МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования

«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА 52

КУРСОВАЯ РАБОТА (ПРОЕКТ)
ЗАЩИЩЕНА С ОЦЕНКОЙ

РУКОВОДИТЕЛЬ

ассистент		А. В. Борисовская
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ОТЧЕТ О КУРСОВОЙ РАБОТЕ

по дисциплине: Мультимедиа технологии

Анализ и реализация иерархического поиска при оценке движения в видеопоследовательности

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. №	5721		А. Е. Ковалева
		подпись, дата	инициалы, фамилия

Санкт-Петербург, 2021

Оглавление

Постановка задачи

Анализ и реализация иерархического поиска при оценке движения в видеопоследовательности.

Описание реализуемой системы

1. Общее описание алгоритма компенсации движении

Несжатый видеосигнал требует большой битовой скорости, из-за чего производится сжатие видео, т.е. происходит уменьшение объема исходной видеопоследовательности.
Для этой цели служат кодер и декодер, кодер переводит данные в более сжатую форму для дальнейшей обработки или хранения, а декодер делает обратное преобразование.

Основным этапом компенсации движения является поиск похожего блока текущего изображения в базовом.

Поиск осуществляется в окрестности данного блока (рисунок 1):

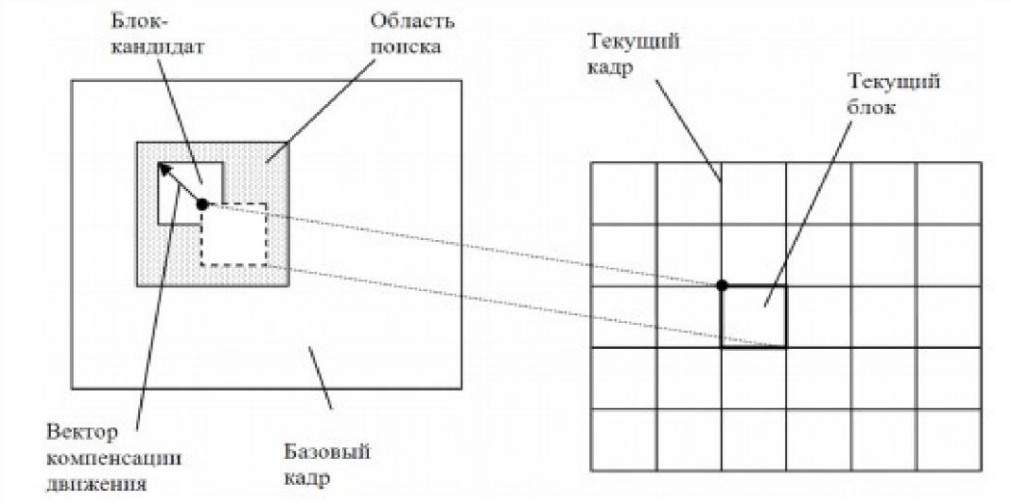

Рисунок 1 – Процедура оценки движения

Результатом оценки движения являются вектора компенсации движения, т. е. то расстояние, на которое блок базового кадра смещен относительно текущего.

Степень похожести блоков оценивается с помощью метрик.

В данной работе использовалась метрика:

Функция абсолютных разностей:

$$F = \sum_{i=0}^{H-1} \sum_{j=0}^{W-1} \left| P_{ij}^{base} - P_{ij}^{cur} \right|$$

Для поиска наиболее похожего блока необходимо минимизировать значение функции.

Для каждого блока выполняется следующая процедура:

1. Поиск в базовом кадре «подходящего» блока.

Это делается путем сравнения фиксированного блока текущего кадра с некоторыми или всеми блоками области поиска и нахождения этого «подходящего» блока. Этот процесс называется оценкой движения.

2. Выбранный кандидат становится прогнозом текущего блока и его вычитают из этого блока для получения разностного блока. Это называется компенсацией движения.

3. Разностный блок кодируется и передается по декодеру, декодер получает координаты вектора смещения текущего блока по отношению к позиции блока-кандидата (вектор движения).

2. Общее описание схемы кодека

Схема кодека имеет вид:

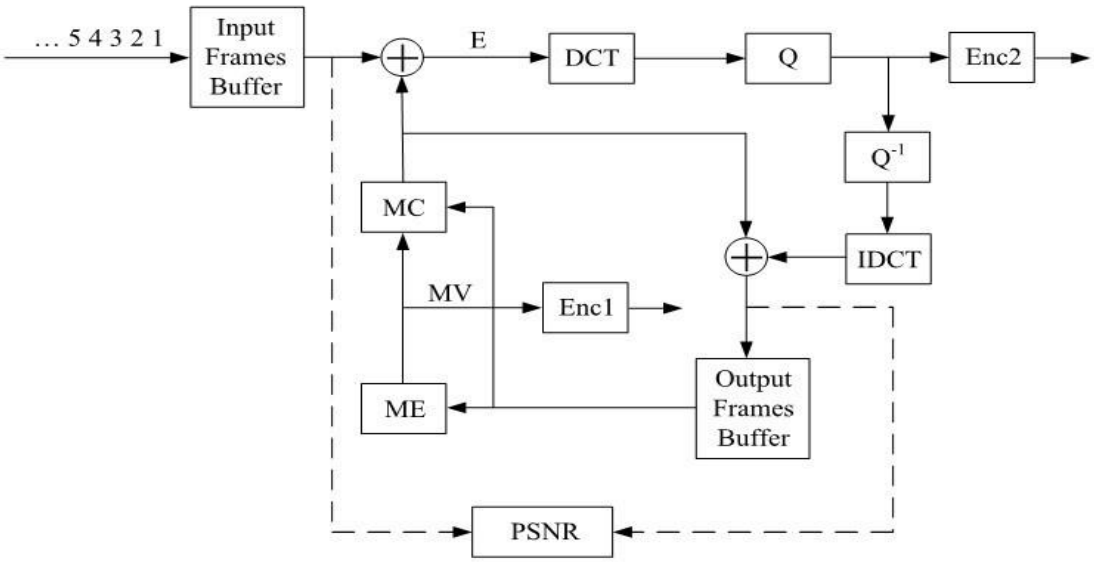

Рисунок 2 – Схема собранного кодека

- На вход подается видеопоследовательность.
- В ***Input Frames Buffer*** хранятся кадры входного видео.
- ***DCT, Q*** – блоки дискретно-косинусного преобразования и квантования. Алгоритмы берутся из стандарта JPEG, после которых в стандарте описано кодирование без потерь.
- ***Q^{-1}, IDCT*** – обратное квантование и обратное дискретно-косинусное преобразование. На выходе этих блоков получается восстановленный разностный кадр.
- ***PSNR*** – блок поиска PSNR между входным и восстановленным кадром.
- В ***Output Frames Buffer*** хранятся кадры выходного видео.
- ***ME*** – оценка движения, на выходе которой получаются вектора оценки движения (***MV***), а также после процедуры компенсации движения получается кадр, состоящий из блоков предыдущего кадра.
- ***MC*** – процедура компенсации движения.
- + - блок поэлементного сложения пикселей кадров.

3. Описание метода иерархического поиска при оценке движения

Идея алгоритмов данной группы заключается в следующем:

Перед началом поиска производится вычисление N - 1 уменьшенных "копий" текущего и опорного кадров, при этом каждая очередная копия в $2n$ (n - натуральное число).

Пары кадров одинакового размера будем называть уровнями, т. е. на одном уровне опорный и текущий кадры одинакового размера. Тогда все множество пар кадров можно представить N уровнями.

Уровни нормируются согласно размеру содержащихся в них кадров от меньшего к большему.

1-й уровень будет содержать кадры минимального размера, N-й - кадры исходного размера.

Процесс оценки движения состоит из N итераций, на каждой из которых обрабатывается пара кадров из уровня с соответствующим номером, т. е. обработка идет от кадров меньшего размера к большему.

На каждой итерации производится оценка движения каким-либо из известных методов, например, трёхшаговый поиск. При этом в качестве стартовой точки на каждой итерации выбирается векторное поле, полученное с предыдущей итерации. Другими словами, каждая очередная итерация производит уточнение векторов, вычисленных на предыдущей итерации.

При переходе на очередную итерацию размеры области поиска и блоков, для которых оцениваются векторы, обычно увеличиваются в $2n$ раз, для того чтобы число блоков в кадре на каждой итерации не менялось.

Схема иерархического поиска представлена на рисунке 3:

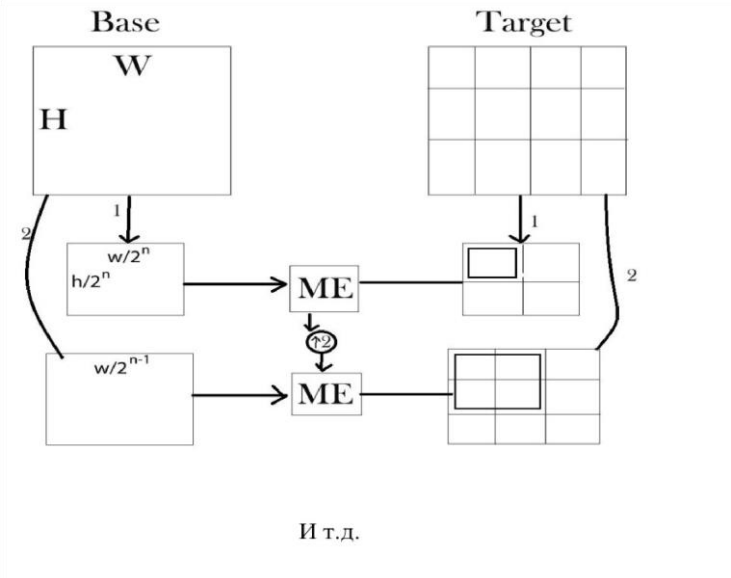

Рисунок 3 — Схема Иерархического поиска

Алгоритм иерархического поиска

Входные параметры:
n – число итераций;
w - длина исходного кадра;
h - ширина исходного кадра.

1 шаг:
Создание n уменьшенных копий базового и обрабатываемого кадра, причём каждая копия в 2 раза меньше предыдущей.
Значения интенсивности пикселя в уменьшенной копии вычисляется по следующей формуле:

$$g_L(p,q) = \left[\frac{1}{4} \left(\sum_{u=0}^{1} \sum_{w=0}^{1} g_{L-1}(2p+u, 2q+w) \right) \right],$$

где $g_L(p, q)$ – значение пикселя уровня L в позиции (p, q).

Таким образом, если использовать три уровня иерархии, один пиксель 2-го уровня соответствует блоку 4x4 0-го уровня и блоку 2x2 1-го уровня соответственно.
В то же время блок размером 16x16 0-го уровня будет соответствовать блоку (16/2L) x16/2L) уровня L.
:
На рисунке 4 представлен пример трёхуровневой иерархической пирамиды:

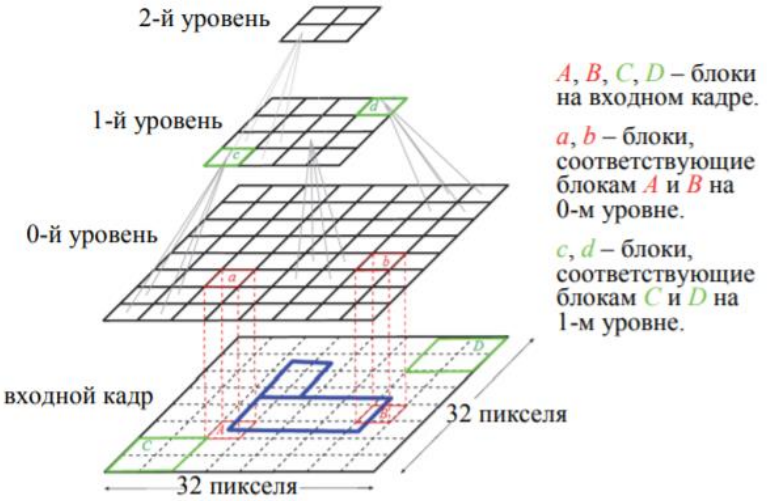

Рисунок 4 - Пример трёхуровневой иерархической пирамиды

2 шаг:
Исходная точка поиска векторов компенсации — самый высокий уровень пирамиды L = n + 1.

3 шаг:
Поиск вектора компенсации на L — уровне.

4 шаг:
Увеличение найденных векторов вдвое. L =L -1

5 шаг:
Пока L >= 0 возврат на 3 шаг.

Трехшаговый алгоритм поиска векторов компенсации на L-ом уровне

На вход алгоритма поступает:

- размер блока bs
- начальный размер шага step (радиус области поиска) = 4
- копия текущего кадра Fcur - копия базового кадра Fbase

1 **шаг:**
Рассматриваются 8 блоков на начальном расстоянии от центра (текущего блока).

2 **шаг:**
Начальное расстояние между центрами блоков уменьшается вдвое, центр поиска сдвигается в точку с минимальными искажениями.

3 **шаг:**
Шаги 1 и 2 повторяются до тех пор, пока начальный шаг не будет меньше единицы.

Схема алгоритма представлена на рисунке 5:

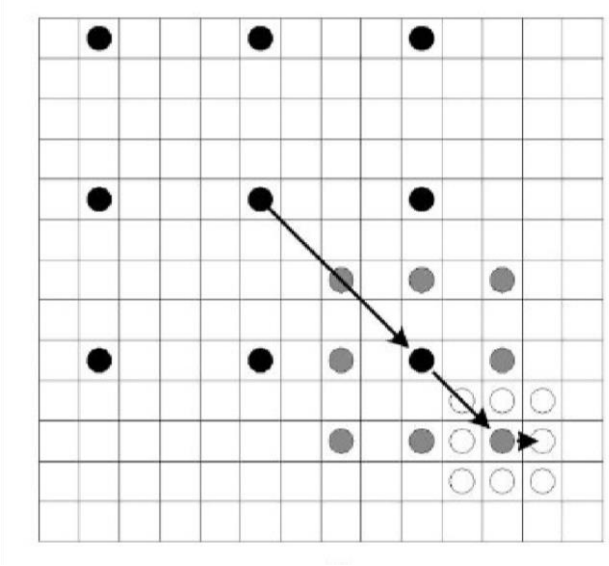

Рисунок 5 — Пример трехшагового алгоритма поиска векторов компенсации на L-ом уровне.

Также в данной работе использовался метод градиентного спуска.

Алгоритм поиска векторов компенсации выглядит следующим образом:

1. Координаты верхнего левого пикселя становятся центром области поиска в базовом кадре.
2. Проверяются все блоки входящие в область поиска с шагом 1 пиксель.
   Радиус области = 1
3. Выбирается наилучший блок и центр области поиска переходит в данную точку.
4. Повторяются шаги 2 и 3 до тех пор, пока значения векторов компенсации на данном шаге итерации не совпадут со значениями на предыдущем шаге.

Результаты работы

Исходные данные

В ходе выполнения курсовой работы были исследованы 2 видеопоследовательности.

Рисунок 6 – Кадр первого видео

Рисунок 7 – Кадр второго видео

Результаты программной реализации

Для анализа иерархического поиска в данной работе была реализована схема сжатия JPEG + ME, которая представлена на рисунке 2.

В ходе выполнения работы были найдены следующие зависимости:

1. Зависимость MSE от номера кадра для трёхшагового поиска и для градиентного спуска;
2. Зависимость PSNR от шага квантования для трёхшагового поиска и для градиентного спуска.

Далее рассмотрим их подробно.

1. Зависимость MSE от номера кадра

MSE - средний квадрат ошибки, который вычисляется между текущим кадром и кадром, построенным с помощью векторов компенсации, вычисленным по предыдущему восстановленному кадру.

$$MSE = \frac{1}{HW} \sum_{i=0}^{H-1} \sum_{j=0}^{W-1} \left| F_{cur\,ij} - F_{rec\,ij} \right|^2$$

,

где
H — ширина кадра;
W — длина кадра;
F_{cur} — текущей кадр;
F_{rec} — предыдущий восстановленный кадр.

Процедура восстановления кадра

Исходные данные:

- Вектора компенсации - V
- Предыдущий восстановленный кадр (базовый кадр) — F_{base}
- Текущий кадр - F_{cur}
- Разностный кадр - F_{dif}
- Новый кадр F_{new}, полученный с помощью V и F_{base}

Алгоритм действий:

1. По векторам компенсации V и базовому кадру F_{base} строим новый кадр F_{new}
2. Находим разностный кадр путём вычитания нового кадра из текущего кадра:
3. Fdif = Fcur - Fnew .
4. Над разностным кадром производим следующие преобразования
5. $F_{dif} \rightarrow DCT \rightarrow Q \rightarrow Q^{-1} \rightarrow IDCT = F_{dec}$ (декодированный разностный кадр).
6. Восстановленный кадр F_{rec} получаем путём сложения F_{dec} и F_{new} : $F_{rec} = F_{dec} + Fnew$.

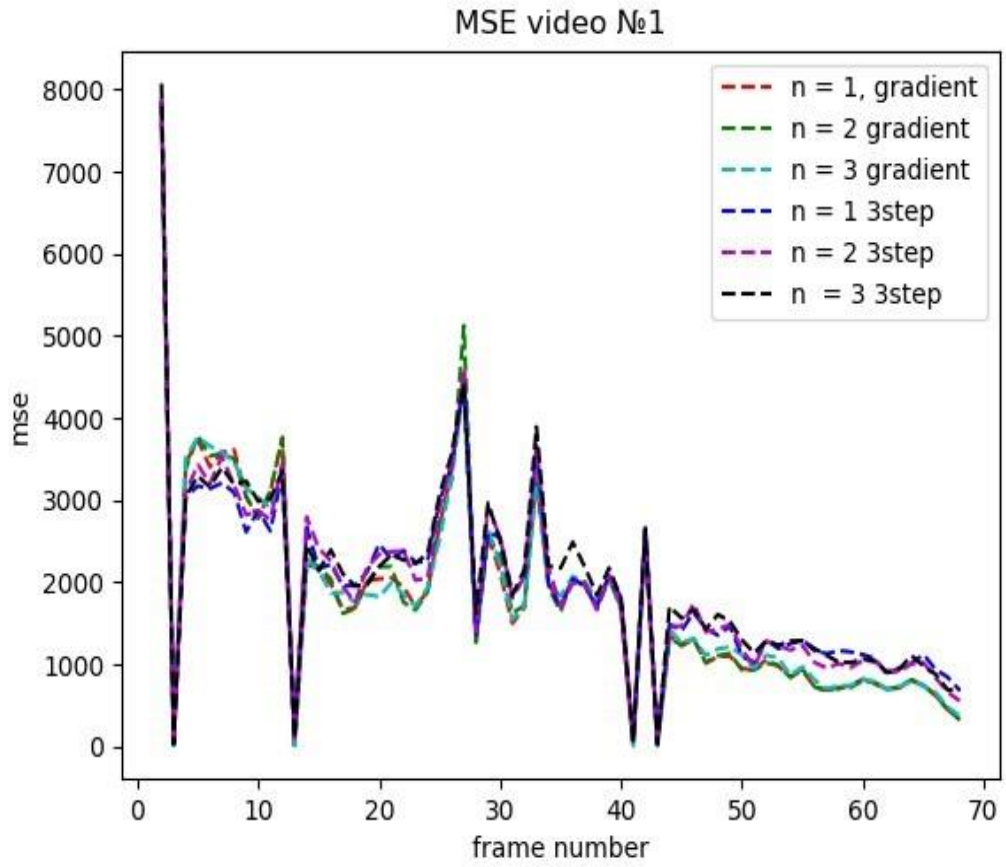

Рисунок 8 — Зависимость MSE от номера кадра для 1 видеопоследовательности

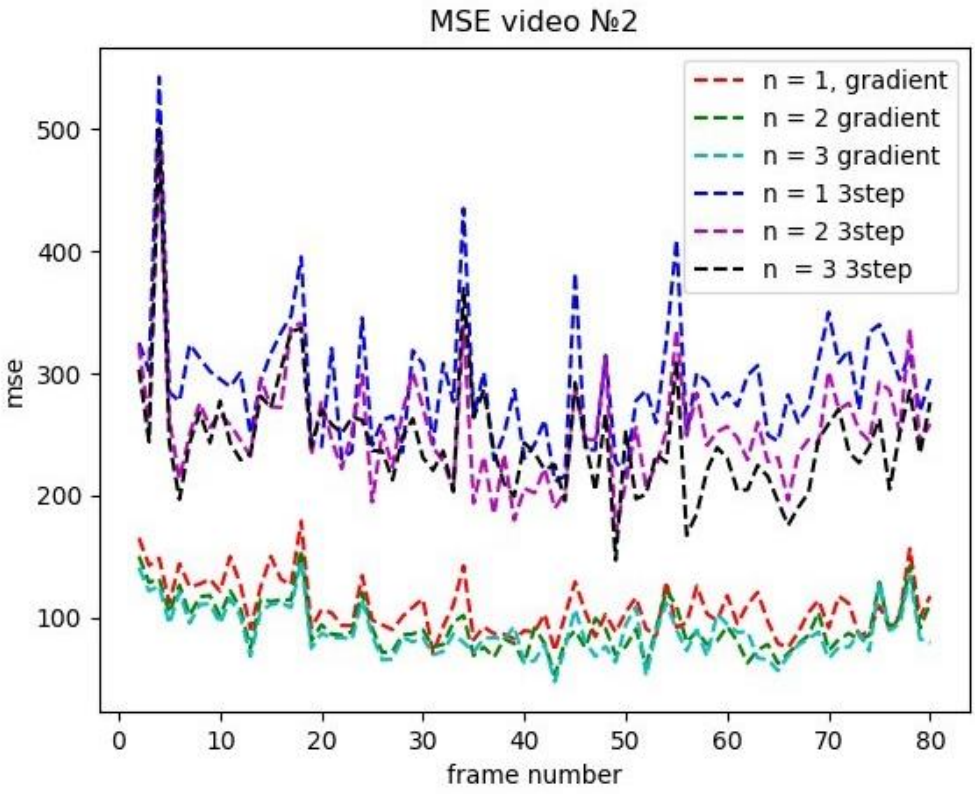

Рисунок 9 — Зависимость MSE от номера кадра для 2 видеопоследовательности

По графикам на рисунках 8 и 9 видно, что квадрат ошибки с увеличением числа уровней в иерархии при оценке движения уменьшается. Особенно это заметно для второй видеопоследовательности.

Таким образом, чем больше уровней иерархии, тем качество сжатия лучше.

Если сравнивать значения квадрата ошибки двух видеопоследовательностей, можно наблюдать, что MSE для первого видео варьируется от 50 до 8000, а для второго видео от 50 до 550. То есть первое видео при сжатии потеряло больше информации, это обуславливается тем, что оно характеризуется более быстрым движением и более сильной детализацией, с такими характеристиками вектора компенсации находятся не так точно, как для медленных видео с меньшей детализацией.

На графиках для каждого видео изображены результаты при использовании методов трёхшагового поиска и градиентного спуска.

Для первой видеопоследовательности отличий в значениях MSE для обоих методов практически нет, а для второй видеопоследовательности со статичным движением средний квадрат ошибки при использовании метода градиентного спуска даёт выигрыш по сравнению с трёхшаговым поиском.

Ниже представлены отдельные восстановленные кадра для двух, трёх и четырёх уровней иерархии соответственно.

1) Кадры первого видео

Кадры для первого видео	Кадры для второго видео
	
Визуально разница в качестве восстановленных кадров не сильно заметна.	

2. Зависимость PSNR от шага квантования

$$PSNR = 10\log_{10}\frac{HW\left(2^8-1\right)^2}{\sum_{i=0}^{H-1}\sum_{j=0}^{W-1}\left(F_{cur\,ij}-F_{rec\,ij}\right)^2}$$

Зависимость PSNR от шага квантования была найдена для второго кадра каждой видеопоследовательности и для трёх значений n.

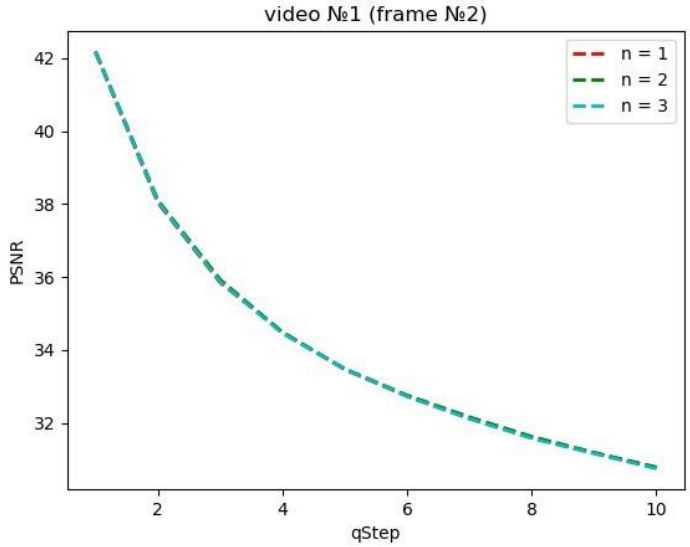

Рисунок 10 — Зависимость PSNR от шага квантования для первого видео.

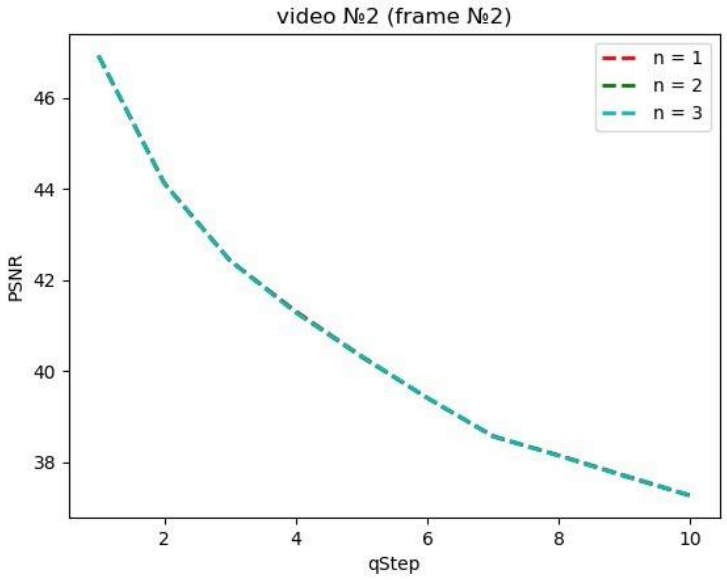

Рисунок 11 — Зависимость PSNR от шага квантования для второго видео.

По графикам на рисунках 11 и 12 видно, что значения PSNR уменьшаются с увеличением шага квантования, так как во время процедуры квантования происходит потеря информации, количество которой зависит как раз от шага квантования.

Также можно заметить, что значения PSNR для разного числа уровней иерархии отличается не сильно для данного кадра (второго кадра).

Выводы

В ходе выполнения работы был исследован алгоритм компенсации движения – иерархический поиск, реализована схема кодека.

Для реализации иерархического метода оценки движения были выбраны два метода поиска — трёхшаговый поиск и градиентный спуск.

Были исследованы:

1. зависимость квадрата ошибки MSE от номера кадра для разного числа уровней иерархии;
2. зависимость PSNR от шага квантования.

Выяснилось, что

1. для видео с медленным движением метод градиентного спуска даёт более весомый выигрыш в качестве сжатия;
2. с увеличением числа уровней иерархии улучшается качество сжатия.

Список использованных источников

1. Нгуен Ван Чыонг, Тропченко А.А. Иерархический адаптивный алгоритм шаблонного поиска для оценки движения при анализе видеопоследовательности // Научно-технический вестник информационных технологий, механики и оптики. 2016. Т. 16. № 3. С. 474–481. doi: 10.17586/ 2226-1494-2016-16-3-474-481

2. Лекции по дисциплине «Мультимедиа технологии», 52 каф., СПБГУАП

Листинг программы

```
import av
import av.datasets
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
from numba import jit

def lineplot3(x_data, y1_data, y2_data, y3_data, x_label="", y_label="", title="",y1_legend="",y2_legend="", y3_legend=""):
    _, ax = plt.subplots()
    ax.plot(x_data, y1_data, 'r--', lw=2)
    ax.plot(x_data, y2_data, 'g--', lw=2)
    ax.plot(x_data, y3_data, 'c--', lw=2)
    ax.set_title(title)
    ax.set_xlabel(x_label)
    ax.set_ylabel(y_label)
    ax.legend([y1_legend, y2_legend, y3_legend])

def lineplot6(x, y1, y2, y3, y4, y5, y6, x_label="", y_label="", title="",y1_l="",y2_l="", y3_l="", y4_l="", y5_l="", y6_l=""):
    _, ax = plt.subplots()
    ax.plot(x, y1, 'r--', lw=1.5)
    ax.plot(x, y2, 'g--', lw=1.5)
    ax.plot(x, y3, 'c--', lw=1.5)
    ax.plot(x, y4, 'b--', lw=1.5)
    ax.plot(x, y5, 'm--', lw=1.5)
    ax.plot(x, y6, 'k--', lw=1.5)
    # Label the axes and provide a title
    ax.set_title(title)
    ax.set_xlabel(x_label)
    ax.set_ylabel(y_label)
    ax.legend([y1_l, y2_l, y3_l, y4_l, y5_l, y6_l])

@jit(nopython=True)
def sum_abs_dif(cur, base, block_size):
    F = 0
    for i in range(block_size):
        for j in range(block_size):
            F += np.abs(cur[i][j] - base[i][j])
    return F

@jit(nopython=True)
def gradient(cur, base, H, W, i, j, block_size):
    R = 1
    best = 1000000
    cur_block = cur[i: i + block_size, j: j + block_size]
    dx = 0
    dy = 0
    prev_dx = -1
    prev_dy = -1
    newx = j
    newy = i
    while prev_dx != dx and prev_dy != dy:
        start_y = newy - R
        if start_y < 0:
            start_y = 0
        start_x = newx - R
        if start_x < 0:
            start_x = 0

        end_x = newx + R
        end_y = newy + R
        if end_x + block_size > W - 1:
            end_x = W - block_size
        if end_y + block_size > H - 1:
            end_y = H - block_size
        prev_dx = dx
        prev_dy = dy
```

```
        for y in np.arange(start_y, end_y, R):
            for x in np.arange(start_x, end_x, R):
                base_block = base[y: y + block_size, x:x + block_size]
                m = sum_abs_dif(cur_block, base_block, block_size)
                if m < best:
                    best = m
                    dx = x - j
                    dy = y - i
                    newx = x
                    newy = y
    return dx, dy

@jit(nopython=True)
def three_steps_search(cur, base, H, W, i, j, R, block_size):
    dx = 0
    dy = 0
    best = 1000000
    cur_block = cur[i: i + block_size, j: j + block_size]
    step = R
    newy = i
    newx = j
    while step > 0:
        start_y = newy - step
        if start_y < 0:
            start_y = 0
        start_x = newx - step
        if start_x < 0:
            start_x = 0

        end_x = newx + step
        end_y = newy + step
        if end_x + block_size > W - 1:
            end_x = W - block_size
        if end_y + block_size > H - 1:
            end_y = H - block_size
        for y in np.arange(start_y, end_y, step):
            for x in np.arange(start_x, end_x, step):
                base_block = base[y: y + block_size, x:x + block_size]
                m = sum_abs_dif(cur_block, base_block, block_size)
                if m < best:
                    best = m
                    dx = x - j
                    dy = y - i
                    newx = x
                    newy = y
        step //= 2

    return dx, dy

def HARPS(base, cur, H, W, bs, n, grad=True):
    base_level = []
    cur_level = []
    h = H
    w = W
    base_level.append(base)
    cur_level.append(cur)
    for k in range(1, n+1):
        h //= 2
        w //= 2
        base_level.append(np.zeros((h, w), np.float64))
        cur_level.append(np.zeros((h, w), np.float64))

    for k in range(1, len(base_level)):
        for i in range(base_level[k].shape[0]):
            for j in range(base_level[k].shape[1]):
                base_level[k][i][j] = (base_level[k-1][2*i][2*j] + base_level[k-1][2*i + 1][2*j]
                                       + base_level[k-1][2*i][2*j + 1] + base_level[k-1][2*i+1][2*j+1])/4

    for k in range(1, len(cur_level)):
```

```
        for i in range(cur_level[k].shape[0]):
            for j in range(cur_level[k].shape[1]):
                cur_level[k][i][j] = (cur_level[k-1][2*i][2*j] + cur_level[k-1][2*i + 1][2*j]
                                      + cur_level[k-1][2*i][2*j + 1] + cur_level[k-1][2*i+1][2*j+1])/4

    #k = int(block_size // (2**n))
    h = cur_level[-1].shape[0]
    w = cur_level[-1].shape[1]
    vectors_prev = []
    for i in range(0, cur_level[-1].shape[0], bs):
        for j in range(0, cur_level[-1].shape[1], bs):
            if grad:
                dx, dy = gradient(cur_level[-1], base_level[-1], h, w, i, j, bs)
            else:
                dx, dy = three_steps_search(cur_level[-1], base_level[-1], h, w, i, j, 4, bs)
            vectors_prev.append((dx, dy))

    for N in range(-2, -len(cur_level) - 1, -1):
        h = cur_level[N].shape[0]
        w = cur_level[N].shape[1]
        k = 0
        vectors_new = []
        for y1 in range(0, h, bs*2):
            for x1 in range(0, w, bs*2):
                dx = vectors_prev[k][1]
                dy = vectors_prev[k][0]
                for y2 in range(y1, y1 + 2*bs, bs):
                    for x2 in range(x1, x1 + 2*bs, bs):
                        if grad:
                            dx_new, dy_new = gradient(cur_level[N], base_level[N], h, w, y2+dy, x2+dx, bs)
                        else:
                            dx_new, dy_new = three_steps_search(cur_level[N], base_level[N], h, w, y2+dy, x2+dx, 4, bs)
                        vectors_new.append((dy_new*2, dx_new*2))
                k += 1
        vectors_prev = vectors_new

    return vectors_new

#JPEG with vectors monotonous search
def do_scheme_hiearchical_search(video_name, frames, W, H, format, codec, rate, block_size, Qstep, n, grad=True):

    clip_np = np.vectorize(clip)
    mse = []
    #recovered_frames = []
    vectors = np.zeros(len(frames), dtype=object)
    dif_frames = []
    new_frames = []
    h = H//(2**n)
    w = W//(2**n)
    bs = block_size
    while np.mod(h*w, bs) != 0:
        bs //= 2
    for i in range(len(frames)):
        cur = frames[i]
        if i == 0:
            new_frames.append(frames[i])
        else:
            base = frames[i - 1]
            if grad:
                cur_vectors = HARPS(base, cur, H, W, bs, n, grad=True)
            else:
                cur_vectors = HARPS(base, cur, H, W, bs, n, grad=False)
            vectors[i] = cur_vectors
            new_frame = get_new_frame(base, H, W, bs, cur_vectors)
            dif_frame = cur - new_frame
            mse.append(MSE(dif_frame, H, W))
            dif_frames.append(clip_np(dif_frame + 128))
            new_frames.append(new_frame)
    if grad:
```

```
        save_video(new_frames, W, H, format, codec, rate, video_name + "_gradient_new_n=" + str(n))
        save_video(dif_frames, W, H, format, codec, rate, video_name + "_gradient_dif_n=" + str(n))
    else:
        save_video(new_frames, W, H, format, codec, rate, video_name + "_3step_new_n=" + str(n))
        save_video(dif_frames, W, H, format, codec, rate, video_name + "_3step_dif_n=" + str(n))
    return mse

@jit( nopython=True )
def get_new_frame(base, H, W, block_size, vectors):
    new_frame = np.zeros((H, W), dtype=np.int64)
    number_block = 0
    for y in range(0, H, block_size):
        for x in range(0, W, block_size):
                dx = vectors[number_block][1]
                dy = vectors[number_block][0]
                for i in range(y, y + block_size):
                    for j in range(x, x + block_size):
                            Y = base[i + int(dy)][j + int(dx)]
                            new_frame[i][j] = Y
                number_block += 1
    return new_frame

def clip(value):
    return 0 if value < 0 else 255 if value > 255 else value

@jit( nopython=True )
def form_frame_on_ME_vectors(base, H, W, block_size1, block_size2, vectors):
    new_frame = np.zeros((H, W), dtype=np.int64)
    number_block = 0
    for y1 in range(0, H, block_size1):
        for x1 in range(0, W, block_size1):
            for y2 in range(y1, y1 + block_size1, block_size2):
                for x2 in range(x1, x1 + block_size1, block_size2):
                    dx = vectors[number_block][1]
                    dy = vectors[number_block][0]
                    for i in range(y2, y2 + block_size2):
                        for j in range(x2, x2 + block_size2):
                            Y = base[i + int(dy)][j + int(dx)]
                            new_frame[i][j] = Y
                    number_block += 1
    return new_frame

@jit(nopython=True)
def clipping(frames, num_frames, H,W):
    res = np.ndarray((num_frames, H, W), np.int64)
    for k in range(num_frames):
        for i in range(H):
            for j in range(W):
                res[k][i][j] = clip(frames[k][i][j])
    return res

def DCT(frame, C, Ct):
    h = frame.shape[0]
    w = frame.shape[1]
    res = np.zeros((h,w),dtype=np.float64)
    i = 0
    for ki in range(h//8):
        j = 0
        for kj in range(w//8):
            pixels = frame[i:i + 8, j:j + 8]
            tmp1 = np.ndarray((8, 8), dtype=np.float64)
            np.matmul(pixels, Ct, tmp1)
            tmp2 = np.ndarray((8, 8), dtype=np.float64)
            np.matmul(C, tmp1, tmp2)
            y2 = 0
            for y in range(i, i + 8):
                x2 = 0
                for x in range(j, j + 8):
                    res[y][x] = tmp2[y2][x2]
```

```
                x2 += 1
            y2 += 1
        j += 8
    i += 8
return res

def IDCT(dct, C, Ct):
    h = dct.shape[0]
    w = dct.shape[1]
    res = np.zeros((h,w), dtype=np.float64)
    i = 0
    for ki in range(h//8):
        j = 0
        for kj in range(w//8):
            pixels = dct[i:i + 8, j:j + 8]
            tmp1 = np.ndarray((8, 8), dtype=np.float64)
            np.matmul(pixels, C, tmp1)
            tmp2 = np.ndarray((8, 8), dtype=np.float64)
            np.matmul(Ct, tmp1, tmp2)
            y2 = 0
            for y in range(i, i+8):
                x2  = 0
                for x in range(j, j+8):
                    res[y][x] = tmp2[y2][x2]
                    x2 += 1
                y2 += 1
            j += 8
        i += 8
    return res

def Q(dct, basic_table):
    h = dct.shape[0]
    w = dct.shape[1]
    q = np.ndarray((h, w), 'int')
    i = 0
    for ki in range(h//8):
        j = 0
        for kj in range(w//8):
            y2 = 0
            for y in range(i, i + 8):
                x2 = 0
                for x in range(j, j + 8):
                    q[y][x] = round(dct[y][x]/basic_table[y2][x2])
                    x2 += 1
                y2 += 1
            j += 8
        i += 8
    return q

def deQ(q, basic_table):
    h = q.shape[0]
    w = q.shape[1]
    dct = np.zeros((h, w), 'int')
    i = 0
    for ki in range(h//8):
        j = 0
        for kj in range(w//8):
            y2 = 0
            for y in range(i, i + 8):
                x2 = 0
                for x in range(j, j + 8):
                    dct[y][x] = q[y][x]*basic_table[y2][x2]
                    x2 += 1
                y2 += 1
            j += 8
        i += 8
    return dct

@jit(nopython=True)
```

```
def MSE(signal, h, w):
    res = 0
    for i in range(h):
        for j in range(w):
            res += signal[i][j]**2
    return res/(h*w)

#PSNR
@jit(nopython=True)
def PSNR(signal1, signal2):
    numerator = len(signal1)*len(signal1[0])*(2**8 - 1)**2
    denominator = 0
    for j in range(len(signal1)):
        for i in range(len(signal1[0])):
            denominator += (signal1[j][i] - signal2[j][i])**2
    if denominator == 0: return np.inf
    return 10*np.log10(numerator/denominator)

#чтение-запись
def read_video_toY(video_name):
    video = av.open(str(video_name) + '.avi')
    stream = video.streams.video[0]
    H = stream.height
    W = stream.width
    format = stream.pix_fmt
    codec = stream.codec_context.name
    rate = stream.average_rate
    frames = []
    for frame in video.decode(video=0):
        Y = np.array(frame.to_image().convert('YCbCr'))[:, :, 0].astype('float64')
        frames.append(Y)
#    video.close()
    return frames, W, H, format, codec, rate

def save_video(Y_frames, W, H, format, codec, rate, video_name):
    output = av.open(str(video_name) + '.AVI', 'w')
    stream_out = output.add_stream(codec, rate=rate)
    stream_out.height = H
    stream_out.width = W
    stream_out.pix_fmt = format

    for frame in Y_frames:
        i = Image.fromarray(frame.astype(np.uint8)).convert('RGB')
        frame = av.VideoFrame.from_image(i)
        for packet in stream_out.encode(frame):
            output.mux(packet)
    output.close()

def main():
    video_name_1 = "LR1_1"
    video_name_3 = "LR1_3"
    block_size = 16
    N = [1, 2, 3]

    frames1, W1, H1, format1, codec1, rate1 = read_video_toY(video_name_1)
    frames3, W3, H3, format3, codec3, rate3 = read_video_toY(video_name_3)

    MSE1_3Step = []
    MSE3_3Step = []
    MSE1_g = []
    MSE3_g = []
    x1 = np.arange(2, len(frames1) + 1)
    x3 = np.arange(2, len(frames3) + 1)

    for n in N:
        mse = do_scheme_hiearchical_search(video_name_1, frames1, W1, H1, format1, codec1, rate1, block_size, 1, n,grad=True)
        MSE1_g.append(mse)
        mse = do_scheme_hiearchical_search(video_name_3, frames3, W3, H3, format3, codec3, rate3, block_size, 1, n,grad=True)
        MSE3_g.append(mse)
```

```
        mse = do_scheme_hiearchical_search(video_name_1, frames1, W1, H1, format1, codec1, rate1, block_size, 1, n,grad=False)
        MSE1_3Step.append(mse)
        mse = do_scheme_hiearchical_search(video_name_3, frames3, W3, H3, format3, codec3, rate3, block_size, 1, n,grad=False)
        MSE3_3Step.append(mse)

    lineplot6(x1, MSE1_g[0], MSE1_g[1], MSE1_g[2], MSE1_3Step[0], MSE1_3Step[1], MSE1_3Step[2], "frame number", "mse",
              "MSE video №1", "n = 1, gradient", "n = 2 gradient", "n = 3 gradient", "n = 1 3step","n = 2 3step", "n  = 3 3step")
    plt.savefig("mse_video1.png")

    lineplot6(x3, MSE3_g[0], MSE3_g[1], MSE3_g[2], MSE3_3Step[0], MSE3_3Step[1], MSE3_3Step[2], "frame number", "mse",
              "MSE video №2", "n = 1, gradient", "n = 2 gradient", "n = 3 gradient", "n = 1 3step","n = 2 3step", "n  = 3 3step")
    plt.savefig("mse_video3.png")

    lineplot3(x1, MSE1_g[0], MSE1_g[1], MSE1_g[2], "frame number", "mse", "video №1 gradient", "n = 1", "n = 2", "n = 3")
    plt.savefig("video1_mse_gradient.png")

    lineplot3(x3, MSE3_g[0], MSE3_g[1], MSE3_g[2], "frame number", "mse", "video №2 gradient", "n = 1", "n = 2", "n = 3")
    plt.savefig("video3_mse_gradient.png")

    lineplot3(x1, MSE1_3Step[0], MSE1_3Step[1], MSE1_3Step[2], "frame number", "mse", "video №1 3step", "n = 1", "n = 2","n =
3")
    plt.savefig("video1_mse_3step.png")

    lineplot3(x3, MSE3_3Step[0], MSE3_3Step[1], MSE3_3Step[2], "frame number", "mse", "video №2 3step", "n = 1", "n = 2", "n =
3")
    plt.savefig("video3_mse_3step.png")

    print("Completed!!!")

if __name__ == '__main__':
    main()
```