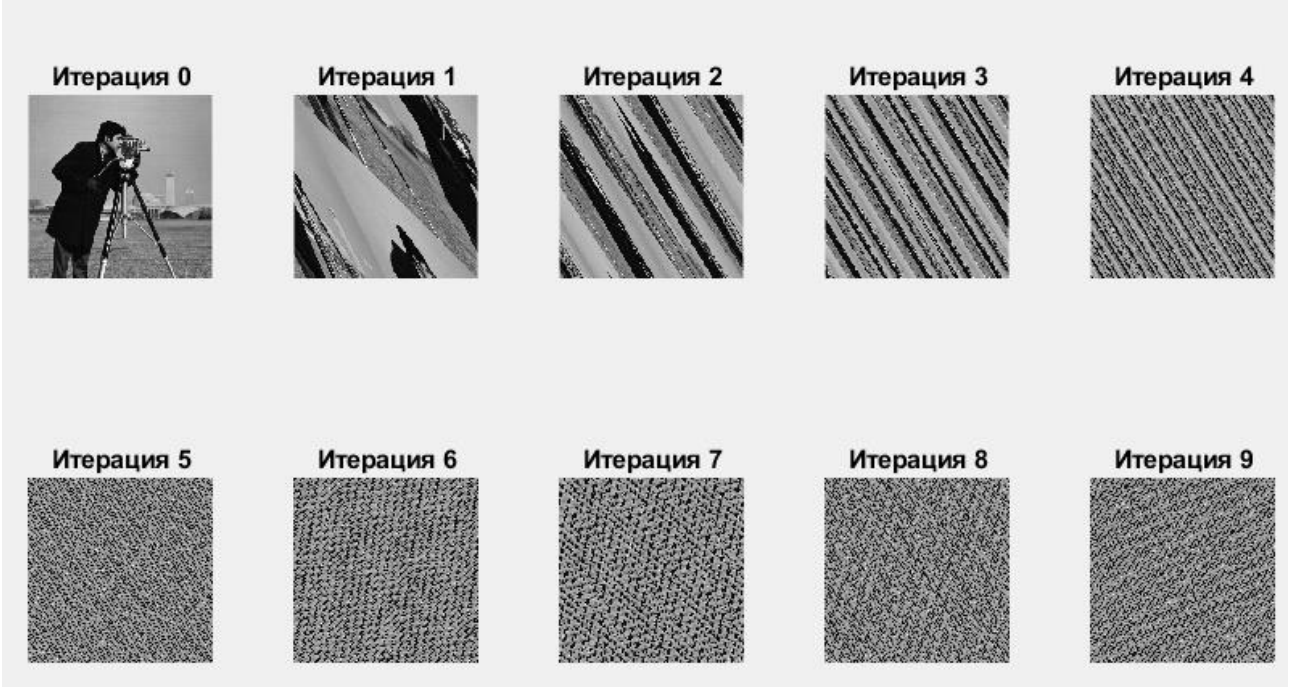

Рисунок 7 – Результат итераций с помощью алгоритма Арнольда изображения «Камерамэн»

Результат итерации изображения «Камерамэн» показан на рисунке 8.

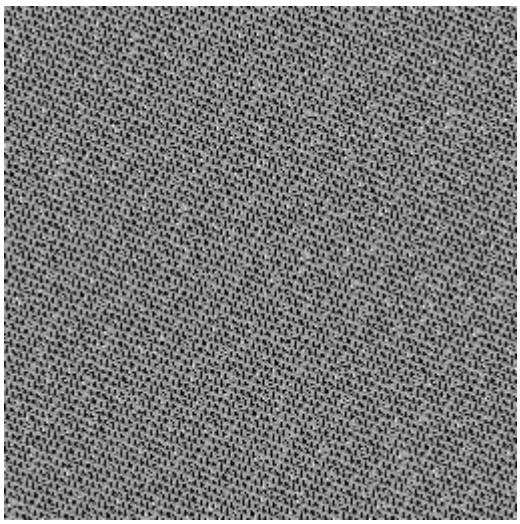

Рисунок 8 – Последний этап итераций с помощью алгоритма Арнольда изображения «Камерамэн»

График зависимости энтропии от количества итераций для изображения «Камерамэн» показан на рисунке 9.

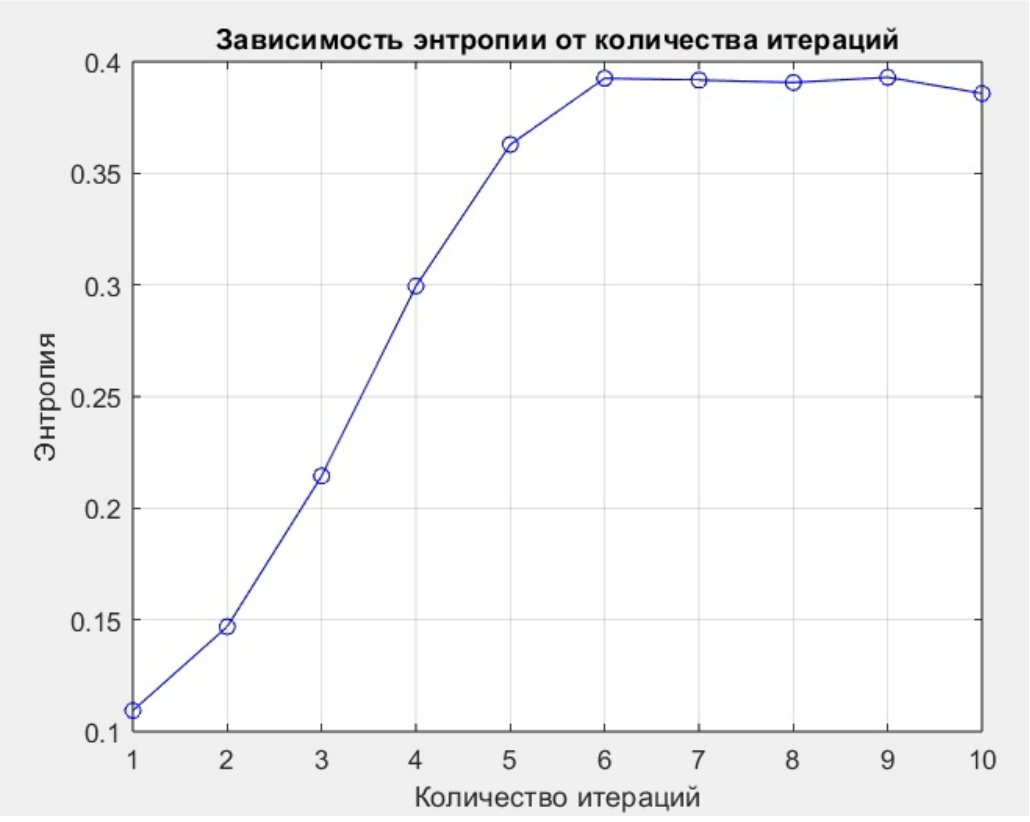

Рисунок 9 – Зависимость энтропии от количества итераций для изображения «Камерамэн»

На рисунке 10 показано стандартное изображение «Pepper».

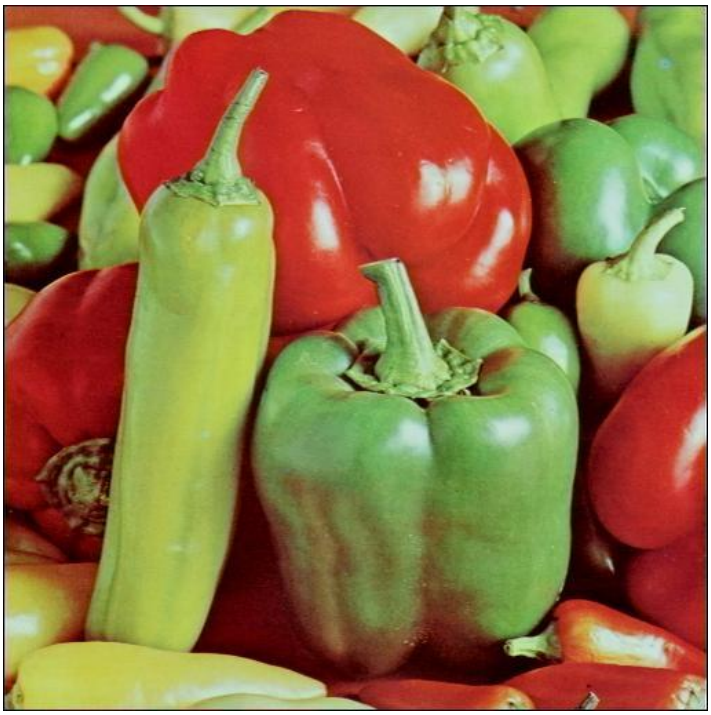

Рисунок 10 – Стандартного тестового изображения «Перец»

Итерации изображения «Перец» показаны на рисунке 11.

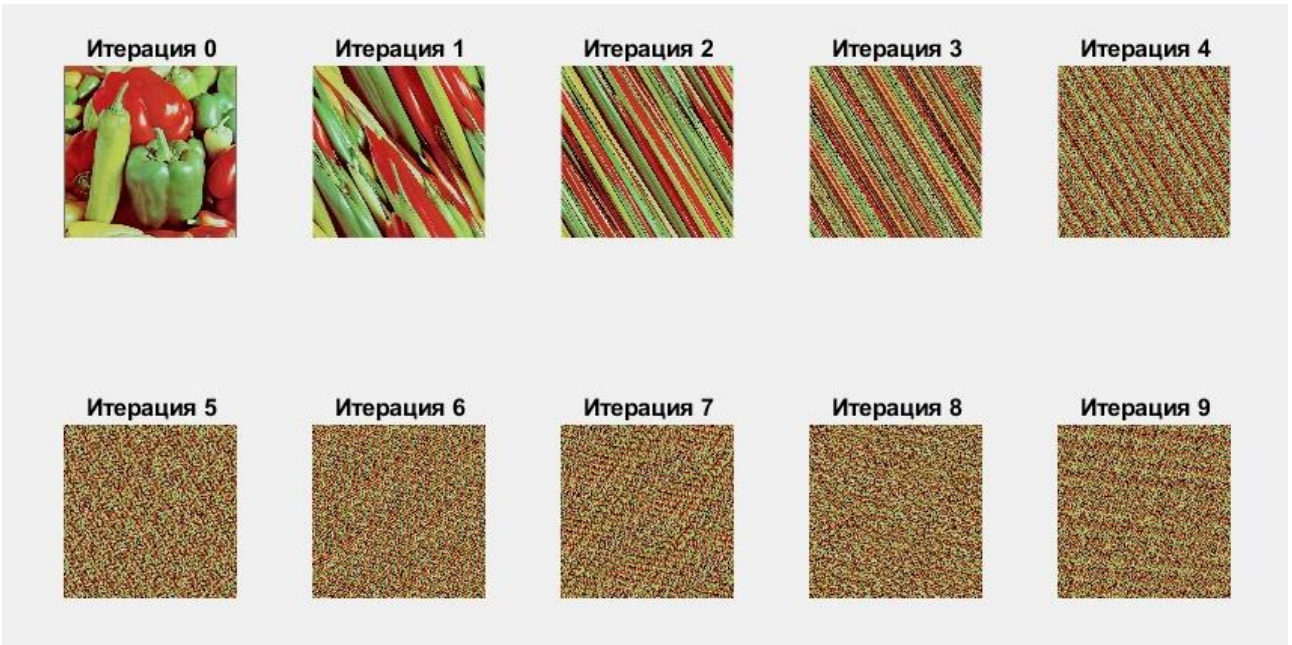

Рисунок 11 – Результат итераций с помощью алгоритма Арнольда изображения «Перец»

Результат итерации изображения «Перец» показан на рисунке 12.

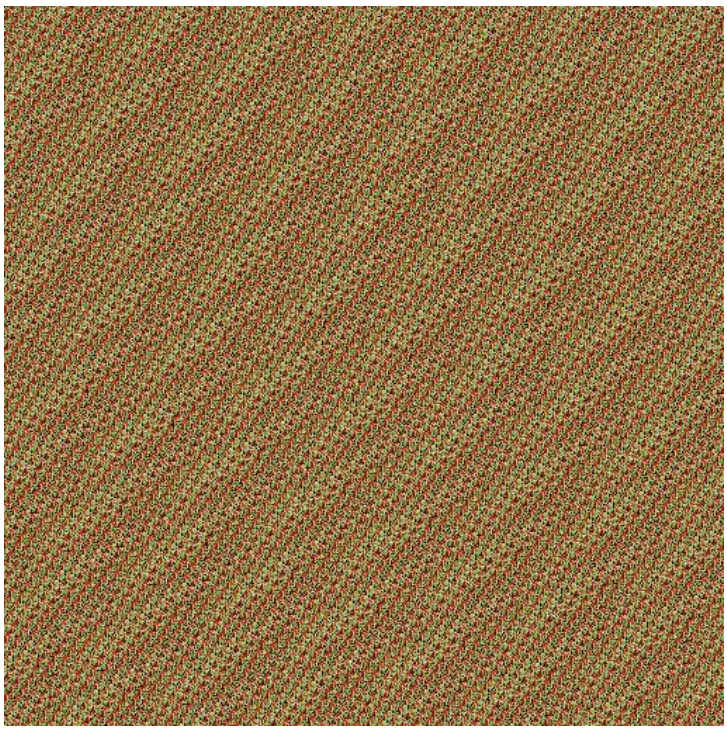

Рисунок 12 – Последний этап итераций с помощью алгоритма Арнольда изображения «Перец»

График зависимости энтропии от количества итераций для изображения «Перец» показан на рисунке 13.

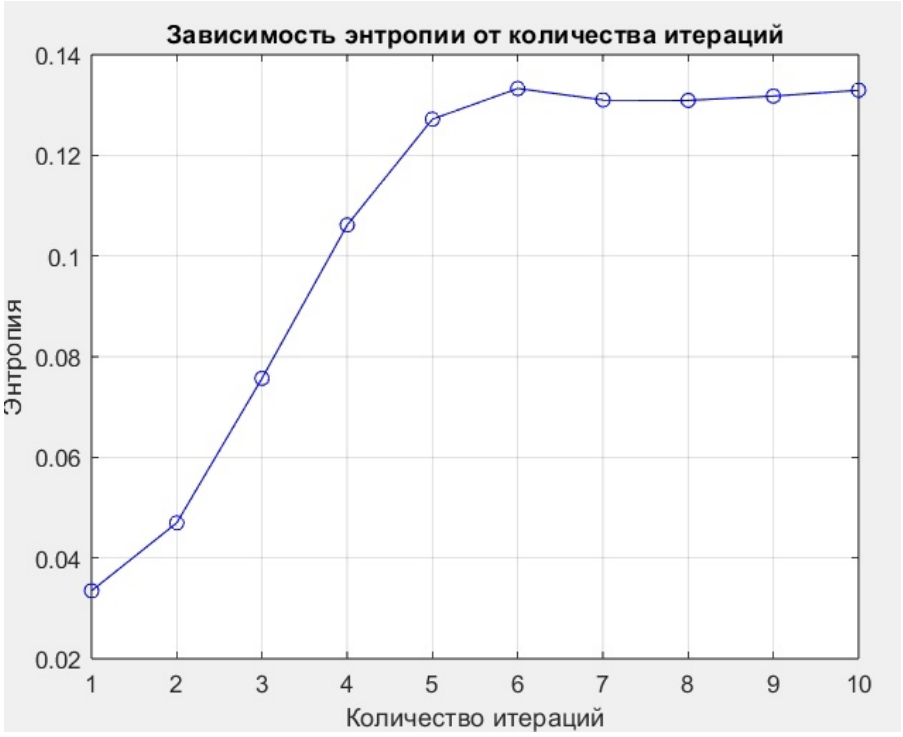

Рисунок 13 – Зависимость энтропии от количества итераций для изображения «Перец»

Алгоритм был написан с разделением на три файла, листинги которых представлены в приложении Б.

В результате визуального анализа алгоритма Арнольда, можно сделать вывод о том, что достаточное количество итераций изображения является 5. Дальнейшие итерации лишь приводят к лишним вычислительным затратам.

Для более лучшего анализа в таблице 1 приведены результаты рассмотрения метрик Ds, DSF, GSF и энтропии при разном количестве итерации изображений.

Таблица 1 – Анализ метрик и энтропии при разном количестве итераций

Изображение	№ итерации	Ds	DSF	GSF	Энтропия
Lena	1	9,47	7,55	1,25	0,18
	2	16,75	7,67	2,18	0,27
	3	38,14	7,5	5,09	0,32
	4	124,85	7,52	16,6	0,35
	5	637,02	7,5	84,94	0,35
	6	826,89	7,50	110,19	0,36
	7	483,39	7,51	64,32	0,36
	8	766,11	7,52	101,94	0,36
	9	699,33	7,51	93,14	0,35
Cameramen	1	8,32	7,6	1,09	0,2
	2	11,22	7,35	1,53	0,27
	3	31,02	7,38	4,2	0,34
	4	268,27	7,5	35,76	0,38
	5	853,91	7,67	111,38	0,36
	6	461,84	7,33	63,02	0,42
	7	388,73	7,31	53,16	0,41
	8	707,37	7,54	93,8	0,39
	9	648,95	7,67	84,6	0,36
Pepper	1	11	7,69	1,43	0,24
	2	23,5	7,56	3,11	0,31
	3	49,4	7,51	6,58	0,38
	4	116,76	7,49	15,58	0,36
	5	197,51	7,66	25,78	0,34
	6	161,51	7,56	21,36	0,39
	7	223	7,53	29,62	0,39
	8	218,87	7,55	29	0,37
	9	180,62	7,66	23,57	0,34

На основании данных можно увидеть, что этот алгоритм обладает высокой эффективностью скремблирования нарастающей с каждой итерацией до установления плато, примерно на 5 итерации. Увеличение энтропии изображения тоже выходит на плато вместе с эффективностью скремблирования, что означает что эти параметры коррелируют друг с другом. Целесообразно использовать этот алгоритм до 5 итерации.

5. Прайм-алгоритм перетасовки

Р-прайм алгоритм перестановки является эффективным методом скремблирования изображений, разработанным для обеспечения защиты цифровых данных и обработки изображений. Этот алгоритм позволяет перемешивать пиксели изображения в соответствии с определенным ключом перестановки, создавая тем самым псевдослучайную последовательность, которая затрудняет восстановление оригинального изображения без знания ключа.

На рисунке 14 изображено блочное представление процесса скремблирования изображения с помощью прайм-алгоритма перетасовки.

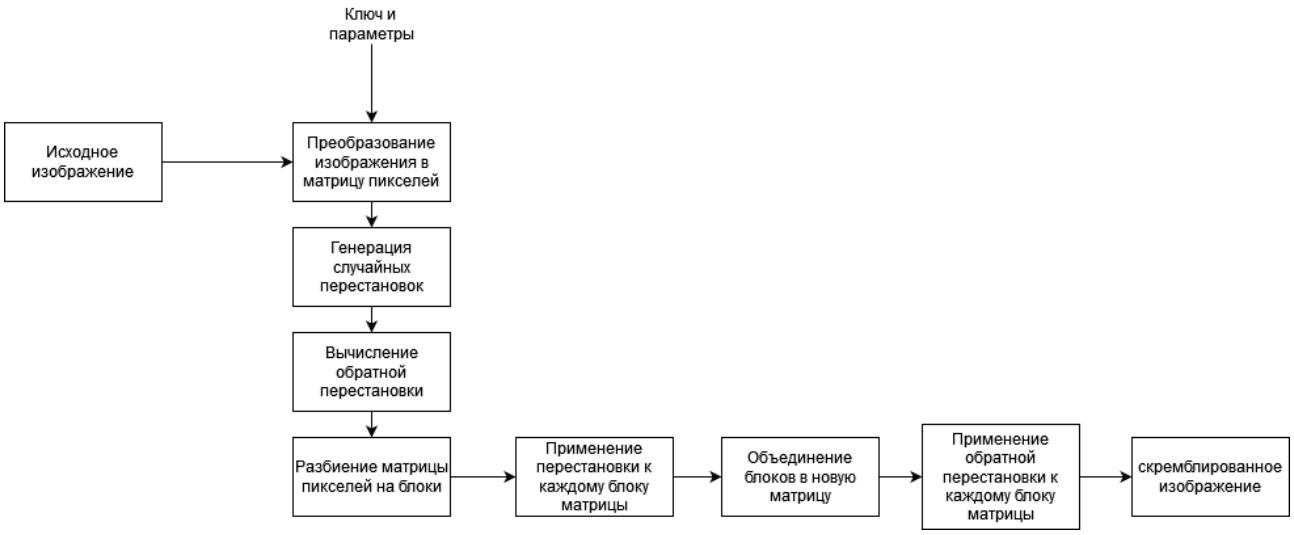

Рисунок 14 – Схема алгоритма прайм-алгоритма перетасовки

Результат работы прайм-алгоритма для скремблирования изображения «Лена» показан на рисунке 15.

Рисунок 15 – Результат работы прайм-алгоритма для изображения «Лена»

Чтобы продемонстрировать влияние знания ключа на расшифровку изображения, было взято зашифрованное изображение «Лены», показанное на рисунке 15, которое было расшифровано с использованием двух неправильных ключей, которые отличаются символами. Результат показан на рисунке 16-17. Визуально изображения не сильно отличаются, но при этом изображение было расшифровано некорректно.

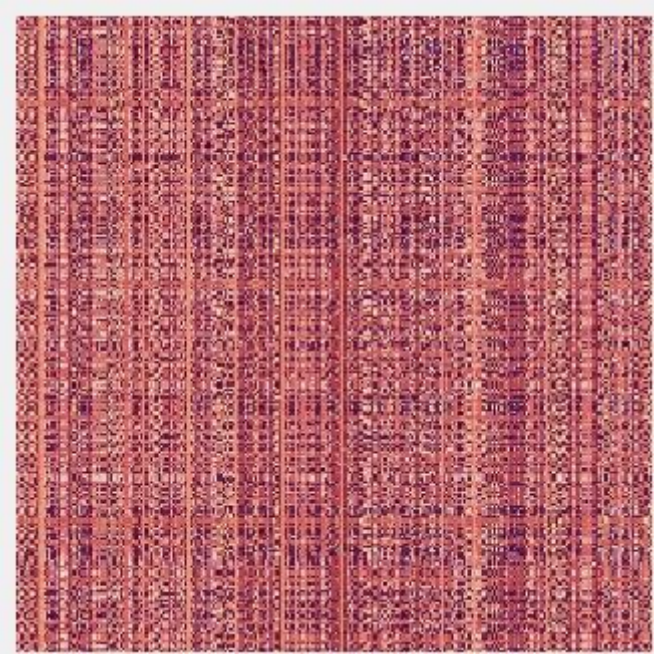

Рисунок 16 – Результат расшифровки зашифрованного изображения «Лена» с использованием неправильного ключа, использован один неправильный символ

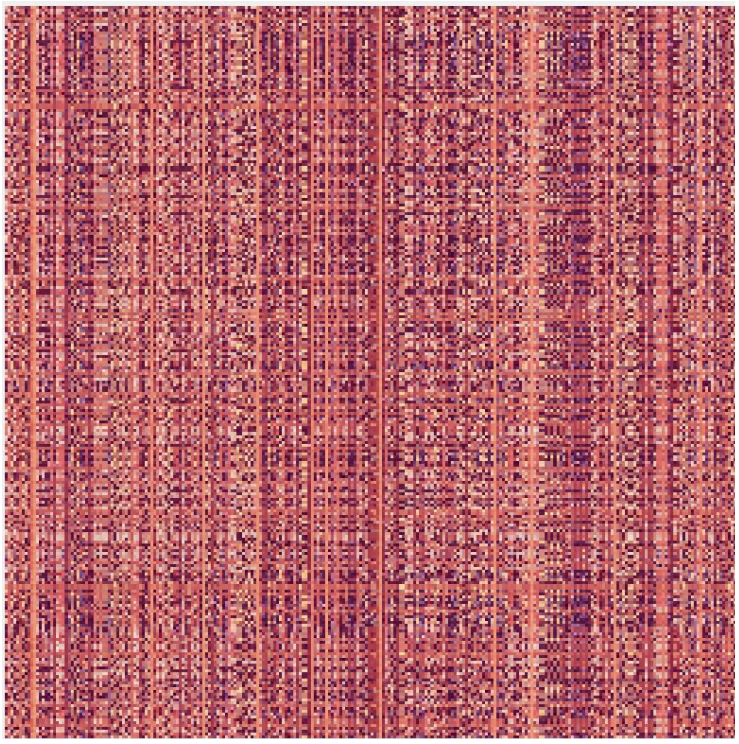

Рисунок 17 – Результат расшифровки зашифрованного изображения «Лена» с использованием неправильного ключа, использовано несколько неправильных символов

Для лучшей демонстрации результатов перестановки для скремблирования изображения «Камерамэн» на рисунках 17-19 показаны результаты перестановки и строк, и столбцов, только столбцов и только строк соответственно.

Результат работы прайм-алгоритма, где происходит перестановка и строк, и столбцов, для скремблирования изображения «Камерамэн» показан на рисунке 17.

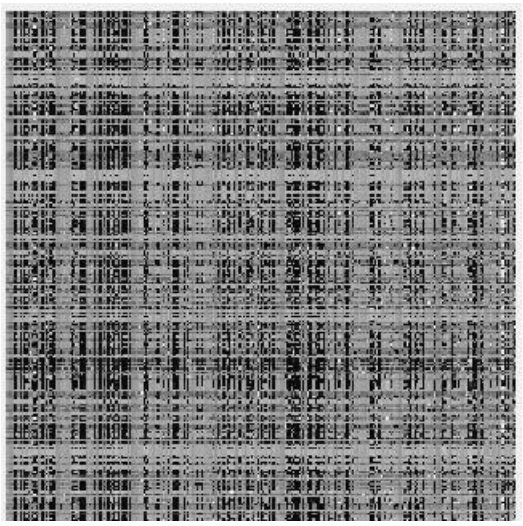

Рисунок 17 – Результат работы прайм-алгоритма для изображения «Камерамэн», перестановка и строк, и столбцов

Результат работы прайм-алгоритма, где происходит перестановка только столбцов, для скремблирования изображения «Камерамэн» показан на рисунке 18.

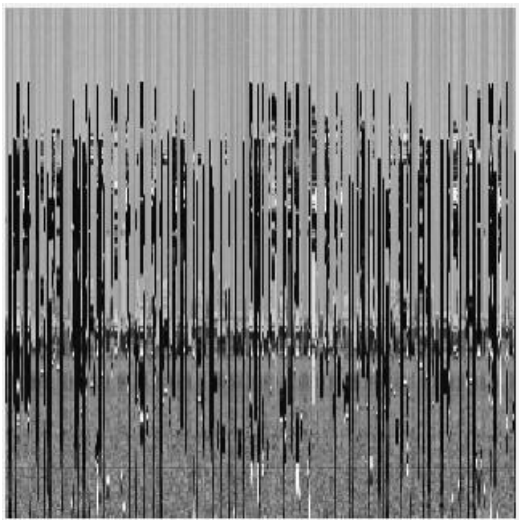

Рисунок 18 – Результат работы прайм-алгоритма для изображения «Камерамэн», перестановка только столбцов

Результат работы прайм-алгоритма, где происходит перестановка только строк, для скремблирования изображения «Камерамэн» показан на рисунке 19.

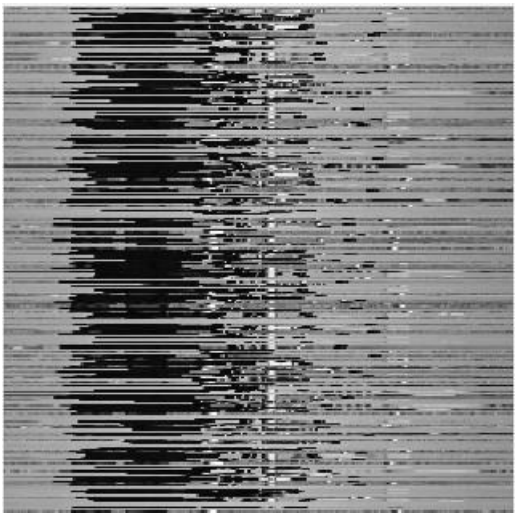

Рисунок 19 – Результат работы прайм-алгоритма для изображения «Камерамэн», перестановка только строк

Результат работы прайм-алгоритма для скремблирования изображения «Перец» показан на рисунке 20.

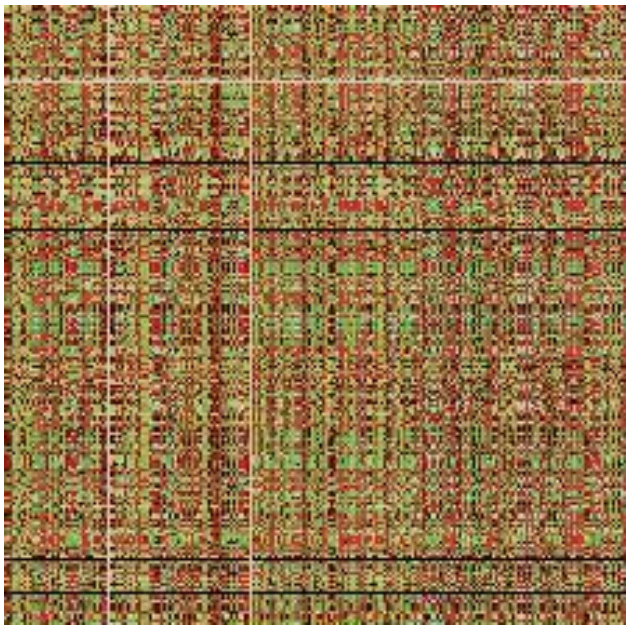

Рисунок 20 – Результат работы прайм-алгоритма для изображения «Перец»

Алгоритм был написан с разделением на три файла, листинги которых представлены в приложении В.

Для анализа зависимости значений метрик от вида перестановки была построена таблица 2, где показаны численные значения метрик.

Таблица 2 – Анализ метрик и энтропии при разных видах перестановки

Изображение	Вариант перестановки	Ds	DSF	GSF	Энтропия
Cameramen	Только строки	19,32	7,21	2,68	0,21
	Только столбцы	21,97	7,38	2,98	0,21
	Строки и столбцы	121,39	8,01	15,15	0,25
Lena	Только строки	40,07	7,55	5,3	0,2
	Только столбцы	119,25	8,14	14,65	0,19
	Строки и столбцы	392,12	8,13	48,24	0,22
Pepper	Только строки	66,39	8,13	8,17	0,22
	Только столбцы	84,63	8,15	10,39	0,22
	Строки и столбцы	355,76	8,32	42,77	0,25

В результате анализа таблицы 2 можно сделать вывод о том, что вид перестановки только строк и только столбцов менее эффективнее, чем перестановка и со строками, и со столбцами.

Так же можно увидеть, что показатели при перестановки со строками и столбцами соответствуют примерно 4 итерации алгоритма арнольда, при меньших вычислительных затратах. Это говорит о высокой эффективности данного метода.

6. Алгоритм кубика Рубика

Алгоритм кубика Рубика это способ кодирования информации, основанный на использовании свойств поворота кубика Рубика.

Главным фактором применения данного алгоритма является обеспечение обратимой трансформации изображения и возможность создания стего-изображения, путем внедрения информации в другое изображение.

Алгоритм шифрования на основе кубика Рубика имеет ограничения, связанные с длинной ключа и сложностью поворота слоев. Так же этот алгоритм использует ключи разной длинны: 16, 32, 64, 128 и 256 байт, и работает с массивами байт, полученные путем преобразования шифруемого изображения.

Принцип действия алгоритма заключается в следующих шагах:

1.1. Изображение разбивается на блоки из одного пикселя (3x3 или nxn).

1.2. Последовательно выбираются 54 (так как в кубике 54 элемента) блока и преобразуется в 6 граней по аналогии с кубом путем индексирования, при этом первый индекс – номер грани, второй – номер элемента в пределах этой грани, как показано на рисунке 21.

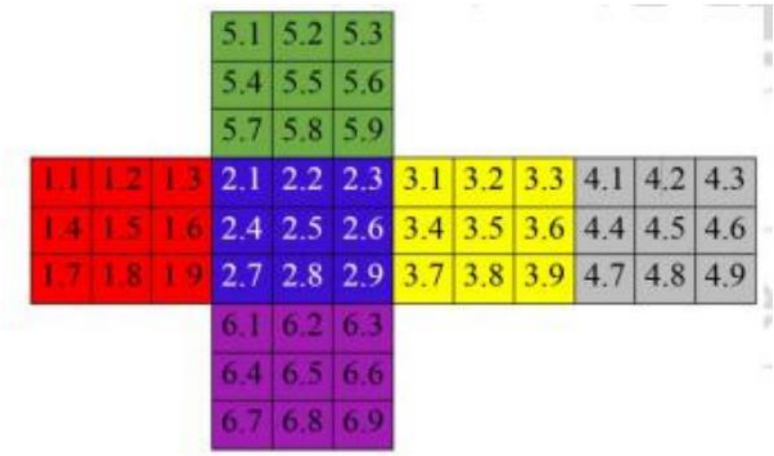

Рисунок 21 – Индексирование блоков в виде кубика Рубика

1.3. Применяются операции, сходные с вращением кубика Рубика. Для этого необходимо задать:

M_p – размер блока, в данном проекте этот параметр выбирается в зависимости от размера изображения. Однако, чтобы обеспечить эффективность скремблирования, размер блока обычно выбирается таким образом, чтобы он делил оба измерения исходного изображения без остатка;

R_p – количество поворотов и направления кубика. Чем больше поворотов применяется к изображению, тем более сложным и надежным будет скремблирование. Однако, слишком большое количество поворотов может привести к потере информации или деградации качества изображения. Для достижения качественного результата при скремблировании методом кубика рубика обычно достаточно применить от 10 до 20 поворотов. Это количество обычно позволяет достаточно хорошо перемешать пиксели изображения, обеспечивая высокую степень защиты от восстановления исходного изображения без ключа;

R_r – определяет одинаковы ли параметры вращения будут применяться ко всем кубикам. Эти параметры определяют направление, по часовой или против часовой, что определяет способ перемешивания пикселей внутри блока, и угол поворота каждого блока пикселей изображения, что определяет на сколько градусов будет повернут каждый блок пикселей, обычно используются углы поворота в диапазоне от 90 до 180 градусов. Эти параметры позволяют достаточно сложным образом перемешивать пиксели изображения, делая процесс восстановления исходного изображения без ключа крайне сложным, то есть повышая надежность шифрования.

1.4. Зашифрованная информация дешифруется путем выполнения шагов алгоритма в обратном порядке.

Схема работы алгоритма с применением свойств кубика Рубика показан на рисунке 22.

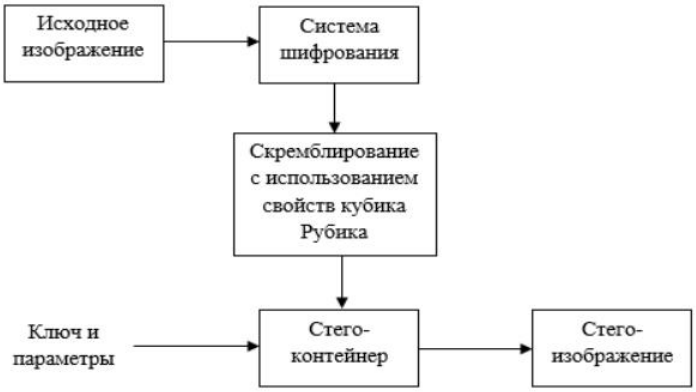

Рисунок 22 – Схема алгоритма кубика Рубика

Результат работы алгоритма кубика Рубика для скремблирования изображения «Лена» показан на рисунке 23.

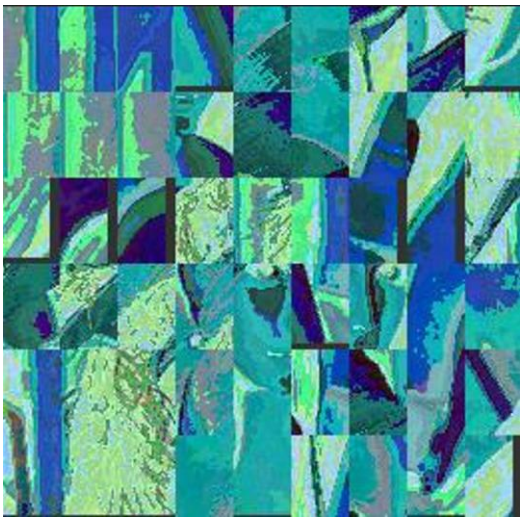

Рисунок 23 – Результат работы алгоритма кубика Рубика для скремблирования изображения «Лена»

Для наглядности работы алгоритма на рисунке 24 показан результат дескремблирования зашифрованного изображения «Лены», который был получен на рисунке 23.

Рисунок 24 – Результат дескремблирования зашифрованного изображения «Лена»

Результат работы алгоритма кубика Рубика для скремблирования изображения «Камерамэн» с размерами 32*32 пикселей показан на рисунке 25.

Рисунок 25 – Результат работы алгоритма кубика Рубика для скремблирования изображения «Камерамэн»

Результат работы алгоритма кубика Рубика для скремблирования изображения «Перец» с размерами 1024*1024 пикселей показан на рисунке 26.

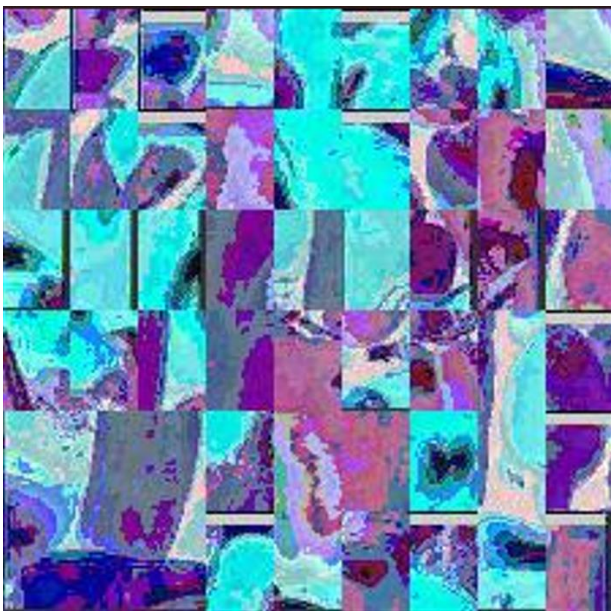

Рисунок 26 – Результат работы алгоритма кубика Рубика для скремблирования изображения «Перец»

Как видно из рисунков 23-26, данный алгоритм способен обрабатывать изображения с разными размерами.

Алгоритм был написан с разделением на три файла, листинги которых представлены в приложении Г.

В результате анализа алгоритма кубика Рубика можно сделать вывод о том, что размер блока имеет прямое влияние на качество скремблирования изображени. Большие блоки позволяют более сильно перемешивать пиксели изображения, что обеспечивает более надежную защиту от восстановления исходного изображения без ключа. Однако слишком большой размер блока может привести к потере некоторой информации и деградации качества изображения.

С другой стороны, маленькие блоки позволяют более точно перемешать пиксели и сохранить более высокое качество изображения. Однако слишком маленький размер блока может привести к недостаточной степени защиты от восстановления исходного изображения без ключа.

Поэтому выбор размера блока зависит от конкретных требований к защите конфиденциальности и качеству изображения. Обычно рекомендуется выбирать

размер блока, который позволяет добиться баланса между степенью защиты и качеством изображения. Например, для изображений с высокой степенью детализации и разрешением 512x512, размер блока может быть выбран в диапазоне от 16x16 до 32x32 пикселей, что обеспечивает достаточно высокое качество скремблирования и сохранение информации.

Для анализа зависимости значений метрик от изображения была построена таблица 3, где показаны численные значения метрик.

Таблица 3 – Анализ метрик и энтропии разных изображений

Изображение	Ds	DSF	GSF	Энтропия
Cameramen	17,15	8,08	2,12	0,19
Lena	16,03	7,76	2,06	0,2
Pepper	14,34	6,99	2,05	0,22

Проанализировав данные можно понять что алгоритм обладает малой эффективностью и должен использоваться совместно с другими алгоритмами скремблирования.

ЗАКЛЮЧЕНИЕ

В рамках данного проекта было выполнено моделирование алгоритмов скремблирования для исходного изображения Лена размером NxN. Исходное изображение было успешно выведено в MATLAB с использованием команды imshow.

Для моделирования алгоритмов скремблирования были использованы методы, описанные в источнике [1]. Были применены различные алгоритмы скремблирования, и для каждого из них были получены скремблированные изображения.

Для оценки каждого алгоритма была проведена визуальная оценка, а также использованы метрики, представленные в работе. Результаты оценки были занесены в таблицу.

Выполнение данного проекта позволило успешно провести моделирование алгоритмов скремблирования для исходного изображения и выбрать наилучший алгоритм, который может быть использован для защиты изображений от несанкционированного доступа.

После изучения различных алгоритмов скремблирования и их применения, был сделан вывод о том, что все они успешно перемешивают изображения. Однако, наилучшую эффективность в этой задаче проявил алгоритм Арнольда на своей интеграции с четвертой по шестую. Этот алгоритм демонстрирует самый высокий коэффициент при оценке его работы и эффективности.

Произведено моделирование и оценивание всех рассмотренных алгоритмов были оценены, где показано, что все алгоритмы работают эффективно и соответствуют поставленным задачам.

Таким образом, на основании проведенных исследований и оценок, можно сделать вывод, что алгоритм Арнольда является наилучшим выбором для скремблирования цифровых изображений, обеспечивая необходимый уровень защиты и эффективности.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Скрэмблирование цифровых изображений А.Н. Земцов, В.Ю. Цыбанов: [Электронный ресурс], URL - https://cyberleninka.ru/article/n/skremblirovanie-tsifrovyh-izobrazheniy/viewer

2. Arnold Transform and Inverse for Image Scrambling: [Электронный ресурс], URL - https://www.mathworks.com/matlabcentral/fileexchange/34983-arnold-transform-and-inverse-for-image-scrambling?tab=discussions%2F1359063

3. Скрэмблирование цифровых изображений БАХРУШИНА Г.И , СИНЬКОВ Д.С, БАХРУШИН А.П: [Электронный ресурс], URL - https://www.elibrary.ru/item.asp?id=35044402

4. Исходное изображение: [Электронный ресурс], URL - http://lenna.org/

ПРИЛОЖЕНИЕ А

Листинг программного кода для оценки алгоритмов

Содержимое файла **efficiency.m**:

```
function main()
    % Загрузка исходного изображения
    file_path_a = 'saved_images/iteration_0_arnold.jpg';
    a = load_image(file_path_a);

    % Загрузка скремблированных изображений из папки
    folder_path = './saved_images';
    results = process_images(folder_path, a);

    % Вывод результатов в виде таблицы и сохранение в Excel
    print_and_save_results_table(results, 'results.xlsx');
end

function image_array = load_image(file_path, target_size)
    image = imread(file_path);
    image = rgb2gray(image); % Convert to grayscale
    if nargin > 1
        image = imresize(image, target_size);
    end
    image_array = image; % Return the resized image array
end

function dsf = calculate_dsf(a, a_prime)
    [m, n] = size(a);
    dsf = sqrt(sum((a(:) - a_prime(:)).^2)) / sqrt((m - 1)^2 +
(n - 1)^2);
end

function o_squared = calculate_o(image)
    k = 8; % размер блока
    [m, n] = size(image);
    o_squared = 0;
    E_E_Bpl = mean(image(:)); % среднее значение для всего
изображения

    for p = 1:k:m
        for l = 1:k:n
            block = image(p:min(p+k-1, m), l:min(l+k-1, n));
            E_Bpl = mean(block(:));
            o_squared = o_squared + ((E_Bpl - E_E_Bpl)^2) / (m
* n);
        end
    end
end

function gsf = calculate_gsf(a, a_prime)
    o1_squared = calculate_o(a);
```

```
        o2_squared = calculate_o(a_prime);
        gsf = o1_squared / o2_squared;
    end

    function efficiency = evaluate_efficiency(a, a_prime)
        dsf = calculate_dsf(a, a_prime);
        gsf = calculate_gsf(a, a_prime);
        efficiency = dsf * gsf;
    end

    function entropy = calculate_entropy(image)
        % Загрузка изображения
        img = image;

        % Методы компрессии
        initial_size = numel(img);
        imwrite(img, 'compressed_image.jpg', 'quality', 50);
        compressed_size = dir('compressed_image.jpg').bytes;
        entropy = compressed_size / initial_size;

        % Удаление временного файла сжатого изображения
        delete('compressed_image.jpg');
    end

    function results = process_images(folder_path, a)
        results = {};
        image_files = dir(fullfile(folder_path, '*.jpg'));
        [m, n] = size(a);

        for i = 1:length(image_files)
            file_name = image_files(i).name;
            file_path_a_prime = fullfile(folder_path, file_name);
            a_prime = load_image(file_path_a_prime, [m, n]); %
Resize 'a_prime' to match 'a'

            dsf = calculate_dsf(a, a_prime);
            gsf = calculate_gsf(a, a_prime);
            efficiency = dsf * gsf;
            entropy = calculate_entropy(a_prime);

            % Store results in a cell array
            results{end+1, 1} = file_name;
            results{end, 2} = efficiency;
            results{end, 3} = dsf;
            results{end, 4} = gsf;
            results{end, 5} = entropy;
        end
    end

    function print_and_save_results_table(results, file_name)
        headers = {'Prime Image', 'Efficiency', 'DSF', 'GSF',
'Entropy'};
```

```
        results_table = cell2table(results, 'VariableNames',
headers);
        disp(results_table);
        writetable(results_table, file_name);
    end
```

ПРИЛОЖЕНИЕ Б

Листинг программного кода для алгоритма Арнольда

Содержимое файла **arnoldAlgorithm.m:**

```
input_image = imread('len_std.jpg'); % Загрузка изображения

max_iterations = 10; % Максимальное количество итераций
entropies = zeros(1, max_iterations); % Вектор для хранения
коэффициентов сжатия
figure('Name', 'Изображения', 'Position', [100, 100, 800,
400]);
for iterations = 0:max_iterations-1
    % Применение алгоритма с текущим количеством итераций
    scrambled_image =
arnoldAlgorithmStraight(input_image,1,1,1,2, iterations);


     % Добавление изображения в форму
    %subplot(2, max_iterations/2, iterations);
    nexttile;
    imshow(scrambled_image);
    title(['Итерация ', num2str(iterations)]);

    % Вычисление коэффициента сжатия
    compression_ratio = EntropyMeasure(scrambled_image);
    entropies(iterations+1) = 1/compression_ratio;
end

% Построение графика
figure;
plot(1:max_iterations, entropies, 'bo-');
xlabel('Количество итераций');
ylabel('Энтропия');
title('Зависимость энтропии от количества итераций');
grid on;
```

Содержимое файла **arnoldAlgorithmReverse.m:**

```
function descrambled_image = descramble_image(scrambled_image,
a, b, c, d, iterations)
    [height, width, ~] = size(scrambled_image);
    N = max(height, width);
    descrambled_image = scrambled_image;

    for iter = 1:iterations
        for y = 1:height
            for x = 1:width
                x_old = mod(d*(x-1) - b*(y-1), N) + 1;
                y_old = mod(-c*(x-1) + a*(y-1), N) + 1;
```

```
                    descrambled_image(y_old, x_old, :) =
scrambled_image(y, x, :);
                end
            end
            scrambled_image = descrambled_image; % Обновляем
скремблированное изображение для следующей итерации
        end
    end
```

Содержимое файла **arnoldAlgorithmStraight.m:**

```
    function scrambled_image = scramble_image(original_image, a,
b, c, d, iterations)
        [height, width, ~] = size(original_image);
        N = max(height, width);
        scrambled_image = original_image;

        for iter = 1:iterations
            for y = 1:height
                for x = 1:width
                    x_new = mod(a*(x-1) + b*(y-1), N) + 1;
                    y_new = mod(c*(x-1) + d*(y-1), N) + 1;
                    scrambled_image(y_new, x_new, :) =
original_image(y, x, :);
                end
            end
            original_image = scrambled_image; % Обновляем
оригинальное изображение для следующей итерации
        end
    end
```

ПРИЛОЖЕНИЕ В

Листинг программного кода для прайм-алгоритм перетасовки

Содержимое файла **primeAlgorithm.m**:

```
    figure('Name', 'Изображения', 'Position', [100, 100, 800,
400]);
    original_image = imread('len_std.jpg'); % Load your image here
    permutation_key = 123456; % Set your permutation key

    nexttile;
    scrambled_image = primeAlgorithmStraight(original_image,
permutation_key);
    imshow(uint8(scrambled_image)); % Display the scrambled image
    imwrite(scrambled_image, "saved_images/prime_res.jpg")
    nexttile;

    unscrambled_image = primeAlgorithmReverse(scrambled_image,
123455);
    imshow(uint8(unscrambled_image)); % Display the unscrambled
image


    compression_ratio = EntropyMeasure(scrambled_image);
    disp(['Коэффициент сжатия: ', num2str(compression_ratio)]);
```

Содержимое файла **primeAlgorithmReverse.m**:

```
    function unscrambled_image = unscramble_image(scrambled_image,
key)
        % Convert the scrambled image to double precision for
calculations
        scrambled_image = double(scrambled_image);

        % Get the size of the scrambled image
        [rows, cols, num_channels] = size(scrambled_image);

        % Initialize unscrambled_image
        unscrambled_image = zeros(size(scrambled_image));

        for ch = 1:num_channels
            % Generate the same row permutation based on the key
            rng(key); % Seed the random number generator with the
key
            row_perm = randperm(rows);
            [~, inv_row_perm] = sort(row_perm);

            % Apply inverse row permutation
            unscrambled_channel = scrambled_image(:,:,ch);
            unscrambled_channel =
unscrambled_channel(inv_row_perm, :);
```

```
            % Generate the same column permutation based on the
new key
            col_key = key + 1; % Use the same different seed for
column permutation
            rng(col_key); % Seed the random number generator with
the new key
            col_perm = randperm(cols);
            [~, inv_col_perm] = sort(col_perm);

            % Apply inverse column permutation
            unscrambled_channel = unscrambled_channel(:,
inv_col_perm);

            % Store the unscrambled channel
            unscrambled_image(:,:,ch) = unscrambled_channel;
        end

        % Convert back to uint8 format for image display
        unscrambled_image = uint8(unscrambled_image);
    end
```

Содержимое файла **primeAlgorithmStraight.m:**

```
    function scrambled_image = scramble_image(image, key)
        % Convert the image to double precision for calculations
        image = double(image);

        % Get the size of the image
        [rows, cols, num_channels] = size(image);

        % Initialize scrambled_image
        scrambled_image = zeros(size(image));

        for ch = 1:num_channels
            % Generate a random permutation for rows based on the
key
            rng(key); % Seed the random number generator with the
key
            row_perm = randperm(rows);

            % Apply row permutation
            scrambled_channel = image(:,:,ch);
            scrambled_channel = scrambled_channel(row_perm, :);

            % Generate a different random permutation for columns
            col_key = key + 1; % Use a different seed for column
permutation
            rng(col_key); % Seed the random number generator with
the new key
            col_perm = randperm(cols);
```

```
            % Apply column permutation
            scrambled_channel = scrambled_channel(:, col_perm);

            % Store the scrambled channel
            scrambled_image(:,:,ch) = scrambled_channel;
        end

        % Convert back to uint8 format for image display
        scrambled_image = uint8(scrambled_image);
    end
```

ПРИЛОЖЕНИЕ Г

Листинг программного кода для алгоритма кубика Рубика

Содержимое файла **rubics.m:**

```
function main()
    % Main script
    input_image_path = './cameraman.png';
    output_image_path = './saved_images/rubics.jpg';

    number_of_actions = 3;
    key = hex2dec('AABBCC'); % Simple key for XOR encryption
    image_cipher = ImageCipher(key);

    % Step 1: Load image
    image = imread(input_image_path);

    [h, w, c] = size(image); % Определение высоты, ширины и
числа каналов изображения
    if c == 1
        % Если изображение черно-белое, дублируем его на три
канала
        image = repmat(image, [1, 1, 3]);
    end

    % Ensure image dimensions are multiples of 18
    [h, w, ~] = size(image);
    if mod(h, 9) ~= 0
        h_new = h + (9 - mod(h, 9));
    else
        h_new = h;
    end
    if mod(w, 9) ~= 0
        w_new = w + (9 - mod(w, 9));
    else
        w_new = w;
    end

    % Create a new image with the adjusted dimensions and fill
with white pixels
    padded_image = uint8(255 * ones(h_new, w_new, 3));
    padded_image(1:h, 1:w, :) = image;
    imwrite(padded_image, "asdf.jpg");
    % Encrypt image
    encrypt_image = image_cipher.encrypt_image(padded_image);

    % Step 2: Split image into 6 parts
    parts = split_image_into_six_parts(encrypt_image);

    % Step 3: Split each part into 9 parts
    sub_parts = cellfun(@split_image_into_nine_parts, parts,
'UniformOutput', false);
```

```
        % Step 4: Assemble Rubik's cube
        cube = assemble_rubiks_cube(sub_parts);

        % Step 5: Apply rotations
        cube = apply_rotations(cube, number_of_actions);

        % Step 6: Flatten cube to image
        result_image = flatten_cube_to_image(cube);

        % Step 7: Save image
        imwrite(result_image, output_image_path);

        % For decryption
        e_parts = split_image_into_six_parts(result_image);
        esub_parts = cellfun(@split_image_into_nine_parts,
e_parts, 'UniformOutput', false);

        ecube = assemble_rubiks_cube(esub_parts);
        ecube = apply_counter_rotations(ecube, number_of_actions);

        eresult_image = flatten_cube_to_image(ecube);

        % Remove padding and decrypt restored image
        restored_image =
image_cipher.decrypt_image(eresult_image);
        restored_image = restored_image(1:h, 1:w, :);  % Remove
padding

        % Display the original, encrypted, and restored images
        figure;
        subplot(1, 3, 1);
        imshow(image);
        title('Original Image');

        subplot(1, 3, 2);
        imshow(result_image);
        title('Encrypted Image');

        subplot(1, 3, 3);
        imshow(restored_image);
        title('Restored Image');
    end

    function parts = split_image_into_six_parts(image)
        [height, width, ~] = size(image);
        width = floor(width / 3);
        height = floor(height / 2);
        parts = cell(1, 6);

        for y = 1:2
            for x = 1:3
```

```
                part = image((y-1)*height+1:y*height, (x-
1)*width+1:x*width, :);
                parts{(y-1)*3 + x} = part;
            end
        end
    end

    function parts = split_image_into_nine_parts(image)
        [height, width, ~] = size(image);
        width = floor(width / 3);
        height = floor(height / 3);
        parts = cell(1, 9);

        for y = 1:3
            for x = 1:3
                part = image((y-1)*height+1:y*height, (x-
1)*width+1:x*width, :);
                parts{(y-1)*3 + x} = part;
            end
        end
    end

    function cube = assemble_rubiks_cube(parts)
        cube = cell(6, 3, 3);

        for i = 1:6
            face = parts{i};
            for y = 1:3
                for x = 1:3
                    cube{i, y, x} = face{(y-1)*3 + x};
                end
            end
        end
    end

    function cube = apply_rotations(cube, number_of_actions)
        for i = 1:number_of_actions
            cube = rotate_face_clockwise(cube, mod(i-1, 6) + 1);
        end
    end

    function cube = rotate_face_clockwise(cube, face_index)
        face = cube(face_index, :, :);
        new_face = cell(3, 3);

        for y = 1:3
            for x = 1:3
                new_face{x, 3-y+1} = face{1, y, x};
            end
        end

        cube(face_index, :, :) = new_face;
        cube = update_adjacent_faces(cube, face_index);
```

```
    end

    function cube = update_adjacent_faces(cube, face_index)
        adjacent_faces = {
            [2, 5, 4, 6]; % Front face (faceIndex = 1)
            [1, 6, 3, 5]; % Top face (faceIndex = 2)
            [2, 5, 4, 6]; % Back face (faceIndex = 3)
            [1, 6, 3, 5]; % Bottom face (faceIndex = 4)
            [1, 2, 3, 4]; % Left face (faceIndex = 5)
            [1, 4, 3, 2]  % Right face (faceIndex = 6)
        };

        face = adjacent_faces{face_index};
        temp = cube(face(1), 3, :);

        for i = 1:3
            cube{face(1), 3, i} = cube{face(2), 3-i+1, 3};
            cube{face(2), 3-i+1, 3} = cube{face(3), 1, 3-i+1};
            cube{face(3), 1, 3-i+1} = cube{face(4), i, 1};
            cube{face(4), i, 1} = temp{1, 1, i};
        end
    end

    function cube = apply_counter_rotations(cube,
number_of_actions)
        for i = number_of_actions:-1:1
            for j = 1:3
                cube = rotate_face_clockwise(cube, mod(i-1, 6) +
1);
            end
        end
    end

    function resultImage = flatten_cube_to_image(cube)
        partSize = size(cube{1, 1, 1});
        width = partSize(2) * 3 * 3;
        height = partSize(1) * 2 * 3;
        resultImage = uint8(zeros(height, width, 3));

        for i = 1:6
            for y = 1:3
                for x = 1:3
                    part = cube{i, y, x};
                    resultImage(get_offset_height(i,
partSize(1)*3) + (y-1)*partSize(1) + 1:get_offset_height(i,
partSize(1)*3) + y*partSize(1), ...
                                get_offset_width(i, partSize(2)*3)
+ (x-1)*partSize(2) + 1:get_offset_width(i, partSize(2)*3) +
x*partSize(2), :) = part;
                end
            end
        end
    end
```

```
function offset = get_offset_width(i, width)
    offset = width * mod(i-1, 3);
end

function offset = get_offset_height(i, height)
    offset = height * floor((i-1) / 3);
end
```