МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
**«Национальный исследовательский
Нижегородский государственный университет им. Н.И. Лобачевского»
(ННГУ)**

**Институт информационных технологий, математики и механики
Кафедра алгебры, геометрии и дискретной математики**

Направление подготовки:
«Фундаментальная информатика и информационные технологии»
Профиль подготовки: «Инженерия программного обеспечения»

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
БАКАЛАВРА**

**Тема:
«Реализация и исследование алгоритма JPEG для сжатия изображений»**

Выполнил:
студент группы: 3821Б1ФИ2
Казанцев Евгений Александрович

Научный руководитель:
к.ф.-м.н., доцент каф. АГДМ
Смирнова Татьяна Геннадьевна

Нижний Новгород
2025

Аннотация

Сжатие изображений — применение алгоритмов сжатия данных к изображениям, хранящимся в цифровом виде. В результате сжатия уменьшается размер изображения, из-за чего уменьшается время передачи изображения по сети и экономится пространство для хранения.

Сжатие изображений подразделяют на сжатие с потерями качества и сжатие без потерь.

Преимущество методов сжатия с потерями над методами сжатия без потерь состоит в том, что первые дают возможность большей степени сжатия, при этом, продолжая удовлетворять поставленным требованиям, а именно — искажения должны быть в допустимых пределах чувствительности органов человеческого восприятия.

Методы сжатия с потерями используются для сжатия аналоговых данных — чаще всего звука или изображений.

В таких случаях распакованный файл может сильно отличаться от оригинала по размеру, но практически неотличим для человека «на слух» и «на глаз» в большинстве применений.

Множество методов фокусируются на физических особенностях органов чувств человека. К примеру психоакустические модели определяют то, как сильно звук может быть сжат без ухудшения воспринимаемого человеком качества звука.

При использовании сжатия с потерями необходимо учитывать, что повторное сжатие обычно приводит к деградации качества. Однако, если повторное сжатие выполняется без каких-либо изменений сжимаемых данных, качество не меняется. Так например, сжатие изображения методом JPEG, восстановление его и повторное сжатие с теми же самыми параметрами не приведет к снижению качества.

ОГЛАВЛЕНИЕ

Введение 4
1 Основные понятия и определения 6
2 Алгоритмы и структуры данных 7
2.1 Преобразование цветового пространства (RGB → YCbCr) 7
2.2 Субдискретизация 8
2.3 Дискретное косинус-преобразование 10
2.4 Двумерное (матричное) DCT 15
2.5 Квантование 18
2.6 Зигзаг-преобразование блоков 21
2.7 RLE-кодирование 23
2.8 Декодирование 24
3 Руководство пользователя 26
3.1 Назначение программы 26
3.2 Интерфейс программы 27
4 Результаты и выводы 35
5 Заключение 45
6 Приложение 47
6.1 Класс подготовки изображения 47
6.2 Класс дикретного косинусного преобразования 49
6.3 Класс квантования 52

Введение

Сжатие информации с потерями — это алгоритмическое преобразование данных, направленное на уменьшение их объёма за счёт допустимых потерь качества. Необходимость сокращения объёма данных актуальна с появлением цифровых технологий и остается актуальной по сей день. Несмотря на развитие технологий хранения и передачи данных, ограничения по объёму памяти и скорости передачи информации по-прежнему требуют эффективных методов сжатия. Объёмы данных продолжают расти, их структура усложняется, а качество — повышается. Это особенно актуально для мультимедийных данных — изображений, видео и звука.

В процессе подготовки и написания работы были использованы как теоретические, так и практико-ориентированные источники. Основу теоретической части составила книга Д. Сэломона «Сжатие данных, изображений и звука» [1], где представлены фундаментальные принципы алгоритмов сжатия, включая алгоритм JPEG и его математическую основу — дискретное косинусное преобразование (DCT). Для понимания базовых алгоритмических структур и оценки эффективности различных методов были использованы материалы из книги «Алгоритмы. Построение и анализ» Т. Кормена и соавторов [2], а также статья А. Н. Земцова и коллег, посвящённая сравнительному анализу методов сжатия на основе DCT и фрактального кодирования [4].

Практическая реализация алгоритма опиралась на официальную документацию библиотеки OpenCV [3], а также на книгу Г. Гарсии и соавторов [5], содержащую конкретные примеры обработки изображений с помощью этой библиотеки. При создании графического интерфейса была использована библиотека PyQt5, с опорой на официальную документацию [6]. Дополнительные технические сведения, связанные с реализацией и настройкой проекта, были получены из официальной документации языка программирования Python 3 [7].

Существует огромное множество методов сжатия данных, основанных на различных предположениях о структуре сжимаемой информации. Все алгоритмы сжатия можно разделить на две основные группы: Сжатие без потерь — предполагает возможность точного восстановления исходных данных после сжатия. Сжатие с потерями — допускает потерю части данных ради значительного сокращения объема. Этот метод часто применяется к изображениям, видео и аудиоданным, где некоторая потеря качества незаметна для восприятия.

В данной работе рассматривается алгоритм сжатия с потерями на примере широко используемого метода JPEG. Алгоритм JPEG основан на использовании дискретного косинусного преобразования (DCT) для преобразования изображения, последующем квантовании и энтропийном кодировании. В рамках данной реализации из рассмотрения исключаются этапы энтропийного кодирования, в частности кодирование Хаффмана, а также альтернативные подходы, такие как вейвлет-преобразования. Этот метод обеспечивает высокую степень сжатия при сохранении приемлемого качества изображения. JPEG стал важным этапом в развитии технологий сжатия данных, объединив достижения математических и инженерных исследо-

ваний своего времени, и дал мощный толчок развитию в области сжатия изображений.

Основной задачей работы является исследование алгоритма JPEG в контексте его математической основы и алгоритмической реализации. Предполагается рассмотреть влияние различных параметров сжатия на качество изображения, а также провести сравнение производительности различных реализаций алгоритма. В качестве возможных модификаций рассматриваются методы адаптивного квантования и постобработки изображения после декодирования.

Целью работы является изучение развития алгоритмов сжатия с потерями, начиная с фундаментальных математических открытий (например, дискретного косинусного преобразования), простых алгоритмов сжатия без потерь, и заканчивая созданием аналогичного алгоритма JPEG. В этой работе реализуется собственный алгоритм, основанный на ключевых этапах стандарта JPEG. Его структура почти аналогична базовому варианту. В дальнейшем по тексту он будет обозначаться как “модификация алгоритма JPEG” или просто “JPEG”. Так же планируется разработать программу с интерфейсом, позволяющую проводить эксперименты по сжатию изображений с использованием модификации алгоритма JPEG. Программа должна обладать следующими функциями:

- кодирование изображения методом JPEG с возможностью настройки параметров квантования и качества;
- визуализация результатов кодирования и декодирования;

Для достижения поставленной цели сформулированы следующие задачи:

- изучить теоретическую основу алгоритма JPEG, включая дискретное косинусное преобразование и квантование;
- реализовать алгоритм JPEG на языке программирования Python с использованием библиотек OpenCV, numpy, scipy;
- разработать интерфейс для управления параметрами алгоритма и отображения результатов, используя библиотеку PyQt5;
- провести эксперименты на различных изображениях, оценить влияние параметров на степень сжатия и качество изображения;
- проанализировать результаты экспериментов и сделать выводы о целесообразности использования различных модификаций алгоритма.

1 Основные понятия и определения

Перед тем как углубиться в описание алгоритма JPEG, важно рассмотреть основные понятия, лежащие в основе цифровой обработки и сжатия изображений.

Цифровое изображение — двумерный массив, где каждая ячейка представляет собой пиксель.

Таблица 1. Упрощенная схема цифрового изображения как двумерного массива пикселей

P(0,0)	P(0,1)	P(0,2)	P(0,3)
P(1,0)	P(1,1)	P(1,2)	P(1,3)
P(2,0)	P(2,1)	P(2,2)	P(2,3)
P(3,0)	P(3,1)	P(3,2)	P(3,3)

Пиксель (pixel) — наименьшая единица изображения, содержащая информацию о цвете или яркости.

Таблица 2. Представление пикселей с помощью RGB-значений и их цветов

RGB значения	**Цвет**
RGB(255,0,0)	●
RGB(0,255,0)	●
RGB(0,0,255)	●

Наиболее распространённой моделью представления цветных изображений является RGB.

RGB (Red, Green, Blue) — аддитивная цветовая модель, в которой каждый цвет формируется путём смешивания трёх базовых компонентов: красного, зелёного и синего. Эта модель близка к принципу работы матриц дисплеев и сенсоров цифровых камер, поэтому RGB часто используется как исходный формат цветовых изображений.

В зависимости от глубины цвета, каждый пиксель кодируется определённым числом бит. Например, в изображении с глубиной 24 бита (8 бит на каждый из каналов RGB) один пиксель может представлять более 16 миллионов оттенков. С увеличением разрешения и глубины цвета возрастает и общий объём изображения, что создаёт необходимость в эффективном сжатии данных.

Однако при сжатии изображений RGB-пространство не является наиболее эффективным. Это связано с тем, что RGB не разделяет яркость и цветовую информацию, а человеческий глаз по-разному чувствителен к этим компонентам. В связи с этим перед сжатием изображения, как правило, преобразуются в цветовое пространство **YCbCr**, где **Y** отражает яркость (luminance), а **Cb** и **Cr** — цветовые отклонения от серого (chrominance).

Также необходимо упомянуть несколько ключевых понятий, тесно связанных с задачами кодирования изображений:

Разрешение изображения — количество пикселей по горизонтали и вертикали.

Избыточность — повторяющаяся или избыточная информация, которую можно исключить без потери качества или с минимальными потерями.

Энтропия — мера неопределённости или информационной насыщенности данных, определяющая нижнюю границу возможной степени сжатия.

Формат **JPEG** представляет собой стандарт кодирования изображений с потерями, определяющий последовательность преобразований, квантования и кодирования данных. Разработка формата началась в 1986 году с формированием международной рабочей группы под названием *Joint Photographic Experts Group*. На тот момент существовало множество несовместимых форматов для хранения графики, что затрудняло обмен изображениями между системами. JPEG стал первым по-настоящему универсальным и широко распространённым стандартом сжатия фотоизображений. Официальная спецификация была опубликована в 1992 году и используется до сих пор.

2 Алгоритмы и структуры данных

2.1 Преобразование цветового пространства (RGB → YCbCr)

Перед сжатием изображений в формате JPEG зачастую используется преобразование из цветового пространства RGB в YCbCr. Это обусловлено как особенностями восприятия цвета человеком, так и требованиями алгоритмов сжатия.

Такое разделение позволяет применить хрома-субдискретизацию — уменьшение разрешения цветовых компонент — без заметного ухудшения качества изображения. Для RGB с диапазоном значений от 0 до 255, преобразование в YCbCr выполняется по следующим формулам:

$$\begin{aligned} Y &= 0.299R + 0.587G + 0.114B; \\ Cb &= -0.168736R - 0.331264G + 0.5B + 128; \\ Cr &= 0.5R - 0.418688G - 0.081312B + 128. \end{aligned} \tag{1}$$

Каждое из этих вычислений требует константного числа операций (умножений и сложений). Поскольку обработка всех пикселей независима, общее время работы линейно зависит от их количества $O(n)$.

Здесь значения R, G, B принимаются в диапазоне $[0, 255]$. Смещение на 128 единиц в формулах для Cb и Cr необходимо для корректного представления как положительных, так и отрицательных значений в беззнаковом формате (unsigned int).

Поскольку компоненты Cb и Cr менее значимы для восприятия, их можно хранить с

пониженным разрешением. Наиболее распространённый формат — 4:2:0, при котором на каждые 4 пикселя хранится 4 значения яркости Y и по 1 значению Cb и Cr. Это даёт значительное уменьшение объёма данных без существенного ущерба для качества изображения.

Преимущества YCbCr

- Дает возможность сжатия на уровне разрешения без видимых потерь качества;
- Разделение яркости и цвета позволяет применять различные методы компрессии к компонентам Y и CbCr;
- Поддерживается большинством стандартов обработки изображений и видео.

Таким образом, преобразование RGB $\rightarrow$ YCbCr является ключевым шагом в JPEG и других алгоритмах сжатия, позволяющим использовать особенности восприятия цвета человеком для достижения более высокой степени сжатия.

2.2 Субдискретизация

На втором этапе сжатия изображения применяется методика, известная как субдискретизация цветности (англ. chroma subsampling). Её ключевая идея заключается в сокращении объёма данных, содержащих информацию о цвете, путём уменьшения пространственного разрешения цветовых компонентов. Это возможно благодаря физиологической особенности зрительной системы человека: она в гораздо большей степени чувствительна к деталям яркости (компонент Y — luminance), чем к изменениям цветовых оттенков (компоненты Cb и Cr — chrominance).

В практической реализации данный метод означает, что цветовая информация (Cr и Cb) кодируется с меньшей точностью по сравнению с яркостной. Существует несколько вариантов схем хрома-субдискретизации, которые различаются по степени уменьшения разрешения цветовых компонентов:

4:4:4 — нет сжатия, все компоненты Y, Cb и Cr сохраняют оригинальное разрешение.

4:2:2 — разрешение Cb и Cr уменьшается в два раза по сравнению с Y.

4:2:0 — стандарт для большинства сжатых форматов, где разрешение Cb и Cr уменьшается в два раза как по вертикали, так и по горизонтали.

Схема 4:2:0 является наиболее распространенной для видео и изображений, так как она дает хороший баланс между качеством и сжатием. Она предполагает, что на каждые четыре пикселя яркости (расположенных в виде блока 2×2) приходится всего один усреднённый пиксел Cr и один усреднённый пиксел Cb. То есть цветовые значения вычисляются как средние значения для группы пикселей, и это значение применяется ко всей группе. В результате разрешение цветовых компонентов уменьшается в четыре раза (на 75%) по сравнению с яркостной компонентой.

Для визуализации можно представить исходное изображение как состоящее из трёх равнозначных слоёв:

Y (яркость),

Cb (синий цветовой компонент),

Cr (красный цветовой компонент).

До субдискретизации каждый из этих компонентов имел одинаковое разрешение и соответственно одинаковый вклад в общий объём данных:

$$Y + Cb + Cr = 1 + 1 + 1 = 3 \text{ единицы информации.} \tag{2}$$

После применения схемы 4:2:0, каждая из цветовых компонент уменьшается до $\frac{1}{4}$ исходного размера, а яркостная остаётся без изменений:

$$Y + \frac{1}{4}Cb + \frac{1}{4}Cr = 1 + 0.25 + 0.25 = 1.5 \text{ единицы информации.} \tag{3}$$

Рисунок 1. Как происходит преобразование

Таким образом, достигается двукратное уменьшение общего объёма данных, подлежащих хранению или передаче.

Что важно, такое преобразование происходит практически без визуальных потерь качества. Несмотря на то, что цветовые данные редуцируются, человеческий глаз практически не воспринимает различие между оригинальным и сжатым изображением. Именно поэтому субдискретизация цветности широко применяется в большинстве алгоритмов сжатия изображений и видео (JPEG, MPEG, H.264), где критически важно достичь компромисса между качеством и эффективностью хранения данных.

2.3 Дискретное косинус-преобразование

После преобразования изображения из цветового пространства RGB в YCbCr и выполнения хрома-субдискретизации (уменьшения разрешения цветовых компонент Cb и Cr), выполняется разбиение изображения на блоки фиксированного размера 8×8 пикселей. Однако размеры исходного изображения не всегда кратны восьми, что делает невозможным непосредственное и равномерное разбиение без остатка. В таких случаях производится предварительная корректировка размеров изображения: вычисляется необходимое количество дополнительных строк и столбцов, которые должны быть добавлены к правому и нижнему краю изображения для достижения кратности 8.

Добавление недостающих строк и столбцов осуществляется с помощью повторения граничных значений (пикселей) изображения, чтобы минимизировать искажения, возникающие на краях.

Таблица 3. Расширенный блок 8×8 с добавленными пикселями

Этот шаг реализуется при помощи библиотеки OpenCV, предоставляющей функцию для симметричного дополнения изображений. После этого скорректированное изображение разбивается на непересекающиеся блоки размером 8×8.

Рисунок 2. Блоки 8x8

Для удобства хранения и последующей обработки эти блоки сохраняются в виде многомерных массивов с использованием библиотеки NumPy. Это позволяет эффективно обращаться к отдельным блокам, выполнять над ними математические преобразования и организовать дальнейший процесс.

Это важный шаг, позволяющий значительно повысить эффективность сжатия. Разделение на небольшие блоки позволяет применять алгоритмы сжатия локально, что упрощает обработку и улучшает степень компрессии.

Дальнейшая задача заключается в оценке уровня детализации внутри каждого блока. Если блок содержит мало визуальных деталей, его можно закодировать с меньшим числом битов, минимизируя потери качества. Для анализа степени детализации и выделения значимых компонент используется дискретное косинусное преобразование (ДКП, англ. Discrete Cosine Transform, DCT). Преобразование позволяет перейти от пространственного представления блока к частотному, выявляя частотные составляющие, которые имеют наибольшее значение для визуального восприятия.

Для понимания принципов работы ДКП целесообразно сначала рассмотреть его одномерную (векторную) форму, поскольку она обладает большей наглядностью и сохраняет все ключевые идеи, лежащие в основе двумерного варианта, применяемого в обработке изображений.

На рисунке 3 показано восемь волн косинуса, $\omega(f) = \cos(f\Theta)$, при $0 \leq \Theta \leq \pi$ с частотами $f = 0, 1, \ldots, 7$.

Рисунок 3. Графики функций $\omega(f) = \cos(f\Theta)$ для различных частот f.

На каждом графике отмечено восемь значений функции $\omega(f)$ с абсциссами

$$\Theta = \frac{\pi}{16}, \frac{3\pi}{16}, \frac{5\pi}{16}, \frac{7\pi}{16}, \frac{9\pi}{16}, \frac{11\pi}{16}, \frac{13\pi}{16}, \frac{15\pi}{16} \tag{4}$$

которые формируют базисный вектор v_f. В результате получится восемь векторов $v_f, f = 0, 1, \ldots, 7$ (всего 64 числа).

Таблица 4. Таблица значений $\cos(k\theta)$ для разных k и θ

θ	0.196	0.589	0.982	1.374	1.767	2.160	2.553	2.945
$\cos 0\theta$	1.	1.	1.	1.	1.	1.	1.	1.
$\cos 1\theta$	0.981	0.831	0.556	0.195	-0.195	-0.556	-0.831	-0.981
$\cos 2\theta$	0.924	0.383	-0.383	-0.924	-0.924	-0.383	0.383	0.924
$\cos 3\theta$	0.831	-0.195	-0.981	-0.556	0.556	0.981	0.195	-0.831
$\cos 4\theta$	0.707	-0.707	-0.707	0.707	0.707	-0.707	-0.707	0.707
$\cos 5\theta$	0.556	-0.981	0.195	0.831	-0.831	-0.195	0.981	-0.556
$\cos 6\theta$	0.383	-0.924	0.924	-0.383	-0.383	0.924	-0.924	0.383
$\cos 7\theta$	0.195	-0.556	0.831	-0.981	0.981	-0.831	0.556	-0.195

Они служат базисом одномерного косинусного преобразования, причём частота смены знаков увеличивается по строкам.

Можно показать, что все векторы v_i ортогональны между собой (из-за специального выбора восьми точек отсчета Θ). То же самое можно обнаружить прямым вычислением с помощью подходящей математической программы. Значит, эти восемь векторов можно поместить в матрицу размером 8 на 8 и рассмотреть соответствующее ей ортогональное преобразование - вращение в восьмимерном пространстве, которое называется одномерным дискретным косинус-преобразованием (DCT).

Одномерное преобразование косинусов Дискретное (DCT) можно интерпретировать как представление вектора в базисе, состоящем из векторов v_i В этом контексте любой вектор p из соответствующего векторного пространства может быть выражен через линейную комбинацию этих базисных векторов v_j.

Например, выберем 8 (коррелированных) чисел $p = (0.6, 0.5, 0.4, 0.5, 0.6, 0.5, 0.4, 0.55)$ в качестве тестовых данных. Выразим вектор p в виде суммы $p = \sum_i \omega_i v_i$ восьми веторов v_i. Решив эту систему из 8 линейных уравнений, находим восемь весов

$$\begin{aligned} &\omega_0 = 0.506, \omega_1 = 0.0143, \omega_2 = 0.0115, \omega_3 = 0.0439, \\ &\omega_4 = 0.0795, \omega_5 = -0.0432, \omega_6 = 0.00478, \omega_7 = -0.0077. \end{aligned} \tag{5}$$

Вес ω_0 не сильно отличается от элементов вектора p , но остальные семь весов гораздо меньше. Это показывает, как DCT (или любое другое ортогональное преобразование) производит сжатие. Теперь можно просто записать эти восемь весов в сжатый файл, где онибудут занимать меньше места, чем восемь компонентов исходного вектора p.

Рисунок 4. Графическое представление одномерного DCT.

Все восемь векторов v_i показаны в виде ряда из восьми маленьких серых квадратиков, причем значение $+1$ представлено белым цветом, а значение -1 окрашено в ченый цвет. Каждый из восьми компонентов вектора p выражен в виде взвешенной суммы восьми серых оттенков.

На практике одномерное DCT проще всего вычислять по формуле

$$G_f = \frac{1}{2} C_f \sum_{t=0}^{7} p_t \cos \frac{(2t+1)f\pi}{16} \tag{6}$$

$$\text{где} \quad C_f = \begin{cases} \frac{1}{\sqrt{2}}, & f = 0, \\ 1, & f > 0. \end{cases} \quad \text{При} \quad f = 0, 1, \ldots, 7.$$

Здесь исходными данными (пикселами, фрагментами звука или другими элементами) являются величины p_t, а им соответствующими коэффициентами DCT служат числа G_f. Формула (6) очень проста, но процесс вычисления по ней медленный. Декодер получает на входе коэффициенты DCT, делит их на восьмерки и применяет к ним обратное преобразование DCT (inverse DCT, IDCT) для восстановления исходных данных (тоже в виде групп по 8 элементов). Простейшая формула для вычисления IDCT имеет вид

$$p_t = \frac{1}{2} \sum_{j=0}^{7} C_j G_j \cos \frac{(2t+1)j\pi}{16}, \text{при } t = 0, 1, \ldots, 7 \tag{7}$$

2.4 Двумерное (матричное) DCT

Из опыта хорошо известно, что пикселы изображения имеют корреляцию не только по горизонтали, но и по вертикали. То есть пиксели взаимосвязаны как с соседними пикселами слева и справа, так и с пикселами сверху и снизу. Поэтому методы сжатия изображений используют двумерное DCT, которое задается формулой

$$G_{ij} = \frac{1}{\sqrt{2n}} C_i C_j \sum_{x=0}^{n-1} \sum_{y=0}^{n-1} p_{xy} \cos \frac{(2y+1)j\pi}{2n} \cos \frac{(2x+1)i\pi}{2n} \tag{8}$$

Для каждого коэффициента G_{ij} необходимо выполнить две вложенные суммы по n элементам, что даёт вычислительную сложность $O(n^2)$ на один коэффициент. Поскольку всего коэффициентов n^2, общая сложность для наивного вычисления двумерного DCT равна:

$$O(n^2) \times O(n^2) = O(n^4) \tag{9}$$

Использование алгоритмов быстрого дискретного косинус-преобразования (Fast DCT), аналогично быстрому преобразованию Фурье (FFT), позволяет дополнительно снизить вычислительную сложность одномерного DCT с $O(n^2)$ до $O(n \log n)$. В таком случае двухмерное

DCT реализуется как две последовательные одномерные операции, и общая сложность становится:

$$O(n \log n) \tag{10}$$

при $0 \leq i, j \leq n-1$. Изображение разбивается на блоки пикселов p_{xy} размера $n \times n$ (в данном примере $n = 8$), и уравнение (8) используется для нахождения коэффициентов G_{ij} для каждого блока пикселов. Если частичная потеря информации допустима, то коэффициенты подвергаются квантованию, что будет подробно рассмотрено в следующем разделе. Декодер восстанавливает сжатый блок данных (точно или приближенно), вычисляя обратное DCT (IDCT) по формуле

$$p_{xy} = \frac{1}{4} \sum_{i=0}^{7} \sum_{j=0}^{7} C_i C_j G_{ij} \cos \frac{(2y+1)j\pi}{16} \cos \frac{(2x+1)i\pi}{16} \tag{11}$$

$$\text{где} \quad C_f = \begin{cases} \frac{1}{\sqrt{2}}, & f = 0, \\ 1, & f > 0. \end{cases}$$

Двумерное дискретное косинусное преобразование (DCT) можно интерпретировать двумя способами: как композицию двух вращений, а также через представление вектора в базисе n-мерного векторного пространства. В первой интерпретации используется блок $n \times n$ пикселов (рис. (5)a, где элементы обозначены буквой "L")

L	L	L	L	L	L	L	L
L	L	L	L	L	L	L	L
L	L	L	L	L	L	L	L
L	L	L	L	L	L	L	L
L	L	L	L	L	L	L	L
L	L	L	L	L	L	L	L
L	L	L	L	L	L	L	L
L	L	L	L	L	L	L	L

(a)

L	S	S	S	S	S	S	S
L	S	S	S	S	S	S	S
L	S	S	S	S	S	S	S
L	S	S	S	S	S	S	S
L	S	S	S	S	S	S	S
L	S	S	S	S	S	S	S
L	S	S	S	S	S	S	S
L	S	S	S	S	S	S	S

(b)

L	S	S	S	S	S	S	S
S	s	s	s	s	s	s	s
S	s	s	s	s	s	s	s
S	s	s	s	s	s	s	s
S	s	s	s	s	s	s	s
S	s	s	s	s	s	s	s
S	s	s	s	s	s	s	s
S	s	s	s	s	s	s	s

(c)

Рисунок 5. Двумерное DCT и двойное вращение.

Сначала рассматриваются строки этого блока как точки $(p_{x,0}, p_{x,1}, \ldots, p_{x,n-1})$ в n- мерном пространстве, которые поворачиваются в этом пространстве с помощью преобразования, задаваемого внутренней суммой

$$G1_{x,j} = C_j \sum_{y=0}^{n-1} p_{xy} \cos \frac{(2y+1)j\pi}{2n}$$

из уравнения (8). Результатом этого вращения служит блок $G1_{x,j}$ из $n \times n$ коэффициентов, в котором в строках доминируют первые элементы (обозначенные как «L» на рис. (5)b), в то время как все остальные элементы малы (они обозначены как «S»). Внешняя сумма,

приведенная в уравнении (8), равна

$$G_{ij} = \frac{1}{\sqrt{2n}} C_i \sum_{x=0}^{n-1} p_{xy} G1_{x,j} \cos \frac{(2x+1)i\pi}{2n}$$

Здесь уже столбцы матрицы $1_{x,j}$ рассматриваются в качестве точек n - мерного векторного пространства, над которыми совершается преобразование вращения. В результате получается один большой коэффициент в верхнем левом углу блока (на рис.(5)c - это «L») и $N^2 - 1$ маленьких коэффициентов в остальных местах («S» и «s» на рисунке). Эта интерпретация рассматривает двумерное DTC в виде двух разных вращений размерности n.

Вторая интерпретация (при $n = 8$) использует уравнение (8) для создания 64 блоков по 8×8 величин в каждом. Все 64 блока рассматриваются в качестве базиса 64-мерного векторного пространства (это базисные изображения). Любой блок B из 8×8 пикселов можно выразить как линейную комбинацию этих базисных изображений, и все 64 веса этой линейной комбинации образуют коэффициенты DCT блока B.

Рисунок 6. 64 базисных изображения двумерного DCT.

На рисунке выше представлено графическое отображение 64 базисных функций двумерного дискретного косинусного преобразования (DCT) при размере блока $n = 8$. Каждый элемент с координатами (i, j) на данном рисунке соответствует блоку размером

8×8, полученному в результате вычисления выражения $\cos(i \cdot s) \cdot \cos(j \cdot t)$, где переменные s и t изменяются независимо в пределах, определённых уравнением (4).

Подведем итог этого, довольно сложного шага. Сжатие любого изображения с помощью DCT можно теперь сделать следующим образом.

1) Разделить изображение на k блоков пикселов размером $n \times n$ (чаще всего используется размер 8×8).
2) Применить дискретное косинусное преобразование (DCT) к каждому блоку $B^{(i)}$, представив его в виде линейной комбинации 64 базисных функций, показанных на рисунке выше. В результате получится набор блоков (далее будем называть их векторами) $W^{(i)}$, каждый из которых содержит 64 коэффициента $\omega_j^{(i)}$, где $j = 0, 1, \ldots, 63$.
3) Векторы $W^{(i)}$, где $i = 1, 2, \ldots, k$, группируются по компонентам, формируя 64 отдельных вектора коэффициентов

$$\mathbf{C_j} = \{\omega_j^{(1)}, \omega_j^{(2)}, \ldots, \omega_j^{(k)}\}.$$

   В частности, вектор $\mathbf{C_0}$ состоит из k коэффициентов DC.
4) Каждый вектор коэффициентов $C^{(j)}$ подвергается квантованию независимо от остальных. Полученные квантованные векторы $Q^{(j)}$ далее могут быть дополнительно сжаты с использованием методов, таких как RLE, код Хаффмана или других алгоритмов, и записаны в сжатый файл.

2.5 Квантование

В процессе сжатия изображений важным этапом является **квантование**, под которым понимается преобразование чисел с высокой точностью (чаще всего вещественных) в значения с меньшей точностью, например — округление до ближайшего целого или замена на приближённое значение из ограниченного набора.

Это позволяет существенно уменьшить объём передаваемой или сохраняемой информации за счёт исключения наименее значимых деталей, воспринимаемых человеком слабо либо вовсе незаметных.

Существует два основных подхода к квантованию — **скалярное** и **векторное**.

Скалярное квантование представляет собой поэлементную обработку данных, когда каждое значение преобразуется независимо от других. Этот метод прост в реализации и интуитивно понятен, но не всегда обеспечивает оптимальный результат: полезная информация может быть утеряна даже в значимых фрагментах, если не учитывать взаимосвязь между соседними значениями.

Для повышения эффективности используется **векторное** квантование, где обрабатываются не отдельные числа, а целые блоки данных — группы пикселей, которые рассмат-

риваются как многомерные векторы. Перед началом сжатия формируется специальный набор типовых блоков — кодовая книга. Каждый блок изображения сравнивается со всеми элементами этой книги, и выбирается наиболее близкий (в смысле минимальной разности значений). Вместо самого блока в выходной поток данных записывается лишь индекс (или указатель) соответствующего эталонного блока из кодовой книги.

Поскольку индекс, как правило, занимает меньше места, чем полный блок, такой подход позволяет добиться существенного уменьшения размера изображения. При этом достигается разумный баланс между степенью сжатия и качеством восстановления, особенно если кодовая книга заранее адаптирована под характеристики изображений, с которыми предполагается работать.

В стандарте JPEG реализовано канально-зависимое скалярное квантование, при котором каждый коэффициент, полученный в результате дискретного косинусного преобразования (DCT), квантуется независимо от остальных. Для каждого из цветовых компонентов — яркостного (Y) и двух хроминансных (Cb и Cr) — применяются отдельные квантовочные матрицы, отражающие различную чувствительность человеческого зрения к яркости и цвету. Производится простая операция округления или деления на фиксированный квантовочный коэффициент. Временная сложность такого подхода $O(n^2)$. Эти матрицы сформированы на основе психовизуальных моделей и оптимизированы так, чтобы минимизировать визуально заметные искажения при сжатии.

Матрица квантования яркости (Y):

$$\begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

Матрица квантования хроматических компонент (Cb и Cr):

$$\begin{bmatrix} 17 & 18 & 24 & 47 & 99 & 99 & 99 & 99 \\ 18 & 21 & 26 & 66 & 99 & 99 & 99 & 99 \\ 24 & 26 & 56 & 99 & 99 & 99 & 99 & 99 \\ 47 & 66 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \end{bmatrix}$$

Избыточное квантование может привести к появлению блочных артефактов, особенно при низком битрейте. Эти артефакты возникают из-за чрезмерной потери информации в результатах квантования и часто становятся заметными на границах блоков.

Когда коэффициенты в матрице квантования слишком велики (например, при слишком сильном сжатии), значительная часть информации теряется, особенно в высокочастотных компонентах, которые отвечают за детали изображения. Этот процесс квантования приводит к значительным потерям точности в преобразованных данных. На изображении это выражается в виде артефактов, которые часто становятся видимыми в местах стыка блоков.

Однако, такого эффекта можно добитья и при слишком малом квантовании. На границах блоков, где алгоритм не может точно передать информацию, начинается видимый переход между различными уровнями яркости или цвета, что приводит к образованию четких линий или квадратных паттернов, которые визуально ощущаются как артефакты.

Так же, поскольку изображение разбивается на блоки размером 8×8 пикселей, и каждый блок сжимается независимо. Это может привести к потере контекста между блоками, особенно в случае сильного сжатия. Границы блоков не могут быть плавными, из-за чего возникают резкие переходы между блоками.

Эти артефакты могут выглядеть как "ступеньки"или резкие изменения яркости и цвета, которые не присутствуют в исходном изображении.

Как уменьшить блочные артефакты

Существует несколько подходов для минимизации блочных артефактов при сжатии JPEG:

1) Использование более низкого коэффициента сжатия: Уменьшение уровня сжатия помогает сохранить больше данных, что уменьшает квантование и, соответственно, количество потерь информации. Это снижает вероятность появления артефактов, но увеличивает размер файла.
2) Адаптивное квантование: В некоторых случаях используется адаптивное квантование,

при котором для разных частей изображения выбираются различные матрицы квантования в зависимости от содержания. Например, для областей с высокой детализацией можно использовать более точное квантование, а для равномерных областей — более грубое.

3) Эксперименты с постобработкой: В некоторых случаях для устранения артефактов применяются методы постобработки, такие как сглаживание или фильтрация на уровне блоков.
4) Использование более сложных алгоритмов сжатия: Алгоритмы сжатия, такие как JPEG2000, используют более сложные методы, которые могут уменьшить или полностью устранить блочные артефакты. В частности, JPEG2000 использует вейвлет-преобразования вместо DCT, что позволяет избежать проблемы разделения изображения на блоки и улучшить качество сжатия.

JPEG не использует векторное квантование в силу его значительно большей вычислительной и алгоритмической сложности: построение и оптимизация кодовой книги требуют значительных ресурсов, а сам процесс кодирования с подбором ближайшего эталона становится существенно менее эффективным для реализации в массовых форматах хранения изображений. Кроме того, векторное квантование хуже адаптируется к переменной структуре изображений и менее совместимо с блочной природой DCT-представления.

2.6 Зигзаг-преобразование блоков

После применения DCT, и последующего квантования блоков, изображение представляется в виде матрицы коэффициентов 8×8, где каждый коэффициент представляет собой степень различной частоты. Эти коэффициенты могут включать как высокочастотные (мелкие детали изображения), так и низкочастотные компоненты (общие очертания). Важным моментом является то, что после квантования высокочастотные коэффициенты, как правило, имеют значения, близкие к нулю, и не играют значительной роли для восприятия изображения. Поэтому их можно исключить или закодировать с меньшей точностью.

Зигзаг-преобразование используется для того, чтобы упорядочить эти коэффициенты в порядке, который позволяет более эффективно их кодировать. Суть метода в том, что он позволяет собирать все нулевые значения (или значения, близкие к нулю) в конец последовательности, что снижает объем данных при их дальнейшем кодировании.

Зигзаг-преобразование упорядочивает эти коэффициенты таким образом, что сначала идут наиболее значимые коэффициенты (низкочастотные), а затем — менее значимые (высокочастотные, которые в основном равны нулю).

Зигзагообразный порядок включает следующие шаги:

1) Начинается с первого элемента в верхнем левом углу матрицы (нулевой индекс).

2) Затем происходит движение по диагоналям матрицы, начиная с верхней левой части, последовательно переходя к правому нижнему углу.
3) Каждая диагональ матрицы представляет собой группу коэффициентов, которые последовательно вытягиваются в одном ряду.

Рисунок 7. Обход блока зиг-заг.

В результате применения зигзагообразного обхода к блоку преобразованных коэффициентов дискретного косинусного преобразования формируется линейная последовательность. Один из примеров такой последовательности приведён ниже.

$$\text{Последовательность после зигзагообразного обхода: } \left\{ \begin{matrix} 52 & -3 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{matrix} \right\} \quad (12)$$

Здесь первый элемент (52) соответствует DC-компоненте, а последующие значения — коэффициенты высоких частот (AC-компоненты), отсортированные по убыванию их вклада в общее изображение.

2.7 RLE-кодирование

После применения зигзаг-преобразования каждый блок коэффициентов преобразуется в одномерный массив, в котором значимая информация (низкочастотные коэффициенты) сосредоточена в начале, а большая часть оставшихся элементов принимает нулевое значение. Это делает полученную последовательность особенно удобной для дальнейшего сжатия с использованием методов, предназначенных для работы с повторяющимися элементами.

На этом этапе в классическом алгоритме JPEG традиционно применяется энтропийное кодирование, наиболее часто — с использованием кода Хаффмана, позволяющего достичь высокой степени сжатия за счёт построения оптимального префиксного кодового дерева, основанного на вероятностях появления различных значений. Однако реализация эффективного кодировщика Хаффмана требует значительных трудозатрат, включая построение статистической модели, генерацию кодовых деревьев и реализацию соответствующего декодирования.

В рамках данной работы энтропийное кодирование было намеренно опущено. Вместо него используется более простой и наглядный метод кодирования с повторениями — Run-Length Encoding (RLE), который позволяет проиллюстрировать фундаментальные принципы постобработки данных после зигзаг-преобразования и обеспечивает компрессию, достаточную для демонстрации эффективности базового JPEG-подобного сжатия. Это решение продиктовано необходимостью сосредоточиться на ключевых этапах сжатия и ограниченностью временных ресурсов, отведённых на реализацию дипломного проекта.

Применение RLE особенно эффективно в контексте JPEG-подобного представления данных, так как вследствие квантования и зигзагообразного упорядочивания большинство коэффициентов в конце последовательности равны нулю. Алгоритм RLE преобразует такие повторы в компактную форму, заменяя длинные последовательности одинаковых значений (в первую очередь нулей) парой «значение–длина повтора», что существенно снижает объём итоговых данных. Временная сложность алгоритма $O(N)$, где N — длина последовательности (в данном случае 64).

Таким образом, хотя в полномасштабной реализации алгоритма JPEG предпочтительно использовать энтропийное кодирование, в рамках данной работы RLE выступает как более простая альтернатива, позволяющая сохранить общий принцип компрессии без существенного увеличения сложности реализации.

Таблица 5. Последовательность коэффициентов после зигзаг-преобразования

№	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
Значение	52	-3	0	0	0	2	0	0	0	0	0	0	0	0	1	0

Таблица 6. Преобразование последовательности с помощью RLE

№	Пара (значение, количество нулей перед)	Пояснение
1	(52, 0)	Первое значение, сразу записывается
2	(-3, 0)	Следующее ненулевое значение без нулей перед ним
3	(0, 3)	Три последовательных нуля
4	(2, 0)	Следующий ненулевой коэффициент
5	(0, 7)	Семь нулей подряд
6	(1, 0)	Следующий ненулевой коэффициент
7	(0, 1)	Завершающий ноль

Применение RLE позволяет существенно сократить объём информации, подлежащей сохранению или передаче, при этом сохраняя возможность точного восстановления исходной последовательности. Хотя степень сжатия, достигаемая с помощью RLE, уступает алгоритмам энтропийного кодирования (таким как Хаффман), его простота и наглядность делают его отличным выбором для учебных и демонстрационных целей.

Таким образом, использование RLE в данной работе позволило реализовать полный цикл JPEG-подобного сжатия — от разделения изображения на блоки и применения косинус-преобразования до финального этапа кодирования, — при этом сосредоточившись на ключевых принципах без чрезмерного усложнения алгоритма.

2.8 Декодирование

Процесс декодирования JPEG является обратной операцией по отношению к этапам кодирования и направлен на восстановление изображения из сжатого представления. На выходе должен быть получен массив пикселей, максимально приближенный к исходному изображению. Процесс декодирования включает следующие этапы:

Распаковка закодированных данных

На первом этапе декодирования выполняется извлечение и интерпретация сжатых данных, полученных после применения метода RLE (run-length encoding, кодирование длин серий). В процессе кодирования многие коэффициенты DCT, особенно высокочастотные, равны нулю. Это свойство используется для повышения степени сжатия — нули заменяются на пары вида (количество нулей, ненулевое значение).

При декодировании производится восстановление полного набора коэффициентов блока размером 8×8 из такой сжатой последовательности. Например, последовательность:

$$
\begin{aligned}
&\text{Сжатая последовательность пар (количество нулей, значение):}\\
&\quad (0,\ 52),\ (0,\ -3),\ (3,\ 2),\ (9,\ 1)\\
&\text{После распаковки:}\\
&\quad 52,\ -3,\ 0,\ 0,\ 0,\ 2,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 0
\end{aligned}
\tag{13}
$$

Обратный зигзагообразный обход

Поскольку в процессе кодирования двумерный массив коэффициентов представлялся в виде одномерного массива с помощью зигзагообразного обхода, теперь необходимо восстановить исходное расположение коэффициентов в блоке 8×8. Каждому элементу последовательности возвращается его позиция в соответствии со стандартной зигзагообразной схемой.

Обратное квантование

На этапе обратного квантования каждый коэффициент умножается на соответствующее значение из таблицы квантования, использованной при кодировании. Если обозначить квантованный коэффициент как $Q_{i,j}$, а таблицу квантования как $T_{i,j}$, то восстановленный DCT-коэффициент можно получить по формуле:

$$D_{i,j} = Q_{i,j} \circ T_{i,j} \tag{14}$$

Это приближает коэффициенты к их исходным значениям до потери точности при квантовании.

Обратное дискретное косинусное преобразование (IDCT)

На этом этапе выполняется обратное дискретное косинусное преобразование (IDCT) для каждого блока. Восстановление значений пикселей в пространственной области осуществляется на основе 64 DCT-коэффициентов, по формуле (11).

Восстановление блоков

После применения IDCT к каждому блоку получаются блоки из 64 значений интенсивности, которые собираются в итоговое изображение. При этом важно точно соблюсти позиционирование блоков и границы изображения.

После этого идет обратное преобразование цветного пространства из **YCbCr** в **RGB**, что позволяет получить восстановленное изображение в привычном для восприятия формате.

3 Руководство пользователя

3.1 Назначение программы

Разработанная программа реализует алгоритм сжатия изображений, повторяющий основные этапы стандарта JPEG. В отличие от оригинального алгоритма, на этапе энтропийного кодирования используется метод RLE (Run-Length Encoding) вместо кодирования по Хаффману. Это позволяет упростить реализацию, сохранив при этом все основные свойства алгоритма JPEG.

Программа предназначена для демонстрации полного цикла кодирования и декодирования изображения:

- переход от **RGB** к **YCbCr**
- разбиение на блоки 8×8
- дискретное косинус-преобразование (DCT)
- квантование
- последовательное сканирование (зиг-заг)
- кодирование RLE
- обратные преобразования (IDCT и т. д.)
- восстановление RGB-изображения

3.2 Интерфейс программы

На рисунке 8 представлено главное окно приложения.

Рисунок 8. Главное окно приложения.

Основные элементы интерфейса:

- **Открыть изображение**
  Позволяет выбрать изображение в формате PNG или JPEG для обработки.

Рисунок 9. Окно выбора изображений.

- **Выбрать реализацию DCT**

  В графическом интерфейсе приложения предусмотрена возможность выбора реализации дискретного косинусного преобразования (DCT). Пользователю предлагаются два варианта:

  Собственная реализация DCT, разработанная вручную в рамках дипломного проекта;

  Готовая реализация из библиотеки SciPy (scipy.fftpack.dct), используемая в качестве эталонной и производительной альтернативы.

  Это позволяет провести наглядное сравнение корректности и эффективности обеих реализаций, а также удостовериться в соответствии собственной реализации общепринятым стандартам.

Рисунок 10. Выбрать реализацию DCT.

- **Задать уровень сжатия**

  Предусмотрена возможность установки уровня сжатия, который напрямую влияет на коэффициенты квантования. Более высокий уровень сжатия приводит к агрессивному округлению высокочастотных коэффициентов, что уменьшает объём выходного файла, но может снизить качество восстановленного изображения. Напротив, меньшие значения сохраняют больше информации, обеспечивая лучшее качество при увеличенном размере файла.

  Это позволяет пользователю находить баланс между качеством изображения и степенью компрессии в зависимости от задач.

Рисунок 11. Выбрать уровень сжатия.

- **Обработать изображение**

  После выбора параметров пользователь может запустить процесс обработки, нажав на кнопку **«Обработать изображение»**. Программа выполняет полный цикл сжатия и восстановления изображения, повторяя основные этапы алгоритма JPEG.

  Сначала изображение преобразуется из цветового пространства RGB в YCbCr. Затем изображение разбивается на блоки. К каждому блоку применяется дискретное косинусное преобразование (DCT), переводя данные из пространственной области в частотную. Полученные коэффициенты проходят этап квантования, при котором незначительные высокочастотные компоненты подавляются — это и есть основное сжатие с потерями.

  После квантования коэффициенты сканируются по зигзагообразной траектории, что группирует нули, появившиеся в результате округлений. Затем выполняется кодирование методом RLE.

  На финальном этапе происходит декодирование, и вывод получившегося изображения в в окне приложения.

  После завершения обработки становятся доступными дополнительные элементы интерфейса, позволяющие более подробно проанализировать результат.

Рисунок 12. Дополнительные элементы.

- **Сохранить изображение**

  Позволяет сохранить на диске восстановленное после сжатия изображение в формате JPEG.

Рисунок 13. Сохранить изображение.

- **Показать карту энергии DCT-коэффициентов**
  Данный элемент визуализирует вклад различных частотных компонент в изображение до этапа квантования. Это позволяет понять, какие частоты наиболее значимы в передаче визуальной информации

Рисунок 14. Карта энергии DCT-коэффициентов.

- **Показать карту энергии квантованных блоков**
  Отображает распределение энергии после квантования. Данная визуализация помогает оценить потери, вызванные округлением коэффициентов.

Рисунок 15. Карта энергии квантованных блоков.

В нижней части окна приложения отображаются числовые параметры, предоставляющие пользователю информацию о производительности и эффективности алгоритма сжатия:

- **Размер изображений** — отображает объём изображения до и после обработки, в килобайтах. Этот позволяет пользователю сравнить исходный размер файла с результатом сжатия и оценить, насколько сильно уменьшился объём данных.
- **Время обработки** — показывает общее время, затраченное на выполнение всех этапов алгоритма, включая преобразование цветового пространства, разбиение на блоки, дискретное косинусное преобразование, квантование, кодирование с помощью RLE и последующее восстановление изображения. Этот показатель даёт представление о производительности реализации и позволяет сравнивать скорость работы разных вариантов DCT или уровней сжатия.

Рисунок 16. Числовые параметры.

4 Результаты и выводы

На рисунке 17 представлена зависимость метрики SSIM от уровня квантования.

SSIM (Structural Similarity Index) — это метрика, используемая для оценки качества изображения, которая измеряет, насколько два изображения похожи на уровне их структуры. Она была предложена в 2004 году как улучшение для традиционных методов оценки качества изображений, таких как PSNR (Peak Signal-to-Noise Ratio). SSIM учитывает восприятие человеческим глазом и включает в себя три важнейших аспекта изображения:

- Яркость (Luminance) — измеряет разницу в яркости между двумя изображениями.
- Контраст (Contrast) — оценивает, насколько изображения различаются по контрасту.
- Структура (Structure) — учитывает, насколько хорошо сохраняется структура или текстура изображения, например, линии, края и другие важные детали.

(значения ближе к 1 означают высокое качество).

Формула SSIM

Метрика рассчитывается по следующей формуле:

$$\mathrm{SSIM}(x, y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)} \tag{15}$$

- x, y — сравниваемые изображения (или блоки изображений).
- μ_x — среднее значение яркости изображения x:

$$\mu_x = \frac{1}{N}\sum_{i=1}^{N} x_i$$

- μ_y — среднее значение яркости изображения y.
- σ_x^2 — дисперсия яркости изображения x:

$$\sigma_x^2 = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu_x)^2$$

- σ_y^2 — дисперсия яркости изображения y.
- σ_{xy} — ковариация между x и y:

$$\sigma_{xy} = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu_x)(y_i - \mu_y)$$

- $C_1 = (K_1L)^2$, $C_2 = (K_2L)^2$ — стабилизирующие константы:
  - L — динамический диапазон значений пикселей (например, 255 для 8-битных изображений),
  - $K_1 = 0.01$, $K_2 = 0.03$.

Рисунок 17. Зависимость метрики SSIM от уровня квантования.

График показывает, что при увеличении уровня квантования (то есть более агрессивном удалении высокочастотной информации) метрика SSIM уменьшается. Это соответствует ожиданиям: чем сильнее сжатие, тем больше потерь информации и ниже визуальное качество. На небольших уровнях квантования (до, 70%) качество остаётся почти неизменным, но при уровнях выше 70% наблюдается значительное ухудшение. Это даёт возможность выбрать компромисс между степенью сжатия и визуальным качеством.

На рисунке 18 показано, как изменяется метрика PSNR (Peak Signal-to-Noise Ratio) в зависимости от размера блока, используемого для дискретного косинусного преобразования.

Рисунок 18. Зависимость PSNR от размера блока.

PSNR — это классическая метрика оценки искажения сигнала (в данном случае, изображения); более высокие значения указывают на лучшее качество восстановления. При небольшом размере блока (8×8) значения PSNR максимальны, что объясняется более точной локальной аппроксимацией содержимого. С увеличением размера блока (например, 32×32) точность снижается из-за усреднения на более крупных участках изображения, что приводит к потере деталей и снижению PSNR.

$$\text{PSNR} = 10 \cdot \log_{10}\left(\frac{MAX_I^2}{\text{MSE}}\right) \tag{16}$$

- MAX_I — максимальное значение интенсивности пикселя (обычно 255).
- MSE — среднеквадратичная ошибка между оригинальным и восстановленным изображениями:

$$\text{MSE} = \frac{1}{MN}\sum_{i=1}^{M}\sum_{j=1}^{N}\left[I(i,j) - K(i,j)\right]^2$$

- $I(i,j)$ — значение пикселя в оригинальном изображении.
- $K(i,j)$ — значение пикселя в искажённом изображении.

Таким образом, блоки 8×8 являются оптимальными с точки зрения компромисса меж-

ду сжатием и качеством, что подтверждает выбор JPEG-стандарта.

Рисунок 19 демонстрирует, как изменяется метрика SSIM в зависимости от размера блока. Аналогично PSNR, максимальное качество достигается при использовании размера 4×4, однако для сохранения скорости обработки компромиссом является размер 8×8. По мере увеличения блока падает точность локального анализа изображения, что особенно критично для SSIM — метрики, учитывающей локальные структуры и контраст.

Рисунок 19. Зависимость SSIM от размера блока.

Таким образом, большие блоки ухудшают восприятие изображения, особенно в областях с высокой детализацией. Это подчёркивает важность выбора разумного размера блока для сохранения визуального качества при сжатии.

На рисунке 20 представлена зависимость времени обработки изображения от размера блока. Время измерялось в миллисекундах для полного цикла сжатия и восстановления изображения фиксированного размера (например, 512×512 пикселей).

Рисунок 20. Зависимость времени обработки от размера блока.

График показывает, что с увеличением размера блока общее количество блоков уменьшается, что снижает число вычислений (например, DCT и RLE выполняются реже), и, как следствие, сокращается общее время обработки. Однако при слишком больших блоках может наблюдаться незначительный рост времени из-за увеличения объёма обработки каждого отдельного блока и возможного падения эффективности кэширования.

Этот график полезен для анализа производительности и помогает выбрать подходящий размер блока в зависимости от требований к скорости и качеству.

Влияние квантования на цветовые искажения

В процессе экспериментов было замечено, что неправильно подобранные коэффициенты квантования особенно для цветовых компонент (Cb и Cr) могут вызывать искажения цветовой палитры изображения.

На ряде примеров наблюдалось:

- Уход цвета в сторону сине-зелёных или красных оттенков;
- Появление "грязных"артефактов в областях с однородным цветом;
- Потеря цветовой согласованности на границах объектов.

Рисунок 21. До квантования с неправильно подобранными коэффициентами.

Рисунок 22. После квантования с неправильно подобранными коэффициентами.

На рисунке 22 видно, как цветовая палитра сместилась в сторону зеленого цвета.

Это подчёркивает важность отдельного подбора квантования для яркостного и цветовых каналов, что в классическом JPEG также учитывается.

Проявление артефактов

Но степень квантования и неправильно подобрнанные коэффициенты для матриц квантования не единственные причины из-за которых могут появиться артефакты в области границ или неоднородного цвета.

При использовании блоков большого размера (например, 32×32), на изображениях начали проявляться артефакты в виде белых точек или резких границ между блоками.

Особенности:

- Артефакты особенно видны в текстурированных или теневых областях.
- В некоторых случаях они проявляются как “решётка” из блоков, так называемый "блокинг-эффект".
- В наиболее заметных случаях артефакты выглядят как одиночные белые пиксели, особенно на тёмных участках изображения.

Рисунок 23. Артефакты в виде белых пикселей.

Дополнительную нагрузку на алгоритм создаёт избыточная детализация — большое количество текстур, тонких линий и резких переходов в изображении ухудшают результаты аппроксимации в частотной области. Особенно чувствительными к искажениям оказываются области с высокими градиентами цвета, где даже при умеренном сжатии возможно появление визуальных дефектов. Таким образом, причины появления артефактов комплексны и зависят как от параметров алгоритма, так и от характера самого изображения.

Способы устранения визуалных дефектов

На основе экспериментов были выделены практические приёмы снижения артефактов:

1) **Снижение уровня квантования** Уменьшает потери, особенно в чувствительных областях изображения (тени, переходы, кожа и т.п.).

2) **Раздельное квантование Y и Cb/Cr** Позволяет сохранять цветовую палитру более точно, снижая искажения. Так же удобно, для анализа отдельных компонентов, упрощает посиск ошибок в приложении и его отладку.
3) **Уменьшение размера блока (например, до 4×4)** Повышает точность локального анализа, особенно в участках с деталями и границами. Однако может значительно увеличить время обработки изображения. В этом случае можно двигаться в сторону параллелизма и модификации дискретного косинусного преобразования (DCT) для ускорения.
4) **Подбор и адаптация матриц квантования** Более мягкие матрицы снижают потери на высоких частотах и уменьшают риск появления "белых точек".
5) **Пред/пост обработка изображения (например, фильтрация шума)** Уменьшение шумов до сжатия помогает избежать ненужных артефактов. После сжатия, так же можно применять фильтры для устранения уже имеющегося шума.
6) **Адаптация к содержимому** Возможность выбора параметров сжатия в зависимости от анализа изображения (например, в сложных участках использовать более мелкие блоки и меньшую степень квантования). Или разработать тест.

Сравнение производительности алгоритма DCT

Целью данного эксперимента является сравнение производительности собственной реализации алгоритма дискретного косинусного преобразования (DCT) с реализацией, предоставляемой сторонней библиотекой. Основной метрикой сравнения является время обработки изображений различного размера при использовании двух реализаций.

Проводилось сжатие изображений различного исходного размера (от 1 до 13МБ) при помощи двух реализаций алгоритма DCT: Библиотечная из scipy, пакета fftpack и собственная, разработанная с использованием библиотеки numpy.

Каждое изображение обрабатывалось с тремя различными коэффициентами сжатия Для каждого случая фиксировалось:

1) Время выполнения алгоритма;
2) Размер входного изображения до сжатия;

Все полученные данные сохранялись в таблицу формата CSV.

Таблица 7. Библиотечная реализация DCT)

Время обработки (сек)	Размер до (Кб)	Размер после (Кб)	Ширина	Высота
12.6747	1754.59	1581.75	3500	2385
22.3224	6860.50	980.25	5120	2880
3.6232	2288.86	281.88	2048	1152
5.5511	2745.00	421.69	1440	2560
13.9913	1745.58	1664.48	3840	2400
21.9425	3683.08	1055.16	5120	2880
51.2789	10064.11	8599.84	7680	4320
6.1473	999.12	547.01	2560	1600
24.3968	1258.02	1172.49	5120	3200
12.2057	1903.22	648.21	3840	2160
13.8569	3414.73	977.16	3840	2400
49.0091	6705.57	1603.21	7680	4320
22.0647	4566.30	1077.42	5120	2880
41.5755	9068.69	3607.46	6350	4285
12.3341	1599.10	620.09	3840	2160
21.8722	5138.39	919.58	5120	2880
12.1027	1187.25	632.94	3840	2160
11.9819	3318.69	650.59	3840	2160
11.8780	3386.16	883.60	3840	2160

Таблица 8. Собственная реализация DCT)

Время обработки (сек)	Размер до (Кб)	Размер после (Кб)	Ширина	Высота
6.7837	404.31	82.92	2218	1247
80.2578	6705.57	1603.48	7680	4320
35.4494	4566.30	1077.59	5120	2880
65.5053	9068.69	3607.05	6350	4285
19.8254	1599.10	620.08	3840	2160
34.7825	5138.39	919.78	5120	2880
19.5823	662.47	219.04	3840	2160
19.4002	1187.25	632.96	3840	2160
19.7136	3318.69	650.54	3840	2160
19.6139	3386.16	883.77	3840	2160
35.5350	6860.50	980.02	5120	2880
4.8934	404.31	82.90	2218	1247

Временные метки использовались как способ сегментации результатов по типу реализации.

На рисунке ниже приведён график зависимости времени обработки от исходного размера изображения для обеих реализаций:

Рисунок 24. Зависимость времени обработки от размера изображения для библиотечной и собственной реализаций DCT.

Анализ графика показывает следующее:

Для изображений размером до 3 МБ, обе реализации показывают приблизительно одинаковую скорость обработки.

При увеличении размера изображения от 4 МБ собственная реализация демонстрирует несколько большее время выполнения по сравнению с библиотечной.

Среднее время обработки собственной реализации составляет 30.7361 сек, что на 61% хуже по сравнению с реализацией из библиотеки SciPy.

Основной причиной некоторого отставания собственной реализации является то, что библиотечная реализация SciPy написана с использованием низкоуровневых оптимизаций на языке C и оптимизирована под конкретные архитектуры процессоров, включая SIMD-инструкции и многопоточность. В свою очередь, собственная реализация выполнена на более высокоуровневом языке (или без таких оптимизаций), что приводит к увеличению времени обработки.

Несмотря на это, разница во времени незначительна и не влияет критически на применимость алгоритма в практических задачах. Кроме того, собственная реализация может быть проще для модификации и адаптации под специфические нужды проекта.

5 Заключение

В ходе выполнения выпускной квалификационной работы была успешно реализована собственная версия алгоритма сжатия изображений, основанная на принципах стандарта JPEG. Разработка включала все основные этапы: преобразование цветового пространства, блочную обработку, дискретное косинусное преобразование (DCT), квантование и обратное преобразование. Вместо кодирования Хаффмана, применяемого в классическом JPEG, в данной реализации использовалась алгоритмически более простая схема RLE-сжатия, что упростило реализацию и позволило сосредоточиться на исследовании ключевых этапов преобразования изображения.

Был разработан удобный пользовательский интерфейс, позволяющий наглядно управлять некоторыми параметрами сжатия (уровень квантования, выбор реализации DCT) и анализировать результаты в реальном времени. Программа визуализирует восстановленное изображение, отображая значения метрик, такие как информацию о размере сжатого файла и времени обработки.

Проведены обширные экспериментальные исследования, в том числе:

- анализ влияния уровня квантования на качество восстановления изображения;
- оценка влияния размера блока на метрики качества и скорость обработки;
- сравнение результатов при различных параметрах на одном и том же наборе данных.
- ставнение реализаций DCT.

Эксперименты подтвердили, что параметры алгоритма оказывают значительное влияние на компромисс между степенью сжатия, качеством восстановления и производительностью. На основе полученных результатов были сделаны выводы, позволяющие адаптировать параметры алгоритма под различные практические задачи.

Разработанная реализация и программный интерфейс могут быть полезны как в рамках дальнейших исследований в области обработки изображений, так и в образовательных целях, благодаря наглядности работы ключевых этапов алгоритма JPEG.

Таким образом, работа соответствует теме «Реализация и исследование алгоритма JPEG для сжатия изображений», так как включает как теоретический анализ, так и практическое улучшение и адаптацию классического подхода, направленные на повышение гибкости и удобства применения алгоритма в современных условиях.

Литература

1. Сэломон, Д. Сжатие данных, изображений и звука / Д. Сэломон. – М.: Техносфера, 2006. – 368 с.

2. Томас, Кормен. Алгоритмы Построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К, Штайн - 2-е изд., - Москва: Вильямс, 2011. – 1296 с.

3. Официальная документация OpenCV. – Режим доступа: -URL: https://docs.opencv.org/4.x/index.html (дата обращения: 19.02.2025). – Текст: электронный.

4. Земцов, А. Н. Сравнительный анализ эффективности методов сжатия изображений на основе дискретного косинусного преобразования и фрактального кодирования (начало) / И. В. Петрова, Н. Г. Мамаев. – Волгоград: Издательство Волгоградского государственного технического университета, 2015. – 8 с.

5. Гарсия, Г. Обработка изображений с помощью OpenCV / Г. Гарсия, О. Суарес, Х. Аранда, Х. Терсеро, И. Грасиа, Н. Энано. – Москва: ДМК, 2016. – 210 с.

6. Официальная документация PyQt5. – Режим доступа: -URL: https://www.riverbankcomputing.com/ static/Docs/PyQt5/. (дата обращения: 05.01.2025). – Текст: электронный.

7. Официальная документация языка Python. – В режиме доступа: -URL: https://docs.python.org/3/ (дата обращения: 05.01.2025). – Текст: электронный.

6 Приложение

6.1 Класс подготовки изображения

Загружает цветное изображение по заданному пути и преобразует его в цветовое пространство YCrCb. Предоставляет методы для субдискретизации цветовых каналов Cb и Cr (например, 4:2:0), дополняет изображение до размеров, кратных блочному размеру, разбивает изображение на блоки заданного размера и собирает изображение обратно из таких блоков.

```

    import numpy as np

class ImagePreparation:
    def __init__(self,
                 image_path,
                 block_size=8
                 ):

        self.block_size = block_size
        print(f"Block_size: {self.block_size}")
        self.image_path = image_path
        self.image = cv2.imread(self.image_path, cv2.IMREAD_COLOR)

        self.h = self.image.shape[0]
        self.w = self.image.shape[1]

        if self.image is None:
            raise FileNotFoundError("Image not loaded!")
        else:
            self.img_ycrcb = cv2.cvtColor(self.image, cv2.COLOR_BGR2YCrCb)
            print("File uploaded successfully!")

    def apply_subsampling(self):
        """
        Apply subsampling to Cb and Cr channels (e.g., 4:2:0).
        """
        y, cr, cb = cv2.split(self.img_ycrcb)

        # Reduce resolution of Cr and Cb channels by half (subsampling
            4:2:0)
        cr_resized = cv2.resize(cr, (self.w // 2, self.h // 2),
            interpolation=cv2.INTER_LINEAR)
        cb_resized = cv2.resize(cb, (self.w // 2, self.h // 2),
            interpolation=cv2.INTER_LINEAR)

        # Resize Y channel if needed to match Cr and Cb sizes
```

```
        if y.shape[:2] != cr_resized.shape[:2]:
            y = cv2.resize(y, (cr_resized.shape[1], cr_resized.shape[0]),
                interpolation=cv2.INTER_LINEAR)

        # Ensure all channels have the same data type
        cr_resized = cr_resized.astype(np.uint8)
        cb_resized = cb_resized.astype(np.uint8)
        y = y.astype(np.uint8)

        # Merge channels back after resizing
        self.img_ycrcb = cv2.merge([y, cr_resized, cb_resized])
        self.h = self.img_ycrcb.shape[0]
        self.w = self.img_ycrcb.shape[1]
        print(f"Subsampled image shape: {self.img_ycrcb.shape}")

    def pad_image_to_multiple(self):
        """
        Pad the image so its dimensions are multiples of the block size.

        Returns:
            numpy.ndarray: Padded image.
        """
        pad_h = (self.block_size - self.h % self.block_size) % self.
            block_size
        pad_w = (self.block_size - self.w % self.block_size) % self.
            block_size

        self.img_ycrcb = cv2.copyMakeBorder(self.img_ycrcb, 0, pad_h, 0,
            pad_w, cv2.BORDER_REPLICATE)
        self.h = self.img_ycrcb.shape[0]
        self.w = self.img_ycrcb.shape[1]

        print(f"Padded image shape: {self.img_ycrcb.shape}")

        return self.img_ycrcb

    def split_into_blocks(self):
        """
        Split the padded image into blocks and store them in `self.blocks`.

        Returns:
            numpy.ndarray: Array of image blocks with shape
            (num_blocks_y, num_blocks_x, block_size, block_size, channels).
        """
        self.apply_subsampling()
        self.pad_image_to_multiple()
```

```python

        blocks = self.img_ycrcb.reshape(self.h // self.block_size, self.block_size,
                                        self.w // self.block_size, self.block_size,
                                        self.img_ycrcb.shape[2])
        self.blocks = blocks.swapaxes(1, 2)
        return self.blocks

    def merge_blocks(self, blocks):
        """
        Reconstruct image from blocks.

        Args:
            blocks (numpy.ndarray): Array of blocks (H, W, block_size, block_size, channels).

        Returns:
            numpy.ndarray: Reconstructed image cropped to original size.
        """
        H, W, block_size, _, channels = blocks.shape
        h, w, _ = self.image.shape

        img = np.zeros((H * block_size, W * block_size, channels), dtype=np.uint8)

        for i in range(H):
            for j in range(W):
                img[i * block_size:(i + 1) * block_size, j * block_size:(j + 1) * block_size, :] = blocks[i, j]

        return img[:h, :w]
```

Листинг 1. Класс подготовки изображения

6.2 Класс дикретного косинусного преобразования

Класс реализует двухмерное дискретное косинусное преобразование (DCT) и обратное преобразование (IDCT) для блоков изображений. В классе есть два варианта вычисления DCT: собственная реализация и использование библиотеки `scipy`. Также реализованы методы для применения DCT и IDCT к набору блоков с учётом цветовых каналов.

```python

from scipy.fftpack import dct
from scipy.fftpack import idct

```

```python
class DCT2D:
    def dct2Down(self, block):
        """
        Performs 2D discrete cosine transform (DCT) on a given square block manually.

        Args:
            block (numpy.ndarray): Input 2D array (block) of size (N, N).

        Returns:
            numpy.ndarray: Transformed block of the same size with applied DCT.
        """

        N = block.shape[0]
        dct_res = np.zeros((N, N), dtype=np.float32)
        alpha = lambda k : np.sqrt(1/N) if k == 0 else np.sqrt(2/N)

        for u in range(N):
            for v in range(N):
                sum_value = 0.0
                for x in range(N):
                    for y in range(N):
                        sum_value += block[x, y] * \
                                     np.cos((2 * x + 1) * u * np.pi / (2 * N)) * \
                                     np.cos((2 * y + 1) * v * np.pi / (2 * N))
                dct_res[u, v] = alpha(u) * alpha(v) * sum_value

        return dct_res


    def dct2D(self, block):
        """
        Performs 2D discrete cosine transform (DCT) using scipy library.

        Args:
            block (numpy.ndarray): Input 2D array (block) of size (N, N).

        Returns:
            numpy.ndarray: Transformed block of the same size with applied DCT.
        """
        dct_rows = dct(block, axis=0, norm='ortho')
        dct_2d = dct(dct_rows, axis=1, norm='ortho')
```

```python

        return dct_2d


    def apply_dct_to_blocks(self, blocks, use_own=False):
        """
        Applies 2D DCT to all image blocks. Can choose between own implementation and scipy.

        Args:
            blocks (numpy.ndarray): 5D array of image blocks with shape (num_blocks_x, num_blocks_y, block_size_x, block_size_y, channels).
            use_own (bool, optional): Whether to use own DCT implementation. Defaults to False.

        Returns:
            numpy.ndarray: 5D array with applied DCT to each block.
        """
        H, W, block_size, _, channels = blocks.shape
        self.dct_blocks = np.zeros_like(blocks, dtype=np.float32)

        for i in range(H):
            for j in range(W):
                for c in range(channels):
                    block = blocks[i, j, :, :, c]
                    block = np.float32(block)

                    if use_own:
                        self.dct_blocks[i, j, :, :, c] = self.dct2Down(block)
                    else:
                        self.dct_blocks[i, j, :, :, c] = self.dct2D(block)

        return self.dct_blocks


    def idct2D(self, block):
        """
        Performs inverse 2D discrete cosine transform (IDCT) using scipy.

        Args:
            block (numpy.ndarray): Input block of DCT coefficients of size (N, N).

```

```
        Returns:
            numpy.ndarray: Reconstructed pixel block.
        """
        temp = idct(block, axis=0, type=2, norm='ortho')
        idct_res = idct(temp, axis=1, type=2, norm='ortho')
        return idct_res

    def apply_idct_to_blocks(self, dct_blocks):
        """
        Applies inverse 2D DCT (IDCT) to all image blocks.

        Args:
            dct_blocks (numpy.ndarray): 5D array of DCT coefficients with shape
                                        (num_blocks_x, num_blocks_y, block_size_x, block_size_y, channels).

        Returns:
            numpy.ndarray: 5D array of reconstructed image blocks.
        """
        H, W, block_size, _, channels = dct_blocks.shape
        img_blocks = np.zeros_like(dct_blocks, dtype=np.float32)

        for i in range(H):
            for j in range(W):
                for c in range(channels):
                    block = dct_blocks[i, j, :, :, c]
                    img_blocks[i, j, :, :, c] = self.idct2D(block)

        return img_blocks
```

Листинг 2. Класс двумерного дикретного косинусного преобразования

6.3 Класс квантования

Этот набор методов выполняет квантование и деквантование блоков коэффициентов дискретного косинусного преобразования (DCT):

```
        def quantize_dct_bloks(self, bloks, quality=0):
        """
        Quantizes the DCT coefficients using the selected quality factor.

        Args:
            blocks (np.array): DCT blocks to quantize.
            quality (int): Quality factor (0 = default matrices).
```

```
        Returns:
            np.array: Quantized blocks.
        """
        #np.set_printoptions(precision=2, suppress=True)
        # print(bloks[:1, :, :, :, 1])
        if quality == 0:
            H, W, block_size, _, channels = bloks.shape
            self.quantized_blocks = np.zeros_like(bloks, dtype=np.int32)

            for c in range(channels):
                for i in range(H):
                    for j in range(W):
                        self.quantized_blocks[i, j, :, :, c] = np.round(bloks[i, j, :, :, c]
                                                / self.q_matrices[c][:self.block_size, :self.block_size])

            #print(self.quantized_blocks[:1, :, :, :, 1])
            return self.quantized_blocks

        else:
            self.scale_quantiztion_matrices(quality)
            H, W, block_size, _, channels = bloks.shape
            self.quantized_blocks = np.zeros_like(bloks, dtype=np.int32)


            for c in range(channels):
                for i in range(H):
                    for j in range(W):
                        self.quantized_blocks[i, j, :, :, c] = np.round(bloks[i, j, :, :, c]
                                                / self.q_scale_matrices[c][:self.block_size, :self.block_size])

            #print(self.quantized_blocks[:1, :, :, :, 1])
            return self.quantized_blocks




        # Decoded
    def dequantize_blocks(self, quantized_blocks, block_size):

        print(f"DEQUANT: {quantized_blocks.shape}")
        H, W, block_size, _, channels = quantized_blocks.shape
```

```
        dct_blocks = np.zeros_like(quantized_blocks, dtype=np.float32)

        for c in range(channels):
            for i in range(H):
                for j in range(W):
                    dct_blocks[i, j, :, :, c] = quantized_blocks[i, j, :, :,
                        c] * self.q_matrices[c][:self.block_size, :self.
                       block_size]

        return dct_blocks
```

Листинг 3. Класс квантования коэффициентов