



Министерство науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«Московский государственный технический университет  
имени Н.Э. Баумана  
(национальный исследовательский университет)»  
(МГТУ им. Н.Э. Баумана)

---

ФАКУЛЬТЕТ ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ

КАФЕДРА ПРИКЛАДНАЯ МАТЕМАТИКА И ИНФОРМАТИКА

# РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

## *К КУРСОВОЙ РАБОТЕ*

*НА ТЕМУ:*

*Применение сингулярного разложения матрицы  
для сжатия изображений*

Студент ИУ9-72Б  
(Группа)

\_\_\_\_\_  
(Подпись, дата)

П. С. Самохвалова  
(И.О.Фамилия)

Руководитель курсовой работы

\_\_\_\_\_  
(Подпись, дата)

Д. П. Посевин  
(И.О.Фамилия)

Консультант

\_\_\_\_\_  
(Подпись, дата)

\_\_\_\_\_  
(И.О.Фамилия)

2023 г.

## СОДЕРЖАНИЕ

ПОСТАНОВКА ЗАДАЧИ.....	4
1. Сингулярное разложение матрицы.....	5
2. Области применения сингулярного разложения матрицы.....	5
3. Методы нахождения сингулярного разложения матрицы.....	6
3.1. Сингулярное разложение с использованием поиска собственных значений матрицы.....	7
3.1.1. Нахождение собственных значений и собственных векторов матрицы методом А. М. Данилевского.....	8
3.1.2. Нахождение собственных значений и собственных векторов матрицы методом А. Н. Крылова.....	13
3.1.3. Реализация сингулярного разложения с использованием поиска собственных значений.....	14
3.2. Сингулярное разложение с использованием QR-разложения.....	15
3.2.1. Нахождение QR-разложения матрицы.....	18
3.2.2. Реализация сингулярного разложения с использованием QR-разложения.....	18
3.3. Итерационные методы.....	22
4. Методы сжатия изображений.....	26
5. Применение сингулярного разложения матрицы для задачи сжатия изображений .....	29
6. Результаты сжатия изображений при помощи сингулярного разложения.....	33
ЗАКЛЮЧЕНИЕ.....	47
СПИСОК ЛИТЕРАТУРЫ.....	48

ПРИЛОЖЕНИЕ.....	50
-----------------	----

## ПОСТАНОВКА ЗАДАЧИ

Целью данной курсовой работы является реализация сингулярного разложения матрицы и применение его для задачи сжатия изображений.

Для достижения поставленной цели необходимо решить следующие задачи:

- рассмотреть различные алгоритмы сингулярного разложения матрицы;
- реализовать алгоритмы сингулярного разложения матрицы на языке программирования Python;
- оценить точность различных алгоритмов сингулярного разложения матрицы;
- рассмотреть методы сжатия изображений;
- применить сингулярное разложение матрицы для задачи сжатия изображений.

## **1. Сингулярное разложение матрицы**

Любая матрица (вещественная или комплексная) может быть представлена в виде произведения трех матриц:

$$X=U\Sigma V^*,$$

где  $U$  – унитарная матрица порядка  $m$ ;  $\Sigma$  – матрица размера  $m \times n$ , на главной диагонали которой лежат неотрицательные числа, называемые сингулярными числами матрицы  $X$ , элементы вне главной диагонали равны нулю;  $V^*$  – эрмитово-сопряжённая к  $V$  матрица порядка  $n$ ;  $m$  столбцов матрицы  $U$  и  $n$  столбцов матрицы  $V$  называются соответственно левыми и правыми сингулярными векторами матрицы  $X$ .

Сингулярное разложение является удобным методом при работе с матрицами. Сингулярное разложение показывает геометрическую структуру матрицы и позволяет наглядно представить имеющиеся данные.

Сингулярное разложение используется при решении различных задач – начиная с приближения методом наименьших квадратов и решения систем уравнений и заканчивая сжатием и распознаванием изображений.

Усеченное сингулярное разложение матриц хорошо подходит для решения плохо обусловленных систем линейных уравнений [2].

## **2. Области применения сингулярного разложения матрицы**

Сингулярное разложение матрицы находит широкое применение в различных областях, включая линейную алгебру, статистику, обработку сигналов, машинное обучение и компьютерное зрение. Рассмотрим несколько конкретных примеров использования SVD.

- 1) Понижение размерности. SVD часто используется для уменьшения размерности данных. Это позволяет эффективно представлять и анализировать данные, удаляя шум и избыточность.

- 2) Решение систем линейных уравнений. SVD может быть использовано для решения переопределенных систем линейных уравнений.
- 3) Рекомендательные системы. SVD применяется в методах коллаборативной фильтрации, используемых в рекомендательных системах, для выявления скрытых паттернов в оценках пользователей.
- 4) Компрессия изображений и сжатие данных. SVD может быть использовано для сжатия изображений и других типов данных, позволяя сохранить основные характеристики и сэкономить место в хранении.
- 5) Анализ данных. SVD позволяет выделить наиболее важные компоненты в данных, что полезно при анализе и визуализации данных.
- 6) Обработка сигналов. В обработке сигналов, например, в контексте фильтрации и сжатия сигналов, SVD находит свое применение.

Это лишь несколько примеров использования сингулярного разложения матрицы. Однако его возможности шире, и оно продолжает находить новые приложения в различных областях науки и техники.

### **3. Методы нахождения сингулярного разложения матрицы**

Существует несколько методов для нахождения сингулярного разложения матрицы.

- 1) Сингулярное разложение с использованием собственных значений. Этот метод часто используется для нахождения SVD для квадратных матриц. Он включает в себя нахождение собственных значений и собственных векторов матрицы  $A^T \cdot A$ , а затем вычисление левых и правых сингулярных векторов.

- 2) Сингулярное разложение с использованием QR-разложения. Этот метод часто применяется для нахождения SVD прямоугольных матриц. Он включает вычисление QR-разложения исходной матрицы и применение его компонент к нахождению левых и правых сингулярных векторов.
- 3) Итерационные методы. Эти методы используют итерационные алгоритмы для поиска приближенного SVD. Они могут быть полезны для больших матриц, когда вычисление точного SVD неоправданно «затратно».

### **3.1. Сингулярное разложение с использованием поиска собственных значений матрицы**

Для нахождения сингулярного разложения матрицы с использованием собственных значений можно применить следующий метод.

- 1) Найти собственные значения матрицы  $A^T \cdot A$ . Для этого необходимо сначала найти матрицу  $A^T \cdot A$ , затем найти её собственные значения.
- 2) Найти собственные векторы матрицы  $A^T \cdot A$ . Каждый собственный вектор соответствует найденному собственному значению.
- 3) Вычислить сингулярные значения матрицы  $A$ , как корни из найденных собственных значений матрицы  $A^T \cdot A$ .
- 4) Найти левые сингулярные векторы матрицы  $A$ , как произведения исходной матрицы на собственные векторы, разделенное на сингулярные значения.
- 5) Найти правые сингулярные векторы, как собственные векторы матрицы  $A^T \cdot A$ .

Этот метод позволяет найти сингулярное разложение матрицы  $A$  с помощью собственных значений.

В ходе работы было реализовано нахождение собственных значений и собственных векторов матрицы методами А. М. Данилевского и А. Н. Крылова (листинги 1 – 5).

### 3.1.1. Нахождение собственных значений и собственных векторов матрицы методом А. М. Данилевского

Метод А. М. Данилевского используется для нахождения собственных значений и собственных векторов матрицы. Основная идея метода заключается в преобразовании матрицы к Фробениусовой форме, что позволяет найти характеристический многочлен и, соответственно, собственные значения матрицы.

Для реализации метода А. М. Данилевского были написаны две вспомогательные функции – `div_half_method`, реализующая метод половинного деления, и `gershgorin_rounds`, реализующая метод Гершгорина для оценки спектра матрицы.

На рисунке 1 представлен код функции `div_half_method`.

```
def div_half_method(a, b, f):  
    s = 0.1  
    d = 0.0001  
    res = []  
    x_last = a  
    x = x_last  
    while x <= b:  
        x = x_last + s  
        if f(x) * f(x_last) < 0:  
            x_left = x_last  
            x_right = x  
            x_mid = (x + x_last) / 2  
            while abs(f(x_mid)) >= d:  
                if f(x_left) * f(x_mid) < 0:  
                    x_right = x_mid  
                else:  
                    x_left = x_mid  
                x_mid = (x_left + x_right) / 2  
            res.append(x_mid)  
        x_last = x  
    return res
```

Рисунок 1 – Код функции `div_half_method`



Приведённая функция представляет метод половинного деления или метод бисекции, который решает уравнение  $f(x) = 0$  в заданном интервале  $[a, b]$ .

Приведём ниже алгоритм, реализуемый данной функцией:

- 1) Сначала инициализируются значения переменных  $s$  – шаг итерации и  $d$  – минимальная разность между ожидаемым корнем и приближенным корнем.
- 2) Создается пустой список `res`, в который будут добавляться найденные корни уравнения.
- 3) Устанавливается начальное значение  $x\_last$  равным  $a$ , и  $x$  принимает это значение.
- 4) Запускается цикл, пока  $x$  меньше или равен  $b$ .
- 5) Значение  $x$  обновляется путем добавления шага  $s$  к  $x\_last$ .
- 6) Проверяется знак функции  $f(x) \cdot f(x\_last)$ . Если он отрицателен, то на данном интервале возможно существует корень уравнения  $f(x) = 0$ .
- 7) Далее определяются границы интервала, в котором находится корень:  $x\_left = x\_last$ ,  $x\_right = x$ ,  $x\_mid = (x + x\_last) / 2$ .
- 8) Запускается вложенный цикл, где происходит итерационный процесс метода бисекции: пока модуль значения функции  $f(x\_mid)$  больше или равен  $d$ , выполняются итерации метода.
- 9) Внутренний цикл обновляет границы интервала в соответствии с условием изменения знака функции на интервале.
- 10) Результаты, полученные после выхода из внутреннего цикла, добавляются в список `res`.
- 11)  $x\_last$  обновляется значением  $x$ , чтобы начать новую итерацию.

Этот метод позволяет находить корни уравнения  $f(x) = 0$  с помощью последовательного деления интервала пополам и итерационного уточнения найденных корней.

Далее рассмотрим функцию `gershgorin_rounds`. На рисунке 2 приведен код данной функции.

```
def gershgorin_rounds(a):
    left = -100000
    right = 100000
    n = len(a)
    for i in range(n):
        s = 0
        for j in range(n):
            if i != j:
                s += abs(a[i][j])
        b1 = a[i][i] - s
        b2 = a[i][i] + s
        if i == 0:
            left = b1
            right = b2
        elif b1 < left:
            left = b1
        elif b2 > right:
            right = b2
    return [left, right]
```

Рисунок 2 – Код функции `gershgorin_rounds`

Данная функция представляет метод Гершгорина для оценки спектра матрицы. Он используется для оценки интервалов, в которых расположены собственные значения матрицы. Функция `gershgorin_rounds` принимает матрицу  $A$  в качестве входного аргумента и вычисляет интервалы, которые содержат собственные значения этой матрицы. Знание границ кругов Гершгорина позволяет локализовать корни уравнения  $f(x) = 0$  для поиска собственных значений.

Алгоритм итеративно проходит по всем строкам матрицы  $A$ . Для каждой строки вычисляется сумма модулей всех элементов этой строки, кроме диагонального элемента. Затем вычисляются левая –  $b1$  и правая –  $b2$  границы интервала Гершгорина для данного диагонального элемента.

Начальные значения левой и правой границ интервала установлены как  $-100000$  и  $100000$  соответственно. Затем для каждой строки матрицы происходит обновление этих границ в соответствии с вычисленными значениями  $b1$  и  $b2$ .

Затем была реализована функция, реализующая метод Данилевского. На рисунках 3 – 5 приведен код данной функции.

```
def danilevsky_method(a):
    n = len(a)
    m = n - 1

    b = [[0] * n for i in range(n)]
    for i in range(n):
        b[i][i] = 1
    for j in range(n):
        if j != m - 1:
            b[m - 1][j] = -a[m][j] / a[m][m - 1]
    b[m - 1][m - 1] = 1 / a[m][m - 1]

    b_mul = copy.deepcopy(b)

    c = [[0] * n for i in range(n)]
    for i in range(n):
        c[i][m - 1] = a[i][m - 1] * b[m - 1][m - 1]
    for i in range(n - 1):
        for j in range(n):
            if j != m - 1:
                c[i][j] = a[i][j] + a[i][m - 1] * b[m - 1][j]
```

Рисунок 3 – Код функции, реализующей метод А. М. Данилевского

```

b_inv = [[0] * n for i in range(n)]
for i in range(n):
    b_inv[i][i] = 1
for j in range(n):
    b_inv[m - 1][j] = a[m][j]

d = [[0] * n for i in range(n)]
for i in range(m - 1):
    for j in range(n):
        d[i][j] = c[i][j]
for j in range(n):
    for k in range(n):
        d[m - 1][j] += a[m][k] * c[k][j]
d[m][m - 1] = 1

for k in range(2, n):
    b = [[0] * n for i in range(n)]
    for i in range(n):
        b[i][i] = 1
    for j in range(n):
        if j != m - k:
            b[m - k][j] = -d[m - k + 1][j] / d[m - k + 1][m - k]
        b[m - k][m - k] = 1 / d[m - k + 1][m - k]

```

Рисунок 4 – Код функции, реализующей метод А. М. Данилевского

```

b_mul = mult_matr_matr(b_mul, b)

b_inv = inv_matr(b)
d = mult_matr_matr(b_inv, d)
d = mult_matr_matr(d, b)

b = copy.deepcopy(b_mul)
p = d[0][:]
g = gershgorin_rounds(a)

ls = div_half_method(g[0], g[1], func(p))

vectors = []
for l in ls:
    y = [1]
    for i in range(1, n):
        y.append(l ** i)
    y = y[::-1]
    y = mult_matr_vec(b, y)
    norm = norm_vec(y)
    for i in range(n):
        y[i] /= norm
    vectors.append(y)
return ls, vectors

```

Рисунок 5 – Код функции, реализующей метод А. М. Данилевского

Функция `danilevsky_method` принимает матрицу  $A$  в качестве входных данных и выполняет серию операций для вычисления собственных значений и соответствующих собственных векторов входной матрицы.

Рассмотрим основные шаги, реализуемые данной функцией:

- 1) На первом этапе происходит инициализация переменных и матрицы, которые будут использоваться для вычислений.
- 2) Затем выполняются итеративные вычисления, чтобы преобразовать входную матрицу  $A$  в ее нормальную форму Фробениуса, используя метод Данилевского.
- 3) Вычисляются круги Гершгорина для преобразованной матрицы.
- 4) Далее используется метод половинного деления, чтобы найти корни (собственные значения) характеристического многочлена, который связан с исходной матрицей  $A$ .
- 5) Вычисляются собственные векторы, соответствующие собственным значениям, найденным на предыдущем шаге.

### **3.1.2. Нахождение собственных значений и собственных векторов матрицы методом А. Н. Крылова**

Также в ходе работы был реализован метод А. Н. Крылова для нахождения собственных значений и собственных векторов матрицы. На рисунках 6 – 7 приведён код данного метода.

```

def krylov_method(a):
    n = len(a)
    y = [[0] * n for _ in range(n + 1)]
    y[0][0] = 1
    for i in range(1, n + 1):
        y[i] = mult_matr_vec(a, y[i - 1])
    m = [[1] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            m[i][j] = y[n - 1 - j][i]
    p = gauss_method(m, y[n])
    g = gershgorin_rounds(a)
    ls = div_half_method(g[0], g[1], func(p))
    p = p[::-1]
    q = [[0] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            if j == 0:
                q[j][i] = 1
            else:
                q[j][i] = ls[i] * q[j - 1][i] - p[n - j]

```

Рисунок 6 – Код метода А. Н. Крылова

```

x = []
for i in range(n):
    xi = [0] * n
    for j in range(n):
        xi = sum_vec(xi, mult_vec_num(q[j][i], y[n - 1 - j]))
    x.append(xi)
vectors = []
for xi in x:
    norm = norm_vec(xi)
    for i in range(n):
        xi[i] /= norm
    vectors.append(xi)
return ls, vectors

```

Рисунок 7 – Код метода А. Н. Крылова

Данная функция представляет собой метод Крылова для поиска собственных значений и собственных векторов квадратной матрицы  $A$ .

Рассмотрим шаги, выполняемые данной функцией:

- 1) Создается матрица  $Y$ , заполненная нулями и инициализируется первый элемент как  $[1, 0, 0, \dots, 0]$ .
- 2) Затем в цикле вычисляются последующие степени матрицы  $A$ , умноженные на векторы из матрицы  $Y$ .
- 3) Создается матрица  $M$  размера  $n \times n$  и заполняется значениями из матрицы  $Y$  с учетом индексов.
- 4) Выполняется метод Гаусса для решения системы уравнений с матрицей  $M$  и вектором  $y[n]$ .
- 5) Вычисляются круги Гершгорина для матрицы  $A$ .
- 6) Применяется метод деления отрезка пополам для поиска собственных значений матрицы.
- 7) Вычисляется матрица  $Q$  размера  $n \times n$  с использованием коэффициентов  $ls$  и  $p$ .
- 8) Находятся собственные векторы матрицы через комбинацию векторов из матрицы  $Y$ , матрицы  $Q$  и коэффициентов  $ls$ .
- 9) Векторы нормируются и возвращаются вместе с найденными собственными значениями.

### **3.1.3. Реализация сингулярного разложения с использованием поиска собственных значений**

Метод сингулярного разложения матрицы с использованием собственных значений был реализован на языке программирования Python (листинги 7 – 8).

На рисунке 8 приведён код функции, реализующей нахождение сингулярного разложения матрицы методом использования собственных значений.

```
def svd_decomposition_eigenvalues_danilevsky(a):  
    a_transp_a = mult_matr_matr(transp_matr(a), a)  
  
    eigenvalues, eigenvectors = danilevsky_method(a_transp_a)  
  
    singular_values = [math.sqrt(eigenvalue) for eigenvalue in eigenvalues]  
  
    left_singular_vectors = mult_matr_matr(a, eigenvectors)  
    for i in range(len(left_singular_vectors)):  
        for j in range(len(left_singular_vectors[0])):  
            left_singular_vectors[i][j] /= singular_values[j]  
  
    right_singular_vectors = eigenvectors  
  
    return left_singular_vectors, singular_values, transp_matr(right_singular_vectors)
```

Рисунок 8 – Метод сингулярного разложения матрицы с использованием собственных значений

Данная функция выполняет сингулярное разложение для заданной матрицы методом использования собственных значений.

- 1) На первом этапе производится вычисление матрицы  $A^T \cdot A$ , выполняется это путём умножения транспонированной исходной матрицы на исходную матрицу.
- 2) Затем в данном методе происходит вычисление собственных значений и собственных векторов матрицы  $A^T \cdot A$  при помощи метода А. М. Данилевского.
- 3) Далее производится вычисление сингулярных значений, как квадратных корней из собственных значений.
- 4) Левые сингулярные векторы вычисляются, как произведение исходной матрицы на собственные векторы, разделенное на сингулярные значения.



5) Правые сингулярные векторы вычисляются на собственные векторы матрицы  $A^T \cdot A$ .

Наконец, функция возвращает левые сингулярные векторы, сингулярные значения и транспонированные правые сингулярные векторы в указанном порядке.

Данный метод был протестирован на примере матрицы 5x5. На рисунке 9 представлена исходная матрица.

```
Matrix:
0.19851120447696490867  0.31073635419645084799  2.84544794211424756369  3.34025608139010365960  0.65068172271983804045
0.31073635419645084799  2.35900151494284759934  1.91450662252381831330  1.32240200174514832554  4.05472827660109835790
2.84544794211424756369  1.91450662252381831330  3.64648903415644820569  1.77157333150020512846  2.72976453119596662589
3.34025608139010365960  1.32240200174514832554  1.77157333150020512846  2.48586200883318841903  3.29879873055611350097
0.65068172271983804045  4.05472827660109835790  2.72976453119596662589  3.29879873055611350097  0.60111500797979666721
```

Рисунок 9 – Исходная матрица A

На рисунке 10 представлена найденная матрица U.

```
U:
0.04935212919006312648  0.52421781722116844726  -0.41938284021712168848  -0.50040022088836733083  0.33132085354712464964
-0.42521134777134739213  0.67065842914749806969  0.58285252450267033275  0.36948198305229273952  0.40894677262357131964
-0.90856859543807722002  1.03967243320217606062  -0.56028580586159359989  0.39538543186496460935  0.44708081341472288850
0.41302055632132056440  0.58957831182243880530  -0.53784677484434539885  0.58505357076817765538  0.44096306594510148447
-1.65674914390236005524  -0.44735158749378312049  -0.24561870919270289981  -0.20145055094963315589  0.47948028255335356507
```

Рисунок 10 – Найденная матрица U

На рисунке 11 представлена найденная матрица  $\Sigma$ .

```
Sigma:
1.41659611880139912898  0.00000000000000000000  0.00000000000000000000  0.00000000000000000000  0.00000000000000000000
0.00000000000000000000  2.08877472537774133698  0.00000000000000000000  0.00000000000000000000  0.00000000000000000000
0.00000000000000000000  0.00000000000000000000  2.81139283326398281559  0.00000000000000000000  0.00000000000000000000
0.00000000000000000000  0.00000000000000000000  0.00000000000000000000  3.90062387389820353079  0.00000000000000000000
0.00000000000000000000  0.00000000000000000000  0.00000000000000000000  0.00000000000000000000  11.05238841704687224876
```

Рисунок 11 – Найденная матрица  $\Sigma$

На рисунке 12 представлена найденная матрица  $V^*$ .

```
V*:
0.05130408273686025422  -0.65816233031071325055  -0.50790163227294771175  0.44867470444750773284  0.32390921596289895534
0.01417914994708021660  -0.52024339101696903676  0.68574929957094665767  -0.29021417120961662262  0.41793468175282616484
-0.75746033508998689143  0.21908842655439544900  -0.23296618264575080781  -0.23121050910300605663  0.52011783250545984281
0.63470616993763251390  0.22596341098108777001  -0.32735009712645463109  -0.45756577405147330628  0.47912847072633724110
0.14341813487064233867  0.44397067393111439015  0.33219347032595147873  0.67204315881873355476  0.46938978520173169073
```

Рисунок 12 – Найденная матрица  $V^*$

Для проверки результата матрицы  $U$ ,  $\Sigma$ ,  $V^*$  были перемножены. На рисунке 13 представлены результат после перемножения матриц.

Result after multiplication				
0.19851120234125474440	0.31073635646678621214	2.84544794233215636581	3.34025608146020669409	0.65068172290237780153
0.31073635408031041738	2.35900151628424081807	1.91450662454522069211	1.32240200169004840092	4.05472827657129997192
2.84544794012089274915	1.91450662650011516064	3.64648903345823871547	1.77157333166375074995	2.72976453163729626539
3.34025608092826065132	1.32240200428683118972	1.77157332992358895751	2.48586200899765419337	3.29879873097052955799
0.65068172229388065997	4.05472827897553056431	2.72976453413995523434	3.29879873048881311348	0.60111500789729443994

Рисунок 13 – Результат перемножения матриц  $U$ ,  $\Sigma$ ,  $V^*$

Погрешность вычислений была рассчитана при помощи матричной нормы Фробениуса. На рисунке 14 представлен результат вычисления погрешности.

Error rate  
7.749410512429923e-09

Рисунок 14 – Погрешность метода сингулярного разложения матрицы с использованием собственных значений для матрицы  $5 \times 5$ , вычисленная при помощи матричной нормы Фробениуса

### 3.2. Сингулярное разложение матрицы с использованием QR-разложения

Метод использования QR-разложения часто применяется для нахождения SVD прямоугольных матриц. Он включает вычисление QR-разложения исходной матрицы и применение его компонент к нахождению левых и правых сингулярных векторов.

#### 3.2.1 Нахождение QR-разложения матрицы

QR-разложение матрицы – это разложение произвольной матрицы  $A$  на произведение ортогональной матрицы  $Q$  и верхнетреугольной матрицы  $R$ .

Процесс QR-разложения может быть выполнен с использованием различных методов, включая методы ортогонализации, такие как метод Грама-Шмидта или метод вращений.

Метод нахождения QR-разложения матрицы был реализован на языке программирования Python (листинг 6).

Для реализации QR-разложения матрицы была написана функция, представленная на рисунке 15.

```
def qr_decomposition(matrix):  
    q = []  
    r = [[0] * len(matrix[0]) for _ in range(len(matrix[0]))]  
  
    for j in range(len(matrix[0])):  
        v = matrix[j]  
        for i in range(len(q)):  
            rij = scalar_mult_vec(q[i], matrix[j])  
            r[i][j] = rij  
            v = sub_vec(v, mult_vec_num(rij, q[i]))  
        r[j][j] = math.sqrt(scalar_mult_vec(v, v))  
        q.append(mult_vec_num(1 / r[j][j], v))  
  
    return q, r
```

Рисунок 15 – Код функции, реализующей QR-разложение матрицы

Данная функция представляет собой реализацию QR-разложения матрицы.

Рассмотрим подробнее шаги, реализуемые данной функцией:

- 1) На первом этапе создаётся пустой список для Q и инициализируется R как матрица из нулей с теми же размерами, что и входная матрица.
- 2) Выполняется итерация по каждому столбцу матрицы.
- 3) Для каждого столбца вычисляется соответствующий вектор и выполняется процесс Грама-Шмидта, чтобы ортогонализировать этот вектор относительно ранее вычисленных векторов.

- 4) Обновляются элементы матрицы R на основе скалярных произведений ортогональных векторов.
- 5) Нормализуется ортогонализированный вектор, чтобы получить следующий столбец Q и сохранить его в списке Q.
- 6) Эти шаги повторяются для каждого столбца матрицы.

Данная функция вычисляет QR-разложение входной матрицы с использованием процесса Грама-Шмидта, в результате чего получается ортогональная матрица Q и верхнетреугольная матрица R.

### 3.2.2 Реализация сингулярного разложения с использованием QR-разложения

Для нахождения сингулярного разложения матрицы методом использования QR-разложения была написана программа на языке программирования Python (листинг 9).

Код, реализующий нахождение сингулярного разложения матрицы методом использования QR-разложения, представлен на рисунке 16.

```
def svd_decomposition_qr(matrix, num_iterations=100):  
    m, n = len(matrix), len(matrix[0])  
  
    u = [[random.random() for _ in range(m)] for _ in range(m)]  
    v = [[random.random() for _ in range(n)] for _ in range(n)]  
  
    for _ in range(num_iterations):  
        u, _ = qr_decomposition(mult_matr_matr(matrix, v))  
        v, _ = qr_decomposition(mult_matr_matr(transp_matr(matrix), u))  
    sigma = mult_matr_matr(transp_matr(u), mult_matr_matr(matrix, v))  
    return u, sigma, transp_matr(v)
```

Рисунок 16 – Функция нахождения сингулярного разложения матрицы методом использования QR-разложения

- 1) На первом этапе определяются размеры матрицы `matrix` как `m` и `n`.
- 2) Затем инициализируются матрицы `u` и `v` случайными значениями размерности  $m \times m$  и  $n \times n$  соответственно.
- 3) Далее запускается цикл с числом итераций, равным `num_iterations`.

Внутри цикла:

Матрица `u` обновляется путем выполнения QR-разложения матрицы `matrix`, умноженной на `v`.

Матрица `v` обновляется путем выполнения QR-разложения матрицы `matrix.T`, умноженной на `u`.

- 4) После завершения цикла вычисляется матрица сингулярных значений `sigma` как произведение `u.T`, `matrix` и `v`.
- 5) Наконец, функция возвращает матрицы `u`, `sigma` и `v.T` (транспонированную матрицу `v`).

Этот метод основан на итеративном применении QR разложения для оценки матриц `u` и `v`, приближенно аппроксимирующих исходную матрицу `matrix` в ее сингулярное разложение.

Данный метод был протестирован на примере матрицы  $5 \times 5$ . На рисунке 17 представлена исходная матрица.

```
Matrix:
3.68784735539730634812  4.21833162060294242224  0.72876103874450426368  3.37114654801294255648  2.70183379722728211902
2.74130712482649041561  4.99519345371021650237  0.97276876415571755796  1.23907989106928617673  3.62729583461322180327
3.01738920821956835283  3.67413247660568931252  1.41461027774562975168  1.07029573022172597163  1.36464377130436842478
4.97576819640272383083  3.02316645006128048578  0.61522539532175735122  0.06624643485016568079  2.24607159386724575256
3.76669497560772770228  4.41806783441744155283  4.51700304128199814357  4.05618822546216151181  0.03251462876062871654
```

Рисунок 17 – Исходная матрица

На рисунке 18 представлена найденная матрица `U`.

```
U:
-0.73026646281538676320  0.12311486184664820198  -0.39501938310161660572  -0.14803162390543794791  0.52306782483490732449
-0.18529503442656702905  -0.00480049375360879719  0.79108325481213181973  0.37529994374876418650  0.44607167770953237707
0.29355338081061310707  -0.79658984122974962894  -0.08681366075435305307  -0.23519548731632888838  0.46520695235205178353
-0.57311930785804132693  -0.58466843759948938786  0.09896574920570443135  0.14934831587645686080  -0.54552537802041023429
0.13318151177599626966  -0.09179939529613664884  -0.44812513846072554724  0.87155880687248699079  0.11577850572056265499
```

Рисунок 18 – Найденная матрица `U`

На рисунке 19 представлена найденная матрица  $\Sigma$ .

Sigma:				
3.77277033776688464073	-0.95657482942379479240	-1.07063239736803161861	-4.74155857218809728693	-3.94064296656308377109
2.01651730845852705798	0.47742693768614768990	-0.34523215575173238356	-5.63062129997050409003	-4.54946091938026508927
-0.04196628604232255366	0.34295436696539643995	3.25156063790483296216	0.07618212376477417891	-1.08574657397805252046
-0.76693964996949848256	1.15021092044380335295	-2.66254766682502541641	5.41226877598692368565	5.03788765918659642296
-1.57113445717361832621	2.87601915097971483348	-0.60003990588683087104	4.10165173511259073535	4.64799214558138196196

Рисунок 19 – Найденная матрица  $\Sigma$

На рисунке 20 представлена найденная матрица  $V^*$ .

V*:				
-0.56096844984707017190	0.24969575962099116451	0.52056210425123450314	-0.45445080065604648301	-0.38138693121825145704
-0.62738446438527151194	0.26315534453030420270	0.02796044850703234899	0.50975903598757765778	0.52583451488224475234
-0.08965676652111716216	0.33734919219126324741	-0.54586761288781959767	-0.65522268281256168532	0.38841855243145229082
0.45682557673622625760	0.23256667964857979936	0.64428853451721634915	-0.15604787817051316012	0.54567799659230142861
0.27381597612406294306	0.83696448808630463656	-0.12310216255867957591	0.28275871455798434750	-0.35973410154799984051

Рисунок 20 – Найденная матрица  $V^*$

Для проверки результата матрицы  $U$ ,  $\Sigma$ ,  $V^*$  были перемножены. На рисунке 21 представлены результат после перемножения матриц.

Result after multiplication				
3.68784735539730412768	4.21833162060295574491	0.72876103874450659514	3.37114654801294122421	2.70183379722729277717
2.74130712482648331019	4.99519345371023160141	0.97276876415571411627	1.23907989106927862721	3.62729583461324178728
3.01738920821956835283	3.67413247660570085884	1.41461027774563263826	1.07029573022172774799	1.36464377130438374586
4.97576819640272205447	3.02316645006129025575	0.61522539532175535282	0.06624643485016379341	2.24607159386725463435
3.76669497560772548184	4.41806783441744066465	4.51700304128199814357	4.05618822546216417635	0.03251462876063415663

Рисунок 21 – Результат перемножения матриц  $U$ ,  $\Sigma$ ,  $V^*$

Погрешность вычислений была рассчитана при помощи матричной нормы Фробениуса. На рисунке 22 представлен результат вычисления погрешности.

Error rate  
4.070508228469245e-14

Рисунок 22 – Погрешность метода сингулярного разложения матрицы с использованием QR-разложения для матрицы  $5 \times 5$ , вычисленная при помощи матричной нормы Фробениуса

### 3.3. Итерационные методы

Далее была написана программа, реализующая SVD разложение с использованием метода степенных итераций для нахождения собственных значений и процесса Грама-Шмидта. (Листинг 10)

Функции, реализующие разложение, представлены на рисунках 23 – 26.

```
def power_iteration(matrix, num_iterations):  
    b_k = np.random.rand(matrix.shape[1])  
    for _ in range(num_iterations):  
        b_k1 = np.dot(matrix, b_k)  
        b_k1_norm = np.linalg.norm(b_k1)  
        b_k = b_k1 / b_k1_norm  
    return b_k
```

Рисунок 23 – Функция power\_iteration

```
def find_eigvals_and_vecs(matrix, num_eigenvalues, num_iterations):  
    eigenvalues = []  
    eigenvectors = []  
    matrix1 = copy.deepcopy(matrix)  
    for _ in range(num_eigenvalues):  
        eigenvector = power_iteration(matrix1, num_iterations)  
        eigenvalue = np.dot(np.dot(eigenvector.T, matrix1), eigenvector)  
        np.dot(eigenvector.T, eigenvector)  
        eigenvalues.append(eigenvalue)  
        eigenvectors.append(eigenvector)  
        matrix1 -= eigenvalue * np.outer(eigenvector, eigenvector)  
    return np.array(eigenvalues), np.array(eigenvectors).T
```

Рисунок 24 – Функция find\_eigvals\_and\_vecs



```
def gram_schmidt_process(vectors):
    orthogonal_vectors = []
    for v in vectors.T:
        for u in orthogonal_vectors:
            v -= np.dot(u, v) * u
        v_norm = np.linalg.norm(v)
        if v_norm > 1e-6:
            v /= v_norm
            orthogonal_vectors.append(v)
    return np.array(orthogonal_vectors).T
```

Рисунок 25 – Функция gram\_schmidt\_process

```
def svd_decomposition_iterations(matrix):
    n = matrix.shape[0]

    eigvals_A_A_T, eigvecs_A_A_T = find_eigvals_and_vecs(np.dot(matrix, matrix.T), n, 100)
    eigvals_A_T_A, eigvecs_A_T_A = find_eigvals_and_vecs(np.dot(matrix.T, matrix), n, 100)

    sing_vals = np.sqrt(eigvals_A_T_A)
    u = gram_schmidt_process(eigvecs_A_A_T.T).T
    s = np.zeros((n, n))
    np.fill_diagonal(s, sing_vals)
    vt = np.zeros((n, n))
    for i, sing_val in enumerate(sing_vals):
        vt[i] = 1 / sing_val * np.dot(matrix.T, eigvecs_A_A_T.T[i])
    return u, s, vt
```

Рисунок 26 – Функция svd\_decomposition\_iterations

Рассмотрим подробнее каждую функцию данного метода.

- 1) В функции power\_iteration реализуется метод степенной итерации для нахождения доминирующего собственного вектора матрицы. Для этого инициализируется случайный вектор  $b_k$ , затем матрица итеративно умножается на  $b_k$  и результат нормализуется.
- 2) В функции find\_eigvals\_and\_vecs происходит поиск нескольких собственных значений и соответствующих им собственных векторов матрицы. Для этого используется метод степенной итерации



итеративно наряду с некоторыми матричными операциями для нахождения собственных значений и собственных векторов.

- 3) В функции `gram_schmidt_process` реализуется процесс Грама-Шмидта, метод ортонормирования набора векторов.
- 4) В функции `svd_decomposition_iterations` происходит вычисление разложения по сингулярным значениям матрицы с использованием ранее определенных функций. Для этого вычисляются собственные значения и векторы матриц  $A \cdot A^T$  и  $A^T \cdot A$ , из которых выводятся сингулярные значения и, в конечном счете, компоненты SVD  $U$ ,  $S$  и  $V^T$ .

Данный метод был протестирован на примере матрицы  $5 \times 5$ . На рисунке 27 представлена исходная матрица.

```
Matrix:
8.74067582715810154070  7.76064385026553615887  2.47777661122614567546  5.89111574481793098812  9.17180764796419012441
0.94287064750128712909  5.20872235870540656322  1.54560293723374986286  7.95613433724728924545  5.74988696457673675866
6.78317806152440194722  6.73153162500896762310  0.96869624683752930672  4.98089435641231226271  3.29921385882991469174
4.37118311955635530097  6.31870291817081319863  7.07902389290884581641  0.05132216316836801795  3.45598415738197939362
0.01540790911304212862  0.77662172025350417748  5.12209422991482021814  9.78646272112751525185  5.79214646797368892805
```

Рисунок 27 – Исходная матрица

На рисунке 28 представлена найденная матрица  $U$ .

```
U:
0.62373194670575449194  -0.27377979879923003415  0.28370115964408321174  -0.63107147388101914043  -0.23930216734986625715
0.41112436183629186282  0.36043720203403284419  0.17691331306642002485  0.55690521642825618898  -0.59968328480076860121
0.41783387502777286082  -0.26838112104442718442  0.36456800673491840392  0.44748141528508822429  0.64825687669780640565
0.34185082860367277391  -0.43503863673009229851  -0.80538319193871965584  0.19592837812433017142  -0.08276098309695352484
0.38792793354643767545  0.73064533364573702734  -0.32659365315999838719  -0.23016912996459656937  0.39500388874420933050
```

Рисунок 28 – Найденная матрица  $U$

На рисунке 29 представлена найденная матрица  $\Sigma$ .

```
Sigma:
25.17319170539557404709  0.000000000000000000  0.000000000000000000  0.000000000000000000  0.000000000000000000
0.000000000000000000  10.14210250434600446567  0.000000000000000000  0.000000000000000000  0.000000000000000000
0.000000000000000000  0.000000000000000000  6.47312015810142948880  0.000000000000000000  0.000000000000000000
0.000000000000000000  0.000000000000000000  0.000000000000000000  3.02173738035161587590  0.000000000000000000
0.000000000000000000  0.000000000000000000  0.000000000000000000  0.000000000000000000  2.44795421734575668893
```

Рисунок 29 – Найденная матрица  $\Sigma$

На рисунке 30 представлена найденная матрица  $V^*$ .

```

V*:
0.40415960223494656889  0.48686647790499615329  0.27778087309539961636  0.51009069760913083869  0.51211455297488017724
-0.56832669256930679058  -0.41760089372964431798  0.02775885720864695524  0.69474218961984490761  0.13848104519438245164
0.24624390801731246836  0.03625457281921962149  -0.93380451518386464738  0.25601356390854868028  0.02270960658241606700
-0.36491029596234847432  0.68660429614668772036  -0.02031696568346335122  0.23147912783905885692  -0.58431126182358439358
0.56556538092387076411  -0.34033905551844556570  0.22285327172037022736  0.37150402238007296241  -0.61369887390169464148

```

Рисунок 30 – Найденная матрица  $V^*$

Для проверки результата матрицы  $U$ ,  $\Sigma$ ,  $V^*$  были перемножены. На рисунке 31 представлены результат после перемножения матриц.

```

Result after multiplication
8.74067582715809976435  7.76064385026553438252  2.47777661122614389910  5.89111574481792921176  9.17180764796418834806
0.94287064750128768420  5.20872235870540745140  1.54560293723375030694  7.95613433724729190999  5.74988696457673675866
6.78317806152440461176  6.73153162500896939946  0.96869624683752819649  4.98089435641231226271  3.29921385882991380356
4.37118311955635618915  6.31870291817081319863  7.07902389290884848094  0.05132216316836815673  3.45598415738197983771
0.01540790911304135147  0.77662172025350351134  5.12209422991482021814  9.78646272112751525185  5.79214646797368892805

```

Рисунок 31 – Результат перемножения матриц  $U$ ,  $\Sigma$ ,  $V^*$

Погрешность вычислений была рассчитана при помощи матричной нормы Фробениуса.

На рисунке 32 представлен результат вычисления погрешности.

```

Error rate
6.752823741165281e-15

```

Рисунок 32 – Погрешность итерационного метода сингулярного разложения для матрицы  $5 \times 5$ , вычисленная при помощи матричной нормы Фробениуса

#### 4. Методы сжатия изображений

Сжатие изображений – это процесс уменьшения размера файла изображения с минимальной потерей качества. Существует несколько методов сжатия изображений, включая методы, основанные на сингулярном разложении матрицы.

Методы сжатия изображений играют важную роль в передаче, хранении и обработке графических данных. Одним из таких методов является сингулярное разложение матриц. SVD используется для сжатия

изображений путем аппроксимации изображения с использованием только наиболее значимых сингулярных значений.

Процесс сжатия изображения с помощью SVD включает следующие шаги:

- исходное изображение представляется в виде матрицы;
- эта матрица подвергается сингулярному разложению на три подматрицы:  $U$ ,  $\Sigma$  и  $V$ ;
- затем, для сокращения размерности, матрицы  $U$ ,  $\Sigma$  и  $V$  усекаются до содержания только наиболее значимой информации, что позволяет удалить лишние данные;
- далее происходит восстановление изображения из усеченного SVD разложения.

При использовании сингулярного разложения для сжатия изображений важно понимать баланс между степенью сжатия и потерей информации. Чем больше сингулярных значений удаляется, тем сильнее сжимается изображение, однако возможна потеря деталей из-за удаления части информации.

Кроме SVD, также существуют другие методы сжатия изображений, такие как преобразование вейвлетов, кодирование JPEG и PNG, а также алгоритмы сжатия потерь и без потерь. Каждый метод имеет свои преимущества и ограничения, и оптимальный выбор зависит от конкретных потребностей и характеристик изображений.

Метод сингулярного разложения используется для сжатия изображений путем декомпозиции матрицы изображения. SVD позволяет разложить матрицу изображения на три более простые матрицы, что позволяет эффективно сжимать данные, убирая ненужную информацию, сохраняя при этом основные характеристики изображения.

Кроме метода SVD, другие методы сжатия изображений включают в себя:

- 1) Run Length Encoding (RLE) – один из самых старых и самых простых алгоритмов архивации графики. Изображение в нем вытягивается в цепочку байт по строкам раstra. Само сжатие в RLE происходит за счет того, что в исходном изображении встречаются цепочки одинаковых байт. Замена их на пары <счетчик повторений, значение> уменьшает избыточность данных.
- 2) Классический алгоритм Хаффмана. Один из классических алгоритмов, известных с 60-х годов. Использует только частоту появления одинаковых байт в изображении. Сопоставляет символам входного потока, которые встречаются большее число раз, цепочку бит меньшей длины. И, напротив, встречающимся редко – цепочку большей длины. Для сбора статистики требует двух проходов по изображению.
- 3) LZW (Lempel-Ziv-Welch). Существует довольно большое семейство LZ-подобных алгоритмов, различающихся, например, методом поиска повторяющихся цепочек. Один из достаточно простых вариантов этого алгоритма, например, предполагает, что во входном потоке идет либо пара <счетчик, смещение относительно текущей позиции>, либо просто <счетчик> «пропускаемых» байт и сами значения байтов (как во втором варианте алгоритма RLE). При разархивации для пары <счетчик, смещение> копируются <счетчик> байт из выходного массива, полученного в результате разархивации, на <смещение> байт раньше, а <счетчик> (т.е. число равное счетчику) значений «пропускаемых» байт просто копируются в выходной массив из входного потока.
- 4) JPEG – один из самых новых и достаточно мощных алгоритмов. Опиерирует алгоритм областями 8x8, на которых яркость и цвет

меняются сравнительно плавно. Вследствие этого, при разложении матрицы такой области в двойной ряд по косинусам значимыми оказываются только первые коэффициенты. Таким образом, сжатие в JPEG осуществляется за счет плавности изменения цветов в изображении.

- 5) Фрактальный алгоритм. Фрактальная архивация основана на том, что мы представляем изображение в более компактной форме – с помощью коэффициентов системы итерируемых функций (Iterated Function System).

При выборе метода сжатия изображения важно учитывать баланс между уровнем сжатия и сохранением качества изображения в зависимости от конкретных потребностей и ограничений проекта.

## **5. Применение сингулярного разложения матрицы для задачи сжатия изображений**

Методы сжатия изображений представляют собой важный инструмент для уменьшения размера файла изображения без значительной потери качества. Существует несколько методов сжатия изображений, включая сжатие с потерей и без потери информации. Одним из наиболее эффективных методов сжатия изображений является метод с использованием сингулярного (SVD) разложения матрицы.

SVD-разложение матрицы является мощным инструментом линейной алгебры, который позволяет разложить матрицу на комбинацию трех более простых матриц. Для изображений SVD-разложение может быть использовано для сжатия данных путем представления изображения в виде набора сингулярных значений, левых сингулярных векторов и правых сингулярных векторов. Эти данные могут быть использованы для восстановления изображения с минимальной потерей качества.

Процесс сжатия с использованием SVD включает в себя следующие шаги:

- выполнение SVD-разложения матрицы изображения;
- выбор сингулярных чисел для достижения определённого процента отношения суммы выбранных сингулярных чисел к сумме всех сингулярных чисел;
- хранение только выбранных сингулярных чисел и соответствующих сингулярных векторов;
- восстановление сжатого изображения из выбранных сингулярных чисел и векторов.

Этот метод сжатия обеспечивает высокую степень компрессии при минимальной потере информации. Однако, процесс SVD-сжатия изображений может быть вычислительно затратным, поэтому его применение часто зависит от специфических требований конкретного приложения.

В ходе работы был написан код, реализующий сжатие изображения при помощи сингулярного разложения матрицы (листинг 11).

На рисунках 33 – 35 приведён код, реализующий сжатие изображения при помощи сингулярного разложения матрицы.

```
def zip_img_by_svd(img, plotId, imgfile, imgfile_size, rate=0.8):
    zip_img = np.zeros(img.shape)

    zip_rate_singular = 0

    for chanel in range(3):
        u, sigma, v = svd_decomposition_iterations(img[:, :, chanel])
        sigma_i = 0
        temp = 0

        while (temp / np.sum(sigma)) < rate:
            temp += sigma[sigma_i]
            sigma_i += 1

        zip_rate_singular += temp / np.sum(sigma)

        SigmaMat = np.zeros((sigma_i, sigma_i))
        for i in range(sigma_i):
            SigmaMat[i][i] = sigma[i]
        zip_img[:, :, chanel] = u[:, 0:sigma_i].dot(SigmaMat).dot(v[0:sigma_i, :])

    zip_rate_singular /= 3
    zip_rate_singular = 1 - zip_rate_singular
```

Рисунок 33 – Применение сингулярного разложения матрицы к задаче сжатия изображения

```
for i in range(3):
    MAX = np.max(zip_img[:, :, i])
    MIN = np.min(zip_img[:, :, i])
    zip_img[:, :, i] = (zip_img[:, :, i] - MIN) / (MAX - MIN)

zip_img = np.round(zip_img * 255).astype("uint8")
plt.imsave("zip_" + imgfile + str(plotId - 1) + ".jpg", zip_img)
size = os.path.getsize("zip_" + imgfile + str(plotId - 1) + ".jpg")

size_rate = 1 - size / imgfile_size
print(plotId - 1)
print("Изображение с " + str(round(zip_rate_singular * 100, 2)) + "% сжатия по сингулярным числам, " +
      str(round(size_rate * 100, 2)) + "% сжатия по памяти, размер " + str(round(size / 1024)) + " КБ")
print()

f = plt.subplot(3, 3, plotId)
plt.subplots_adjust(hspace=0.5, wspace=0.9)
f.imshow(zip_img)
f.axis('off')
f.set_title(str(round(zip_rate_singular * 100, 2)) + "% сжатия по сингулярным числам\n" +
            str(round(size_rate * 100, 2)) + "% сжатия по памяти\nРазмер " + str(round(size / 1024)) + " КБ")
```

Рисунок 34 – Применение сингулярного разложения матрицы к задаче сжатия изображения

```

if __name__ == '__main__':
    imgfile = "img1.jpg"
    imgfile_size = os.path.getsize(imgfile)
    plt.figure(figsize=(12, 12))
    plt.rcParams['font.sans-serif'] = 'SimHei'
    img = plt.imread(imgfile)
    f1 = plt.subplot(331)
    plt.subplots_adjust(hspace=0.5, wspace=0.9)
    f1.imshow(img)
    f1.axis('off')
    f1.set_title("Исходное изображение\nРазмер " + str(round(imgfile_size / 1024)) + " КБ")
    for i in range(8):
        rate = (i + 1) / 10.0
        zip_img_by_svd(img, i + 2, imgfile[:4], imgfile_size, rate)
    plt.show()

```

Рисунок 35 – Применение сингулярного разложения матрицы к задаче сжатия изображения

Этот код реализует алгоритм сжатия изображений с использованием сингулярного разложения. Он отображает исходное изображение и восемь последующих изображений, каждое из которых сжато с разными степенями сжатия.

Разберём основные шаги выполнения алгоритма:

- загрузка библиотек и изображения;
- определение функции `zip_img_by_svd`, в которой выполняется сжатие изображения с использованием SVD;
- циклический вызов `zip_img_by_svd` с разными степенями сжатия для создания последующих изображений;
- отображение исходного изображения и всех созданных сжатых изображений на одном графике.

Разберём основные шаги кода более подробно.

- 1) В функции `zip_img_by_svd` происходит сжатие входного изображения с использованием SVD. В качестве входных данных принимаются данные изображения, идентификатор участка и степень сжатия.



- 2) В рамках функции SVD выполняется для каждого цветового канала входного изображения отдельно.
- 3) Затем выполняется итерация по сингулярным значениям SVD до тех пор, пока совокупная сумма сингулярных значений не достигнет определенного процента от общей суммы, как определено параметром `rate`.
- 4) Используя выбранные сингулярные значения, сжатое изображение восстанавливается с использованием усеченных компонентов SVD.
- 5) Восстановленное изображение затем нормализуется и сохраняется в формате JPEG с помощью `plt.imsave`.
- 6) Вычисление степени сжатия достигается путем сравнения размеров исходного и сжатого изображений.
- 7) Наконец, в основном блоке кода считывается файл изображения, настраивается фигура для построения графика, отображается исходное изображение и вызывается функция `zip_img_by_svd` для различных степеней сжатия.
- 8) Полученные сжатые изображения отображаются в таблице вспомогательных графиков для сравнения.

Этот код демонстрирует применение SVD для сжатия изображений и сравнивает влияние различных степеней сжатия на входное изображение.

## **6. Результаты сжатия изображений при помощи сингулярного разложения матрицы**

На рисунках 36 – 43 представлены результаты сжатия изображений при помощи сингулярного разложения матрицы с разными степенями сжатия.

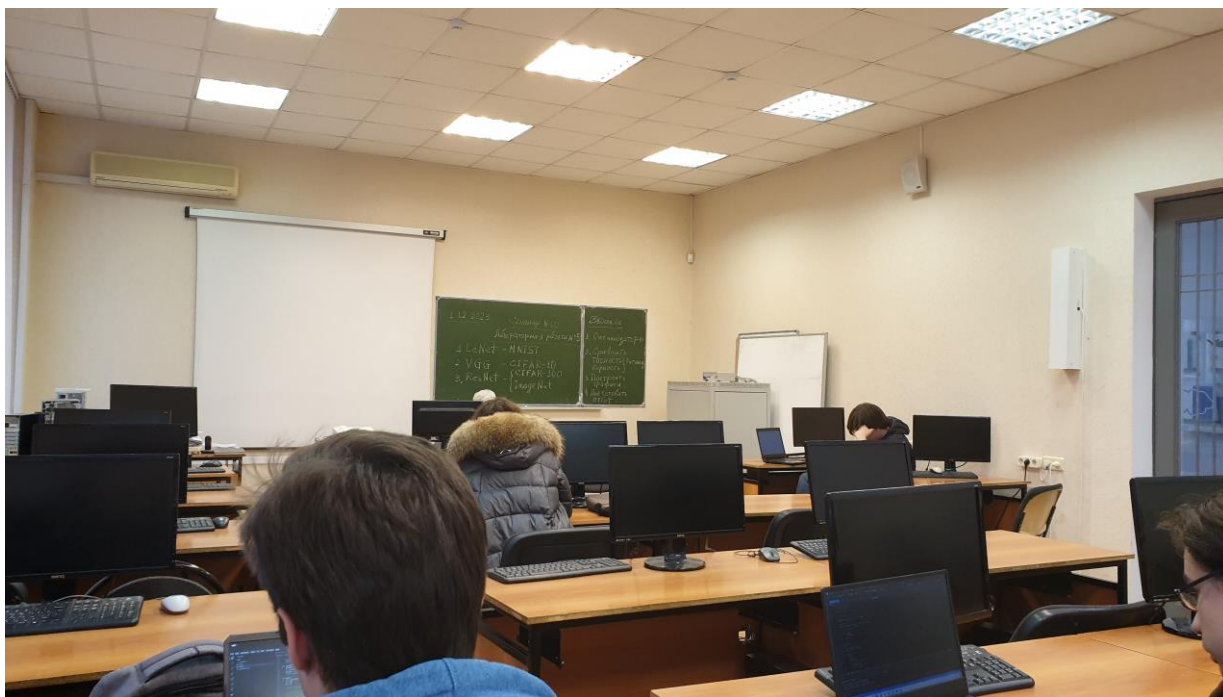


Рисунок 36 – Исходное изображение, размер 2419 КБ

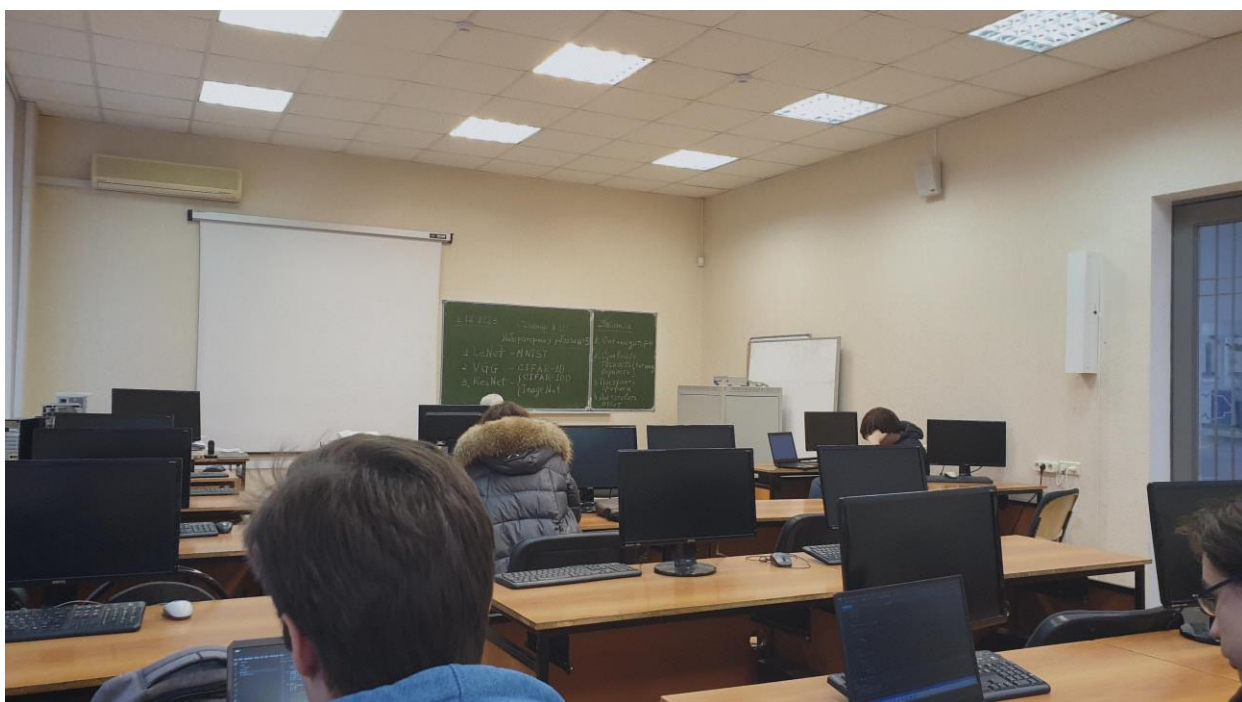


Рисунок 37 – Изображение с 19.97% сжатия по сингулярным числам,  
74.1% сжатия по памяти, размер 626 КБ



Рисунок 38 – Изображение с 29.97% сжатия по сингулярным числам,  
75.17% сжатия по памяти, размер 600 КБ



Рисунок 39 – Изображение с 39.91% сжатия по сингулярным числам,  
78.49% сжатия по памяти, размер 520 КБ



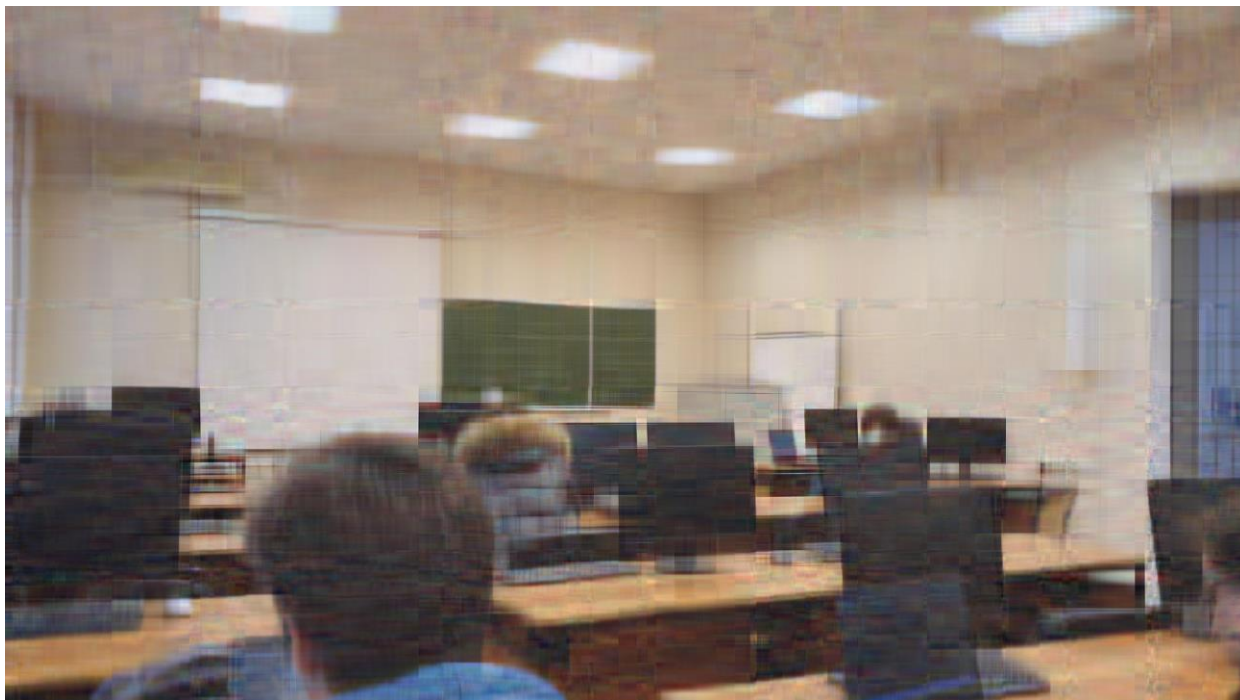


Рисунок 40 – Изображение с 49.91% сжатия по сингулярным числам,  
81.42% сжатия по памяти, размер 449 КБ

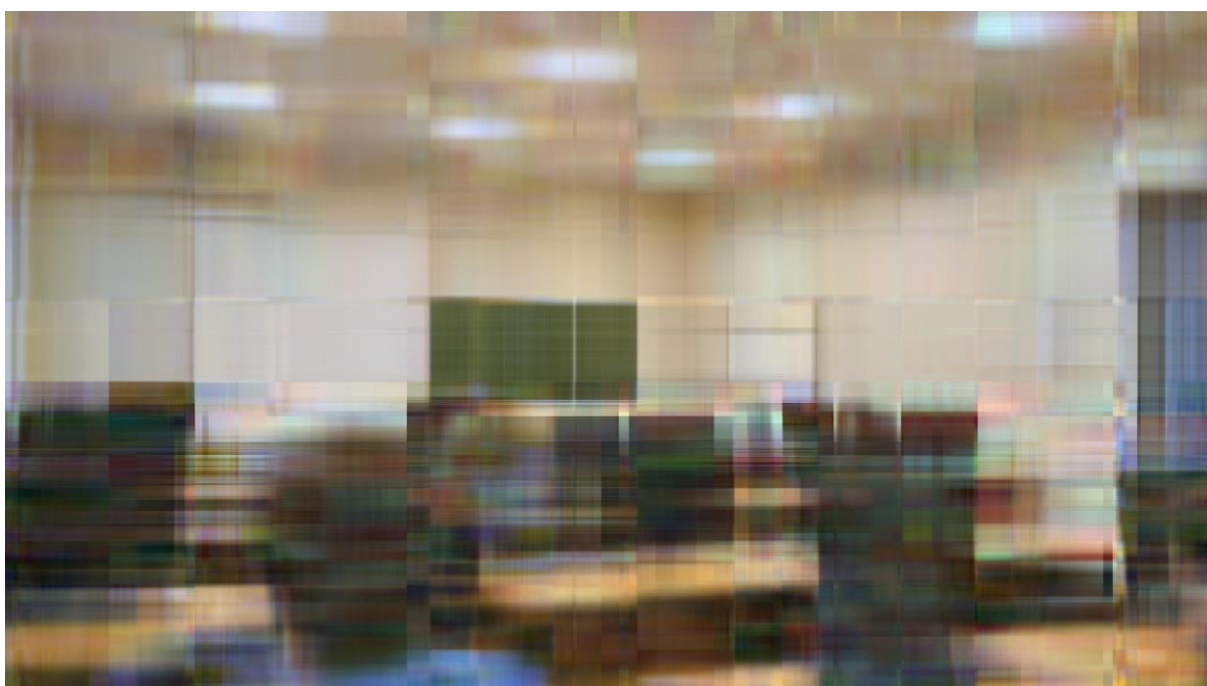


Рисунок 41 – Изображение с 59.39% сжатия по сингулярным числам,  
84.08% сжатия по памяти, размер 385 КБ

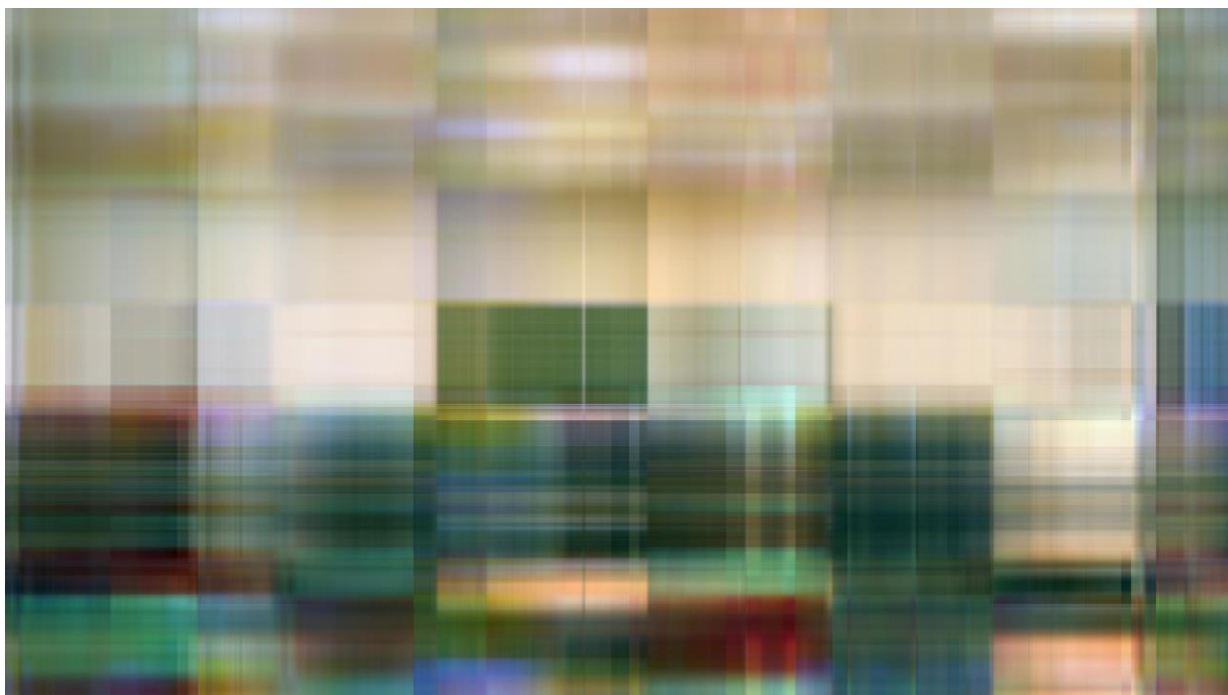


Рисунок 42 – Изображение с 68.8% сжатия по сингулярным числам, 85.8% сжатия по памяти, размер 343 КБ

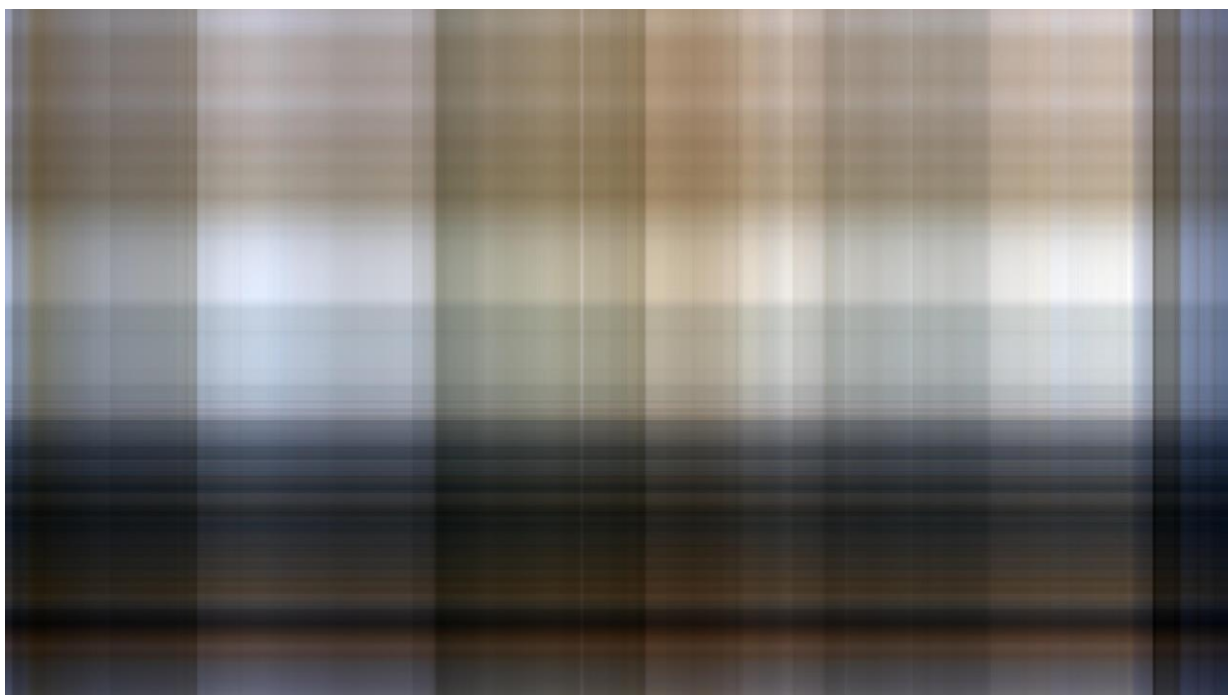
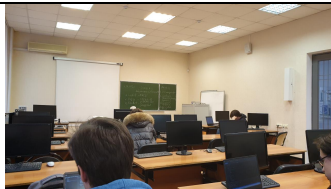
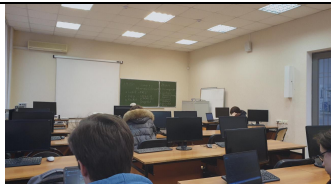
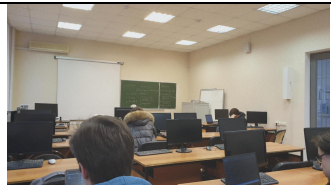


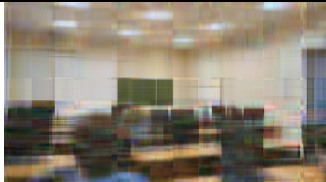

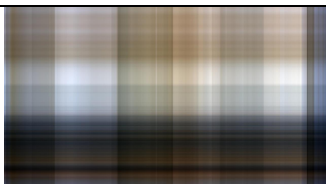


Рисунок 43 – Изображение с 77.73% сжатия по сингулярным числам, 88.04% сжатия по памяти, размер 289 КБ

Была вычислена степень сжатия изображения по сингулярным числам и по памяти, а также размер полученного изображения. Результаты представлены в таблице 1.

Таблица 1 – Результаты сжатия изображения

Изображение	Степень сжатия по сингулярным числам	Степень сжатия по памяти	Размер изображения
 <p>Рисунок 36</p>	—	—	2419 КБ
 <p>Рисунок 37</p>	19.97%	74.1%	626 КБ
 <p>Рисунок 38</p>	29.97%	75.17%	600 КБ
 <p>Рисунок 39</p>	39.91%	78.49%	520 КБ

 <p>Рисунок 40</p>	49.91%	81.42%	449 КБ
 <p>Рисунок 41</p>	59.39%	84.08%	385 КБ
 <p>Рисунок 42</p>	68.8%	85.8%	343 КБ
 <p>Рисунок 43</p>	77.73%	88.04%	289 КБ

Также была протестирована работа программы для второго изображения. На рисунках 44 – 52 представлены результаты сжатия изображений при помощи сингулярного разложения матрицы с разными степенями сжатия.





Рисунок 44 – Исходное изображение, размер 3629 КБ



Рисунок 45 – Изображение с 19.99% сжатия по сингулярным числам,  
74.76% сжатия по памяти, размер 916 КБ





Рисунок 46 – Изображение с 29.98% сжатия по сингулярным числам,  
78.57% сжатия по памяти, размер 778 КБ



Рисунок 47 – Изображение с 39.95% сжатия по сингулярным числам,  
81.52% сжатия по памяти, размер 671 КБ



Рисунок 48 – Изображение с 49.88% сжатия по сингулярным числам,  
83.75% сжатия по памяти, размер 590 КБ



Рисунок 49 – Изображение с 59.92% сжатия по сингулярным числам,  
86.15% сжатия по памяти, размер 503 КБ



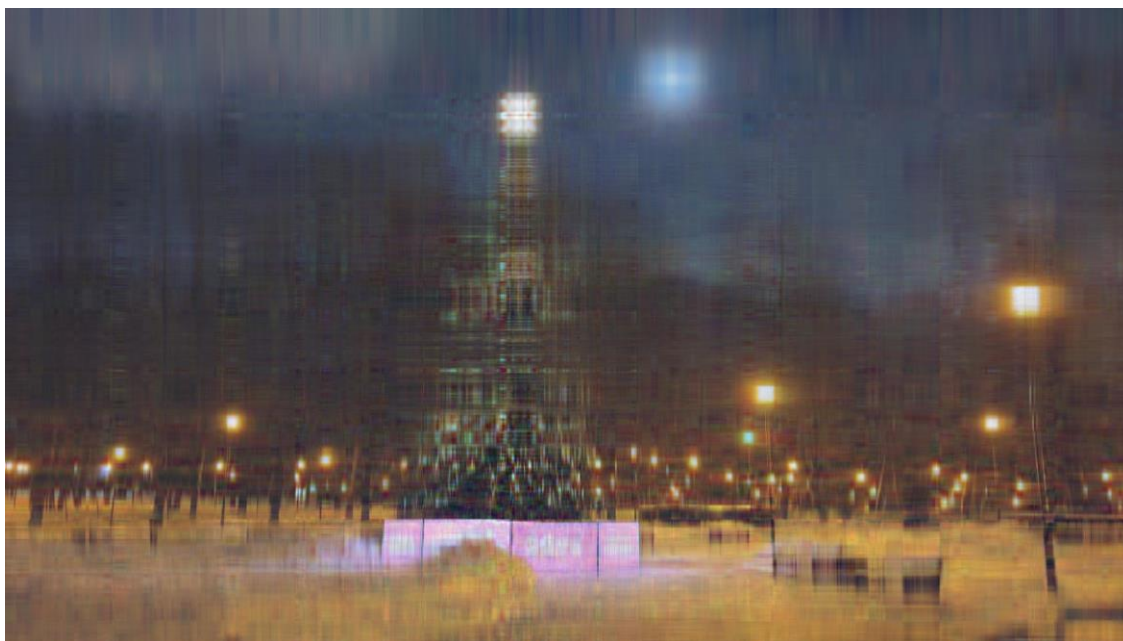


Рисунок 50 – Изображение с 69.76% сжатия по сингулярным числам,  
87.94% сжатия по памяти, размер 437 КБ

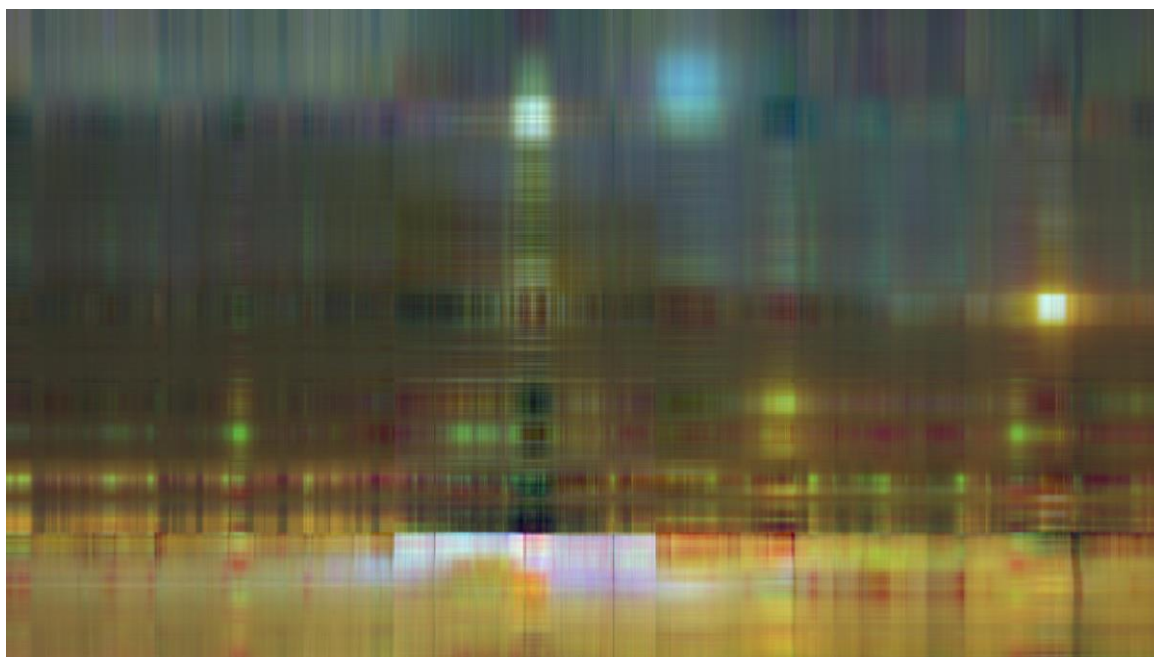


Рисунок 51 – Изображение с 79.53% сжатия по сингулярным числам,  
90.41% сжатия по памяти, размер 348 КБ

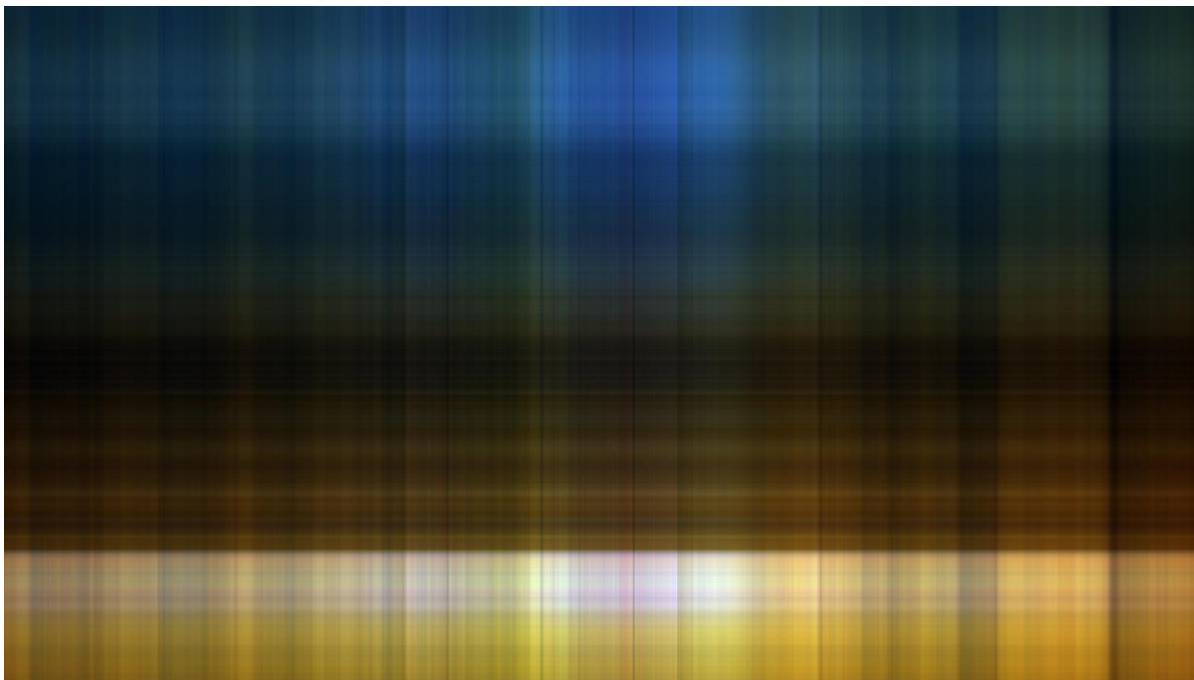


Рисунок 52 – Изображение с 87.51% сжатия по сингулярным числам,  
91.7% сжатия по памяти, размер 301 КБ

Была вычислена степень сжатия изображения по сингулярным числам и по памяти, а также размер полученного изображения. Результаты представлены в таблице 2.

Таблица 2 – Результаты сжатия изображения



Изображение	Степень сжатия по сингулярным числам	Степень сжатия по памяти	Размер изображения
 Рисунок 44	—	—	3629 КБ
	19.99%	74.76%	916 КБ






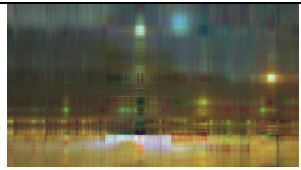
Рисунок 45			
 <p>Рисунок 46</p>	29.98%	78.57%	778 КБ
 <p>Рисунок 47</p>	39.95%	81.52%	671 КБ
 <p>Рисунок 48</p>	49.88%	83.75%	590 КБ
 <p>Рисунок 49</p>	59.92%	86.15%	503 КБ
 <p>Рисунок 50</p>	69.76%	87.94%	437 КБ
 <p>Рисунок 51</p>	79.53%	90.41%	348 КБ

 Рисунок 52	87.51%	91.7%	301 КБ
---	--------	-------	--------

## ЗАКЛЮЧЕНИЕ

В результате выполнения курсовой работы:

- были реализованы методы А. М. Данилевского и А. Н. Крылова для поиска собственных значений и собственных векторов матрицы, для данных методов также были реализованы вспомогательные методы – метод деления отрезка пополам и метод нахождения кругов Гершгорина;
- был реализован метод QR-разложения матрицы;
- были реализованы методы сингулярного разложения матрицы – метод, использующий собственные значения, метод, использующий QR-разложение, итерационные методы;
- была оценена точность реализованных методов;
- были рассмотрены различные методы сжатия изображений;
- сингулярное разложение матрицы было применено для задачи сжатия изображений.

## СПИСОК ЛИТЕРАТУРЫ

1. Афанасьева, А. А. Применение сингулярного разложения для сжатия изображений и решения плохо обусловленных систем линейных уравнений / А. А. Афанасьева, А. В. Старченко // Всероссийская молодежная научная конференция студентов, аспирантов и молодых ученых "Все грани математики и механики" : Сборник статей, Томск, 23–27 апреля 2019 года / Под редакцией А.В. Старченко. – Томск: Издательский Дом Томского государственного университета, 2019. – С. 99-110.
2. Старовойтов, В. В. Сингулярное разложение матриц в анализе цифровых изображений / В. В. Старовойтов // Информатика. – 2017. – № 2(54). – С. 70-83.
3. Ибрагимова, Ф. И. Применение сингулярного разложения матриц / Ф. И. Ибрагимова, А. А. Ковальчук // Дни науки : материалы Национальной научно-технической конференции студентов и курсантов, Калининград, 11–25 апреля 2022 года / Калининградский государственный технический университет. – Калининград: Обособленное структурное подразделение "Балтийская государственная академия рыбопромыслового флота" федерального государственного бюджетного образовательного учреждения высшего профессионального образования "Калининградский государственный технический университет", 2022. – С. 106-110.
4. Электронный ресурс // [http://lib.ru/TECHBOOKS/ALGO/VATOLIN/algcomp.htm#\\_Toc448152502](http://lib.ru/TECHBOOKS/ALGO/VATOLIN/algcomp.htm#_Toc448152502) (дата обращения 07.12.23)
5. Сайт // <https://habr.com/ru/articles/275273/> (дата обращения 07.12.23).
6. Сайт // <https://russianblogs.com/article/91441564148/> (дата обращения 07.12.23).
7. Сайт //



[http://www.machinelearning.ru/wiki/index.php?title=%D0%A1%D0%B8%D0%BD%D0%B3%D1%83%D0%BB%D1%8F%D1%80%D0%BD%D0%BE%D0%B5\\_%D1%80%D0%B0%D0%B7%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5](http://www.machinelearning.ru/wiki/index.php?title=%D0%A1%D0%B8%D0%BD%D0%B3%D1%83%D0%BB%D1%8F%D1%80%D0%BD%D0%BE%D0%B5_%D1%80%D0%B0%D0%B7%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5) (дата обращения 07.12.23).

## ПРИЛОЖЕНИЕ

В данном приложении представлен код программы.

В листинге 1 представлен полный код программы.

### Листинг 1 – Полный код программы

```
import math
import copy
import random
import numpy as np

def scalar_mult_vec(vec1, vec2):
    n = len(vec1)
    s = 0
    for i in range(n):
        s += vec1[i] * vec2[i]
    return s

def norm_vec(vec):
    n = len(vec)
    s = 0
    for i in range(n):
        s += vec[i] ** 2
    norm = s ** 0.5
    return norm

def norm_vec_sq(vec):
    n = len(vec)
    s = 0
    for i in range(n):
        s += vec[i] ** 2
    norm = s
    return norm

def sum_vec(vec1, vec2):
    n = len(vec1)
    vec = [0] * n
    for i in range(n):
        vec[i] = vec1[i] + vec2[i]
    return vec

def sub_vec(vec1, vec2):
    n = len(vec1)
    vec = [0] * n
    for i in range(n):
        vec[i] = vec1[i] - vec2[i]
    return vec

def mult_vec_num(num, vec):
    n = len(vec)
    vec1 = [0] * n
    for i in range(n):
        vec1[i] = num * vec[i]
```

```

    return vec1

def mult_matr_matr(matr1, matr2):
    n1 = len(matr1)
    m1 = len(matr1[0])
    m2 = len(matr2[0])
    matr = [[0] * m2 for i in range(n1)]
    for i in range(n1):
        for j in range(m2):
            for k in range(m1):
                matr[i][j] += matr1[i][k] * matr2[k][j]
    return matr

def sub_matr_matr(matr1, matr2):
    n = len(matr1)
    matr = [[0] * n for i in range(n)]
    for i in range(n):
        for j in range(n):
            matr[i][j] = matr1[i][j] - matr2[i][j]
    return matr

def norm_matr(matr):
    n = len(matr)
    s = 0
    for i in range(n):
        for j in range(n):
            s += matr[i][j] ** 2
    norm = s ** 0.5
    return norm

def print_matr(matr):
    n = len(matr)
    m = len(matr[0])
    for i in range(n):
        for j in range(m):
            print("{value:4.20f}".format(value=matr[i][j]), end=" ")
        print()

def mult_matr_vec(matr, vec):
    n = len(vec)
    m = len(matr[0])
    vec1 = [0] * n
    for i in range(n):
        for j in range(m):
            vec1[i] += matr[i][j] * vec[j]
    return vec1

def mult_matr_num(num, matr):
    n = len(matr)
    matr1 = [[0] * n for i in range(n)]
    for i in range(n):
        for j in range(n):
            matr1[i][j] = num * matr[i][j]
    return matr1

def transp_matr(matr):

```

```

n = len(matr)
m = len(matr[0])
matr1 = [[0] * n for i in range(m)]
for i in range(m):
    for j in range(n):
        matr1[i][j] = matr[j][i]
return matr1

def generate_matrix(n, m, v1, v2):
    a = [[0] * m for _ in range(n)]
    for i in range(n):
        for j in range(m):
            a[i][j] = random.uniform(v1, v2)
    return a

def generate_symm_matrix(n, v1, v2):
    a = [[0] * n for i in range(n)]
    for i in range(n):
        for j in range(i, n):
            a[i][j] = random.uniform(v1, v2)
            if i != j:
                a[j][i] = a[i][j]
    return a

def det_matr(matr):
    n = len(matr)
    if n == 2:
        return matr[0][0] * matr[1][1] - matr[1][0] * matr[0][1]
    else:
        d = 0
        for i in range(n):
            d += matr[0][i] * alg_add(matr, 0, i)
        return d

def alg_add(matr, i, j):
    n = len(matr)
    matr1 = []
    for k in range(n):
        if k != i:
            matr1.append([])
            for q in range(n):
                if q != j:
                    matr1[-1].append(matr[k][q])
    return (-1) ** (i + j) * det_matr(matr1)

def inv_matr(matr):
    n = len(matr)
    matr_inv = [[0] * n for i in range(n)]
    c = 1 / det_matr(matr)
    for i in range(n):
        for j in range(n):
            matr_inv[j][i] = c * alg_add(matr, i, j)
    return matr_inv

def gauss_method(a, b):
    n = len(a)
    for i in range(n):

```

```

        for j in range(i + 1, n):
            c = - a[j][i] / a[i][i]
            for k in range(i, n):
                if k == i:
                    a[j][k] = 0
                else:
                    a[j][k] += c * a[i][k]
            b[j] += c * b[i]
    x = [0] * n
    for i in range(n - 1, -1, -1):
        x[i] = b[i]
        for j in range(n - 1, i, -1):
            x[i] -= x[j] * a[i][j]
        x[i] /= a[i][i]
    return x

def func(p):
    n = len(p)
    p = [1] + p
    return lambda x: (-1) ** n * sum([x ** i * p[n - i] * (1 if i == n else -1) for i in range(n, -1, -1)])

def div_half_method(a, b, f):
    s = 0.1
    d = 0.0001
    res = []
    x_last = a
    x = x_last
    while x <= b:
        x = x_last + s
        if f(x) * f(x_last) < 0:
            x_left = x_last
            x_right = x
            x_mid = (x + x_last) / 2
            while abs(f(x_mid)) >= d:
                if f(x_left) * f(x_mid) < 0:
                    x_right = x_mid
                else:
                    x_left = x_mid
            x_mid = (x_left + x_right) / 2
            res.append(x_mid)
        x_last = x
    return res

def gershgorin_rounds(a):
    left = -100000
    right = 100000
    n = len(a)
    for i in range(n):
        s = 0
        for j in range(n):
            if i != j:
                s += abs(a[i][j])
        b1 = a[i][i] - s
        b2 = a[i][i] + s
        if i == 0:
            left = b1
            right = b2
        elif b1 < left:
            left = b1

```

```

        elif b2 > right:
            right = b2
    return [left, right]

def danilevsky_method(a):
    n = len(a)
    m = n - 1

    b = [[0] * n for i in range(n)]
    for i in range(n):
        b[i][i] = 1
    for j in range(n):
        if j != m - 1:
            b[m - 1][j] = -a[m][j] / a[m][m - 1]
    b[m - 1][m - 1] = 1 / a[m][m - 1]

    b_mul = copy.deepcopy(b)

    c = [[0] * n for i in range(n)]
    for i in range(n):
        c[i][m - 1] = a[i][m - 1] * b[m - 1][m - 1]
    for i in range(n - 1):
        for j in range(n):
            if j != m - 1:
                c[i][j] = a[i][j] + a[i][m - 1] * b[m - 1][j]

    b_inv = [[0] * n for i in range(n)]
    for i in range(n):
        b_inv[i][i] = 1
    for j in range(n):
        b_inv[m - 1][j] = a[m][j]

    d = [[0] * n for i in range(n)]
    for i in range(m - 1):
        for j in range(n):
            d[i][j] = c[i][j]
    for j in range(n):
        for k in range(n):
            d[m - 1][j] += a[m][k] * c[k][j]
    d[m][m - 1] = 1

    for k in range(2, n):
        b = [[0] * n for i in range(n)]
        for i in range(n):
            b[i][i] = 1
        for j in range(n):
            if j != m - k:
                b[m - k][j] = -d[m - k + 1][j] / d[m - k + 1][m - k]
        b[m - k][m - k] = 1 / d[m - k + 1][m - k]

        b_mul = mult_matr_matr(b_mul, b)

        b_inv = inv_matr(b)
        d = mult_matr_matr(b_inv, d)
        d = mult_matr_matr(d, b)

    b = copy.deepcopy(b_mul)
    p = d[0][:]
    g = gershgorin_rounds(a)

    ls = div_half_method(g[0], g[1], func(p))

```

```

vectors = []
for l in ls:
    y = [1]
    for i in range(1, n):
        y.append(l ** i)
    y = y[::-1]
    y = mult_matr_vec(b, y)
    norm = norm_vec(y)
    for i in range(n):
        y[i] /= norm
    vectors.append(y)
return ls, vectors

def krylov_method(a):
    n = len(a)
    y = [[0] * n for _ in range(n + 1)]
    y[0][0] = 1
    for i in range(1, n + 1):
        y[i] = mult_matr_vec(a, y[i - 1])
    m = [[1] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            m[i][j] = y[n - 1 - j][i]
    p = gauss_method(m, y[n])
    g = gershgorin_rounds(a)
    ls = div_half_method(g[0], g[1], func(p))
    p = p[::-1]
    q = [[0] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            if j == 0:
                q[j][i] = 1
            else:
                q[j][i] = ls[i] * q[j - 1][i] - p[n - j]
    x = []
    for i in range(n):
        xi = [0] * n
        for j in range(n):
            xi = sum_vec(xi, mult_vec_num(q[j][i], y[n - 1 - j]))
        x.append(xi)
    vectors = []
    for xi in x:
        norm = norm_vec(xi)
        for i in range(n):
            xi[i] /= norm
        vectors.append(xi)
    return ls, vectors

def qr_decomposition(matrix):
    q = []
    r = [[0] * len(matrix[0]) for _ in range(len(matrix[0]))]

    for j in range(len(matrix[0])):
        v = matrix[j]
        for i in range(len(q)):
            rij = scalar_mult_vec(q[i], matrix[j])
            r[i][j] = rij
            v = sub_vec(v, mult_vec_num(rij, q[i]))
        r[j][j] = math.sqrt(scalar_mult_vec(v, v))
        q.append(mult_vec_num(1 / r[j][j], v))

```

```

    return q, r

def svd_decomposition_eigenvalues_danilevsky(a):
    a_transp_a = mult_matr_matr(transp_matr(a), a)

    eigenvalues, eigenvectors = danilevsky_method(a_transp_a)

    singular_values = [math.sqrt(eigenvalue) for eigenvalue in eigenvalues]

    left_singular_vectors = mult_matr_matr(a, eigenvectors)
    for i in range(len(left_singular_vectors)):
        for j in range(len(left_singular_vectors[0])):
            left_singular_vectors[i][j] /= singular_values[j]

    right_singular_vectors = eigenvectors

    return left_singular_vectors, singular_values,
    transp_matr(right_singular_vectors)

def svd_decomposition_eigenvalues_krylov(a):
    a_transp_a = mult_matr_matr(transp_matr(a), a)

    eigenvalues, eigenvectors = krylov_method(a_transp_a)

    singular_values = [math.sqrt(eigenvalue) for eigenvalue in eigenvalues]

    left_singular_vectors = mult_matr_matr(a, eigenvectors)
    for i in range(len(left_singular_vectors)):
        for j in range(len(left_singular_vectors[0])):
            left_singular_vectors[i][j] /= singular_values[j]

    right_singular_vectors = eigenvectors

    return left_singular_vectors, singular_values,
    transp_matr(right_singular_vectors)

def svd_decomposition_qr(matrix, num_iterations=100):
    m, n = len(matrix), len(matrix[0])

    u = [[random.random() for _ in range(m)] for _ in range(m)]
    v = [[random.random() for _ in range(n)] for _ in range(n)]

    for _ in range(num_iterations):
        u, _ = qr_decomposition(mult_matr_matr(matrix, v))
        v, _ = qr_decomposition(mult_matr_matr(transp_matr(matrix), u))
    sigma = mult_matr_matr(transp_matr(u), mult_matr_matr(matrix, v))
    return u, sigma, transp_matr(v)

def power_iteration(matrix, num_iterations):
    b_k = np.random.rand(matrix.shape[1])
    for _ in range(num_iterations):
        b_k1 = np.dot(matrix, b_k)
        b_k1_norm = np.linalg.norm(b_k1)
        b_k = b_k1 / b_k1_norm
    return b_k

def find_eigvals_and_vecs(matrix, num_eigenvalues, num_iterations):
    eigenvalues = []

```



```

eigenvectors = []
matrix1 = copy.deepcopy(matrix)
for _ in range(num_eigenvalues):
    eigenvector = power_iteration(matrix1, num_iterations)
    eigenvalue = np.dot(np.dot(eigenvector.T, matrix1), eigenvector)
    np.dot(eigenvector.T, eigenvector)
    eigenvalues.append(eigenvalue)
    eigenvectors.append(eigenvector)
    matrix1 -= eigenvalue * np.outer(eigenvector, eigenvector)
return np.array(eigenvalues), np.array(eigenvectors).T

def gram_schmidt_process(vectors):
    orthogonal_vectors = []
    for v in vectors.T:
        for u in orthogonal_vectors:
            v -= np.dot(u, v) * u
        v_norm = np.linalg.norm(v)
        if v_norm > 1e-6:
            v /= v_norm
            orthogonal_vectors.append(v)
    return np.array(orthogonal_vectors).T

def svd_decomposition_iterations(matrix):
    matrix = np.array(matrix)
    n = matrix.shape[0]

    eigvals_A_A_T, eigvecs_A_A_T = find_eigvals_and_vecs(np.dot(matrix,
matrix.T), n, 100)
    eigvals_A_T_A, eigvecs_A_T_A = find_eigvals_and_vecs(np.dot(matrix.T,
matrix), n, 100)

    sing_vals = np.sqrt(eigvals_A_T_A)
    u = gram_schmidt_process(eigvecs_A_A_T.T).T
    s = np.zeros((n, n))
    np.fill_diagonal(s, sing_vals)
    vt = np.zeros((n, n))
    for i, sing_val in enumerate(sing_vals):
        vt[i] = 1 / sing_val * np.dot(matrix.T, eigvecs_A_A_T.T[i])
    return u, s, vt

matrix = generate_symm_matrix(5, 0, 5)
print("Matrix:")
print_matr(matrix)
print()

u, s, vt = svd_decomposition_eigenvalues_danilevsky(matrix)
# u, s, vt = svd_decomposition_eigenvalues_krylov(matrix)

sigma = [[0] * len(s) for _ in range(len(s))]
for i in range(len(s)):
    sigma[i][i] = s[i]

print("Result:")
print("U:")
print_matr(u)
print()
print("Sigma:")
print_matr(sigma)
print()
print("V*:")

```

```

print_matr(vt)
print()

res = mult_matr_matr(mult_matr_matr(u, sigma), vt)
print("Result after multiplication")
print_matr(res)
print()

print("Error rate")
print(norm_matr(sub_matr_matr(matrix, res)))
print()

# matrix = generate_matrix(5, 5, 0, 5)
# print("Matrix:")
# print_matr(matrix)
# print()
#
# u, sigma, vt = svd_decomposition_qr(matrix)
#
# print("Result:")
# print("U:")
# print_matr(u)
# print()
# print("Sigma:")
# print_matr(sigma)
# print()
# print("V*:")
# print_matr(vt)
# print()
#
# res = mult_matr_matr(mult_matr_matr(u, sigma), vt)
# print("Result after multiplication")
# print_matr(res)
# print()
#
# print("Error rate")
# print(norm_matr(sub_matr_matr(matrix, res)))
# print()

# matrix = generate_matrix(5, 5, 0, 5)
# print("Matrix:")
# print_matr(matrix)
# print()
#
# u, sigma, vt = svd_decomposition_iterations(matrix)
#
# print("Result:")
# print("U:")
# print_matr(u)
# print()
# print("Sigma:")
# print_matr(sigma)
# print()
# print("V*:")
# print_matr(vt)
# print()
#
# res = mult_matr_matr(mult_matr_matr(u, sigma), vt)
# print("Result after multiplication")
# print_matr(res)
# print()

```

```

#
# print("Error rate")
# print(norm_matr(sub_matr_matr(matrix, res)))
# print()

import numpy as np
import matplotlib.pyplot as plt
import os

def zip_img_by_svd(img, plotId, imgfile, imgfile_size, rate=0.8):
    zip_img = np.zeros(img.shape)
    zip_rate_singular = 0
    for chanel in range(3):
        u, sigma, v = svd_decomposition_qr(img[:, :, chanel])
        u = np.array(u)
        sigma = np.array(sigma)
        v = np.array(v)
        sigma = np.diag(sigma)
        sigma_i = 0
        temp = 0
        while (temp / np.sum(sigma)) < rate:
            temp += sigma[sigma_i]
            sigma_i += 1
        zip_rate_singular += temp / np.sum(sigma)
        SigmaMat = np.zeros((sigma_i, sigma_i))
        for i in range(sigma_i):
            SigmaMat[i][i] = sigma[i]
        zip_img[:, :, chanel] = u[:,
0:sigma_i].dot(SigmaMat).dot(v[0:sigma_i, :])
        zip_rate_singular /= 3
        zip_rate_singular = 1 - zip_rate_singular
        for i in range(3):
            MAX = np.max(zip_img[:, :, i])
            MIN = np.min(zip_img[:, :, i])
            zip_img[:, :, i] = (zip_img[:, :, i] - MIN) / (MAX - MIN)
        zip_img = np.round(zip_img * 255).astype("uint8")
        plt.imsave("zip_" + imgfile + str(plotId - 1) + ".jpg", zip_img)
        size = os.path.getsize("zip_" + imgfile + str(plotId - 1) + ".jpg")
        size_rate = 1 - size / imgfile_size
        print(plotId - 1)
        print("Изображение с " + str(round(zip_rate_singular * 100, 2)) + "%
сжатия по сингулярным числам, " + str(round(size_rate * 100, 2)) + "% сжатия
по памяти, размер " + str(round(size / 1024)) + " КБ")
        print()
        f = plt.subplot(3, 3, plotId)
        plt.subplots_adjust(hspace=0.5, wspace=0.9)
        f.imshow(zip_img)
        f.axis('off')
        f.set_title(str(round(zip_rate_singular * 100, 2)) + "% сжатия по
сингулярным числам\n" + str(round(size_rate * 100, 2)) + "% сжатия по
памяти\nРазмер " + str(round(size / 1024)) + " КБ")

if __name__ == '__main__':
    imgfile = "img1.jpg"
    imgfile_size = os.path.getsize(imgfile)
    plt.figure(figsize=(12, 12))
    plt.rcParams['font.sans-serif'] = 'SimHei'
    img = plt.imread(imgfile)
    f1 = plt.subplot(331)
    plt.subplots_adjust(hspace=0.5, wspace=0.9)

```

```

f1.imshow(img)
f1.axis('off')
f1.set_title("Исходное изображение\nРазмер " + str(round(imgfile_size /
1024)) + " КБ")
for i in range(8):
    rate = (i + 1) / 10.0
    zip_img_by_svd(img, i + 2, imgfile[:4], imgfile_size, rate)
plt.show()

```

В листинге 2 представлен метод деления отрезка пополам.

## Листинг 2 – Метод деления отрезка пополам

```

def div_half_method(a, b, f):
    s = 0.1
    d = 0.0001
    res = []
    x_last = a
    x = x_last
    while x <= b:
        x = x_last + s
        if f(x) * f(x_last) < 0:
            x_left = x_last
            x_right = x
            x_mid = (x + x_last) / 2
            while abs(f(x_mid)) >= d:
                if f(x_left) * f(x_mid) < 0:
                    x_right = x_mid
                else:
                    x_left = x_mid
            x_mid = (x_left + x_right) / 2
            res.append(x_mid)
        x_last = x
    return res

```

В листинге 3 представлен метод нахождения кругов Гершгорина.

## Листинг 3 – Метод нахождения кругов Гершгорина

```

def gershgorin_rounds(a):
    left = -100000
    right = 100000
    n = len(a)
    for i in range(n):
        s = 0
        for j in range(n):
            if i != j:
                s += abs(a[i][j])
        b1 = a[i][i] - s
        b2 = a[i][i] + s
        if i == 0:
            left = b1
            right = b2
        elif b1 < left:
            left = b1
        elif b2 > right:

```

```

        right = b2
    return [left, right]

```

В листинге 4 представлен метод А. М. Данилевского.

#### Листинг 4 – Метод А. М. Данилевского

```

def danilevsky_method(a):
    n = len(a)
    m = n - 1

    b = [[0] * n for i in range(n)]
    for i in range(n):
        b[i][i] = 1
    for j in range(n):
        if j != m - 1:
            b[m - 1][j] = -a[m][j] / a[m][m - 1]
    b[m - 1][m - 1] = 1 / a[m][m - 1]

    b_mul = copy.deepcopy(b)

    c = [[0] * n for i in range(n)]
    for i in range(n):
        c[i][m - 1] = a[i][m - 1] * b[m - 1][m - 1]
    for i in range(n - 1):
        for j in range(n):
            if j != m - 1:
                c[i][j] = a[i][j] + a[i][m - 1] * b[m - 1][j]

    b_inv = [[0] * n for i in range(n)]
    for i in range(n):
        b_inv[i][i] = 1
    for j in range(n):
        b_inv[m - 1][j] = a[m][j]

    d = [[0] * n for i in range(n)]
    for i in range(m - 1):
        for j in range(n):
            d[i][j] = c[i][j]
    for j in range(n):
        for k in range(n):
            d[m - 1][j] += a[m][k] * c[k][j]
    d[m][m - 1] = 1

    for k in range(2, n):
        b = [[0] * n for i in range(n)]
        for i in range(n):
            b[i][i] = 1
        for j in range(n):
            if j != m - k:
                b[m - k][j] = -d[m - k + 1][j] / d[m - k + 1][m - k]
        b[m - k][m - k] = 1 / d[m - k + 1][m - k]

        b_mul = mult_matr_matr(b_mul, b)

        b_inv = inv_matr(b)
        d = mult_matr_matr(b_inv, d)
        d = mult_matr_matr(d, b)

    b = copy.deepcopy(b_mul)
    p = d[0][:]

```

```

g = gershgorin_rounds(a)

ls = div_half_method(g[0], g[1], func(p))

vectors = []
for l in ls:
    y = [1]
    for i in range(1, n):
        y.append(l ** i)
    y = y[::-1]
    y = mult_matr_vec(b, y)
    norm = norm_vec(y)
    for i in range(n):
        y[i] /= norm
    vectors.append(y)
return ls, vectors

```

В листинге 5 представлен метод А. Н. Крылова.

### Листинг 5 – Метод А. Н. Крылова

```

def krylov_method(a):
    n = len(a)
    y = [[0] * n for _ in range(n + 1)]
    y[0][0] = 1
    for i in range(1, n + 1):
        y[i] = mult_matr_vec(a, y[i - 1])
    m = [[1] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            m[i][j] = y[n - 1 - j][i]
    p = gauss_method(m, y[n])
    g = gershgorin_rounds(a)
    ls = div_half_method(g[0], g[1], func(p))
    p = p[::-1]
    q = [[0] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            if j == 0:
                q[j][i] = 1
            else:
                q[j][i] = ls[i] * q[j - 1][i] - p[n - j]
    x = []
    for i in range(n):
        xi = [0] * n
        for j in range(n):
            xi = sum_vec(xi, mult_vec_num(q[j][i], y[n - 1 - j]))
        x.append(xi)
    vectors = []
    for xi in x:
        norm = norm_vec(xi)
        for i in range(n):
            xi[i] /= norm
        vectors.append(xi)
    return ls, vectors

```

В листинге 6 представлен метод нахождения QR-разложения матрицы.

### Листинг 6 – Метод QR-разложения матрицы

```

def qr_decomposition(matrix):
    q = []
    r = [[0] * len(matrix[0]) for _ in range(len(matrix[0]))]

    for j in range(len(matrix[0])):
        v = matrix[j]
        for i in range(len(q)):
            rij = scalar_mult_vec(q[i], matrix[j])
            r[i][j] = rij
            v = sub_vec(v, mult_vec_num(rij, q[i]))
        r[j][j] = math.sqrt(scalar_mult_vec(v, v))
        q.append(mult_vec_num(1 / r[j][j], v))

    return q, r

```

В листинге 7 представлен метод сингулярного разложения матрицы при помощи поиска собственных значений методом А. М. Данилевского.

**Листинг 7 – Метод сингулярного разложения матрицы при помощи поиска собственных значений методом А. М. Данилевского**

```

def svd_decomposition_eigenvalues_danilevsky(a):
    a_transp_a = mult_matr_matr(transp_matr(a), a)

    eigenvalues, eigenvectors = danilevsky_method(a_transp_a)

    singular_values = [math.sqrt(eigenvalue) for eigenvalue in eigenvalues]

    left_singular_vectors = mult_matr_matr(a, eigenvectors)
    for i in range(len(left_singular_vectors)):
        for j in range(len(left_singular_vectors[0])):
            left_singular_vectors[i][j] /= singular_values[j]

    right_singular_vectors = eigenvectors

    return left_singular_vectors, singular_values,
    transp_matr(right_singular_vectors)

```

В листинге 8 представлен метод сингулярного разложения матрицы при помощи поиска собственных значений методом А. Н. Крылова.

**Листинг 8 – Метод сингулярного разложения матрицы при помощи поиска собственных значений методом А. Н. Крылова**

```

def svd_decomposition_eigenvalues_krylov(a):
    a_transp_a = mult_matr_matr(transp_matr(a), a)

    eigenvalues, eigenvectors = krylov_method(a_transp_a)

    singular_values = [math.sqrt(eigenvalue) for eigenvalue in eigenvalues]

    left_singular_vectors = mult_matr_matr(a, eigenvectors)
    for i in range(len(left_singular_vectors)):

```

```

        for j in range(len(left_singular_vectors[0])):
            left_singular_vectors[i][j] /= singular_values[j]

    right_singular_vectors = eigenvectors

    return left_singular_vectors, singular_values,
    transp_matr(right_singular_vectors)

```

В листинге 9 представлен метод сингулярного разложения матрицы при помощи QR-разложения матрицы.

Листинг 9 – Метод сингулярного разложения матрицы при помощи QR-разложения матрицы

```

def svd_decomposition_qr(matrix, num_iterations=100):
    m, n = len(matrix), len(matrix[0])

    u = [[random.random() for _ in range(m)] for _ in range(m)]
    v = [[random.random() for _ in range(n)] for _ in range(n)]

    for _ in range(num_iterations):
        u, _ = qr_decomposition(mult_matr_matr(matrix, v))
        v, _ = qr_decomposition(mult_matr_matr(transp_matr(matrix), u))
        sigma = mult_matr_matr(transp_matr(u), mult_matr_matr(matrix, v))
    return u, sigma, transp_matr(v)

```

В листинге 10 представлен итерационный метод нахождения сингулярного разложения матрицы.

Листинг 10 – Итерационный метод нахождения сингулярного разложения матрицы

```

def power_iteration(matrix, num_iterations):
    b_k = np.random.rand(matrix.shape[1])
    for _ in range(num_iterations):
        b_k1 = np.dot(matrix, b_k)
        b_k1_norm = np.linalg.norm(b_k1)
        b_k = b_k1 / b_k1_norm
    return b_k

def find_eigvals_and_vecs(matrix, num_eigenvalues, num_iterations):
    eigenvalues = []
    eigenvectors = []
    matrix1 = copy.deepcopy(matrix)
    for _ in range(num_eigenvalues):
        eigenvector = power_iteration(matrix1, num_iterations)
        eigenvalue = np.dot(np.dot(eigenvector.T, matrix1), eigenvector)
        np.dot(eigenvector.T, eigenvector)
        eigenvalues.append(eigenvalue)
        eigenvectors.append(eigenvector)
        matrix1 -= eigenvalue * np.outer(eigenvector, eigenvector)
    return np.array(eigenvalues), np.array(eigenvectors).T

```



```

def gram_schmidt_process(vectors):
    orthogonal_vectors = []
    for v in vectors.T:
        for u in orthogonal_vectors:
            v -= np.dot(u, v) * u
        v_norm = np.linalg.norm(v)
        if v_norm > 1e-6:
            v /= v_norm
            orthogonal_vectors.append(v)
    return np.array(orthogonal_vectors).T

def svd_decomposition_iterations(matrix):
    matrix = np.array(matrix)
    n = matrix.shape[0]

    eigvals_A_A_T , eigvecs_A_A_T = find_eigvals_and_vecs(np.dot (matrix,
matrix.T), n, 100)
    eigvals_A_T_A , eigvecs_A_T_A = find_eigvals_and_vecs(np.dot (matrix.T,
matrix), n, 100)

    sing_vals = np.sqrt(eigvals_A_T_A)
    u = gram_schmidt_process(eigvecs_A_A_T.T).T
    s = np.zeros((n, n))
    np.fill_diagonal(s, sing_vals)
    vt = np.zeros((n, n))
    for i, sing_val in enumerate(sing_vals):
        vt[i] = 1 / sing_val * np.dot(matrix.T, eigvecs_A_A_T.T[i])
    return u, s, vt

```

В листинге 11 представлен код программы, реализующей сжатие изображения при помощи сингулярного разложения.

**Листинг 11 – Сжатие изображения при помощи сингулярного разложения матрицы**

```

import numpy as np
import matplotlib.pyplot as plt
import os

def zip_img_by_svd(img, plotId, imgfile, imgfile_size, rate=0.8):
    zip_img = np.zeros(img.shape)
    zip_rate_singular = 0
    for chanel in range(3):
        u, sigma, v = svd_decomposition_qr(img[:, :, chanel])
        u = np.array(u)
        sigma = np.array(sigma)
        v = np.array(v)
        sigma = np.diag(sigma)
        sigma_i = 0
        temp = 0
        while (temp / np.sum(sigma)) < rate:
            temp += sigma[sigma_i]
            sigma_i += 1
        zip_rate_singular += temp / np.sum(sigma)

```

```

        SigmaMat = np.zeros((sigma_i, sigma_i))
        for i in range(sigma_i):
            SigmaMat[i][i] = sigma[i]
            zip_img[:, :, chanel] = u[:,
0:sigma_i].dot(SigmaMat).dot(v[0:sigma_i, :])
            zip_rate_singular /= 3
            zip_rate_singular = 1 - zip_rate_singular
        for i in range(3):
            MAX = np.max(zip_img[:, :, i])
            MIN = np.min(zip_img[:, :, i])
            zip_img[:, :, i] = (zip_img[:, :, i] - MIN) / (MAX - MIN)
        zip_img = np.round(zip_img * 255).astype("uint8")
        plt.imsave("zip_" + imgfile + str(plotId - 1) + ".jpg", zip_img)
        size = os.path.getsize("zip_" + imgfile + str(plotId - 1) + ".jpg")
        size_rate = 1 - size / imgfile_size
        print(plotId - 1)
        print("Изображение с " + str(round(zip_rate_singular * 100, 2)) + "%
сжатия по сингулярным числам, " + str(round(size_rate * 100, 2)) + "% сжатия
по памяти, размер " + str(round(size / 1024)) + " КБ")
        print()
        f = plt.subplot(3, 3, plotId)
        plt.subplots_adjust(hspace=0.5, wspace=0.9)
        f.imshow(zip_img)
        f.axis('off')
        f.set_title(str(round(zip_rate_singular * 100, 2)) + "% сжатия по
сингулярным числам\n" + str(round(size_rate * 100, 2)) + "% сжатия по
памяти\nРазмер " + str(round(size / 1024)) + " КБ")

if __name__ == '__main__':
    imgfile = "img1.jpg"
    imgfile_size = os.path.getsize(imgfile)
    plt.figure(figsize=(12, 12))
    plt.rcParams['font.sans-serif'] = 'SimHei'
    img = plt.imread(imgfile)
    f1 = plt.subplot(331)
    plt.subplots_adjust(hspace=0.5, wspace=0.9)
    f1.imshow(img)
    f1.axis('off')
    f1.set_title("Исходное изображение\nРазмер " + str(round(imgfile_size /
1024)) + " КБ")
    for i in range(8):
        rate = (i + 1) / 10.0
        zip_img_by_svd(img, i + 2, imgfile[:4], imgfile_size, rate)
    plt.show()

```