ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Университет ИТМО

Отчёт по лабораторной работе № 2

«Проблема собственных значений матрицы»

Выполнил работу:

Демьянов Фёдор Александрович

Академическая группа: № J3114

Санкт-Петербург 2025

Цель лабораторной работы:

Изучить и реализовать итерационные методы для вычисления собственных и сингулярных чисел матриц цветовых каналов изображения, сравнить эффективность степенного метода и метода вращений Якоби, а также продемонстрировать низкоранговую аппроксимацию SVD при сжатии и восстановлении изображения.

Задачи:

1. Загрузка изображения формата JPEG (не менее 640×640) и разбиение на каналы R, G, B.

2. Реализация степенного метода для вычисления ведущего сингулярного числа \sigma_1 и соответствующих сингулярных векторов u_1, v_1 каждого канала. Исследовать сходимость и оценить норму ошибки приближения.

3. Построение полного SVD (NumPy) для каналов, анализ распределения σ_i, выбор k для усечённого SVD.

4. Реализация метода вращений Якоби для вычисления всех сингулярных чисел одного из каналов, сравнение с SVD и степенным методом по точности и количеству операций.

5. Построение гистограмм глубины цвета для каждого канала, вычисление среднего и стандартного отклонения, анализ связи с числом необходимых σ _i.

Реализация:

```
## 1. Импорт библиотек
"""

import numpy as np
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
from matplotlib import rc
import seaborn as sns
import pandas as pd

"""## 2. Загрузка изображения и выделение каналов R, G, B

### 2.1 Работа с изображением

Задаем путь до изображения
"""

image_path = 'C:/Users/5fedo/Jupyter/NM/Mass Effect_picture.jpg'

"""Загрузка изображения"""

img = mpimg.imread(image_path)

"""Приводим данные к типу float64 и нормализуем"""

img = img.astype(np.float64) / 255.0

print("Тип данных:", img.dtype)

img.shape

"""Разделим на 3 матрицы"""

R = img[:, :, 0]  # красный
G = img[:, :, 1]  # зелёный
B = img[:, :, 2]  # синий
```

Загрузка изображения и выделение каналов

Визуализация:

```
fig, axes = plt.subplots(1, 4, figsize=(16, 4))
axes[0].imshow(img)
axes[0].set_title('Оригинал')
axes[0].axis('off')

axes[1].imshow(R, cmap='gray')
axes[1].set_title('Красный канал (R)')
axes[1].axis('off')

axes[2].imshow(G, cmap='gray')
axes[2].set_title('Зелёный канал (G)')
axes[2].axis('off')

axes[3].imshow(B, cmap='gray')
axes[3].set_title('Синий канал (B)')
axes[3].axis('off')

plt.tight_layout()
plt.show()
```

Рисунок 1. Оригинальное изображение и каналы R, G, B в градациях серого

```
fig, axes = plt.subplots(1, 3, figsize=(12, 4))

axes[0].imshow(R, cmap='Reds')
axes[0].set_title('Красный канал (R)')
axes[0].axis('off')

axes[1].imshow(G, cmap='Greens')
axes[1].set_title('Зелёный канал (G)')
axes[1].axis('off')

axes[2].imshow(B, cmap='Blues')
axes[2].set_title('Синий канал (B)')
axes[2].axis('off')

plt.tight_layout()
plt.show()
```

Рисунок 2. Отдельные цветные представления каналов R, G, B на белом фоне.

```
zeros = np.zeros_like(R)

red_only   = np.stack([R,  zeros, zeros], axis=2)
green_only = np.stack([zeros, G,  zeros], axis=2)
blue_only  = np.stack([zeros, zeros, B ], axis=2)

fig, axes = plt.subplots(1, 3, figsize=(12, 4))

axes[0].imshow(red_only)
axes[0].set_title('Красный канал (R)')
axes[0].axis('off')

axes[1].imshow(green_only)
axes[1].set_title('Зелёный канал (G)')
axes[1].axis('off')

axes[2].imshow(blue_only)
axes[2].set_title('Синий канал (B)')
axes[2].axis('off')

plt.tight_layout()
plt.show()
```

Рисунок 3. Отдельные цветные представления каналов R, G, B на черном фоне.

```python
def step_method_for_svd(A: np.ndarray, num_iters: int = 500, tol: float = 1e-6, verbose: bool = False):

    # Размеры A
    m, n = A.shape

    # Инициализируем правый сингулярный нормированный вектор v случайным образом
    v = np.random.randn(n).astype(np.float64)
    v = v / np.linalg.norm(v)

    sigma_vals = []

    # Итеративный процесс:
    sigma_old = 0.0
    for k in range(num_iters):
        # Вычисляем y = A * v  (размер m)
        y = np.dot(A, v)

        # Вычисляем z = A^T * y  (размер n)
        z = np.dot(A.T, y)

        # Нормируем z, чтобы получить новый приближённый v_new
        norm_z = np.linalg.norm(z)

        if norm_z == 0:
            # Выродившийся случай: все значения нули => сингулярная величина 0
            return 0.0, np.zeros(m), np.zeros(n)

        v_new = z / norm_z

        # Текущее приближение σ₁ через ||A v_new||
        sigma_new = np.linalg.norm(A @ v_new)
        sigma_vals.append(sigma_new)

        # Критерий остановки
        if np.linalg.norm(v_new - v) < tol:
            v = v_new
            if verbose:
                print(f"Итерация {k+1:4d}: сходимость по вектору (||v_new - v_old|| = {np.linalg.norm(v_new - v):.2e}).")
            break


        # Обновляем для следующей итерации
        v = v_new
        sigma_old = sigma_new

        if verbose and (k + 1) % 50 == 0:
            print(f"Итерация {k+1:4d}: текущее приближение σ ≈ {sigma_new}")

    else:
        # Если не сработало условие tol
        sigma = sigma_new
        if verbose:
            print(f"Достигнут максимум итераций ({num_iters}). σ ≈ {sigma}")


    # Теперь у нас есть приближённый правый сингулярный вектор v и σ.
    # Вычислим левый сингулярный вектор u = (A * v) / σ.
    # При этом A * v = y (для последней итерации) — можем вычислить заново:
    y_final = np.dot(A, v)
    norm_y_final = np.linalg.norm(y_final)

    if norm_y_final == 0:
        # Вырождение
        u = np.zeros(m)
    else:
        u = y_final / norm_y_final

    sigma = norm_y_final

    return sigma_vals, sigma, u, v
```

Позволяет найти ведущий собственный вектор и собственное значение квадратной матрицы M. Для поиска ведущего сингулярного числа A применяют его к $M = A^T A$

1. Инициализировать случайный $v^0 \in \mathbb{R}^n, \| v^0 \| = 1$.
2. Повторять k = 0,1,… до сходимости:

$$y^k = A v^k$$

$$z^k = A^T y^k = A^T A v^k,$$

$$v^{k+1} = \frac{z^k}{\|z^k\|}$$

$$\sigma^{k+1} = \| A v^{k+1} \|$$

   Останавливаются, когда $\| v^{k+1} - v^k \| <$ tol или при max_iter.

3. При сходимости $v^k \to v_1, \sigma^k \to \sigma_1$. Левый сингулярный вектор $u_1 = \frac{A v_1}{\sigma_1}$

Сложность одной итерации: два умножения «матрица × вектор» O(mn) + вычисление норм O(m) и O(n)

```python
Ns = [1, 5, 10, 30]
tol = 1e-30

for N in Ns:
    # Канал R
    sigma_vals_R, sigma_R, u_R, v_R = step_method_for_svd(R, num_iters=N, tol=tol)
    R_approx = sigma_R * np.outer(u_R, v_R)
    err_R = np.linalg.norm(R - R_approx, ord='fro')

    # Канал G
    sigma_vals_G, sigma_G, u_G, v_G = step_method_for_svd(G, num_iters=N, tol=tol)
    G_approx = sigma_G * np.outer(u_G, v_G)
    err_G = np.linalg.norm(G - G_approx, ord='fro')

    # Канал B
    sigma_vals_B, sigma_B, u_B, v_B = step_method_for_svd(B, num_iters=N, tol=tol)
    B_approx = sigma_B * np.outer(u_B, v_B)
    err_B = np.linalg.norm(B - B_approx, ord='fro')

    print(f"\n=== N = {N} итераций ===")
    print(f"Канал R: σ = {sigma_R}, ‖R - σ*u*v^T‖_F = {err_R}")
    print(f"Канал G: σ = {sigma_G}, ‖G - σ*u*v^T‖_F = {err_G}")
    print(f"Канал B: σ = {sigma_B}, ‖B - σ*u*v^T‖_F = {err_B}")
```

Проводим эксперимент при N = {1, 5, 10, 30} итерациях. Для каждого N:

1. Получаем $\sigma^{(N)}, u^{(N)}, v^{(N)}$
2. Строим rank-1 приближение A $A^{(N)}_{\text{rank}-1} = \sigma^{(N)} u^{(N)} (v^{(N)})^\top$
3. Считаем Фробениус-норму ошибки:

$$\text{err}_{\text{F}}^{(\text{N})} = ||A - A^{(N)}_{\text{rank-1}}||F$$

```
=== N = 1 итераций ===
Канал R: σ = 952.5278595859724, ‖R - σ*u*v^T‖_F = 672.7650810575864
Канал G: σ = 860.2620595430451, ‖G - σ*u*v^T‖_F = 651.4223340505292
Канал B: σ = 1215.7793646447767, ‖B - σ*u*v^T‖_F = 936.5645162535521

=== N = 5 итераций ===
Канал R: σ = 966.3540028694986, ‖R - σ*u*v^T‖_F = 652.7496600657497
Канал G: σ = 876.295662994032, ‖G - σ*u*v^T‖_F = 629.6886368729637
Канал B: σ = 1350.5661089604578, ‖B - σ*u*v^T‖_F = 728.8646938415363

=== N = 10 итераций ===
Канал R: σ = 966.354003845593, ‖R - σ*u*v^T‖_F = 652.7496586207043
Канал G: σ = 876.2956629950318, ‖G - σ*u*v^T‖_F = 629.6886368715725
Канал B: σ = 1350.566108960464, ‖B - σ*u*v^T‖_F = 728.8646938415249

=== N = 30 итераций ===
Канал R: σ = 966.3540038455931, ‖R - σ*u*v^T‖_F = 652.7496586207047
Канал G: σ = 876.2956629950318, ‖G - σ*u*v^T‖_F = 629.6886368715723
Канал B: σ = 1350.566108960464, ‖B - σ*u*v^T‖_F = 728.864693841525
```

N = 1 итерация:
После одной итерации векторы u,v ещё очень далеки от истинных ведущих сингулярных векторов. Из-за этого оценка σ занижена (по сравнению с тем, что она будет в более точном разложении) и ошибка достаточно велика.

N = 5 итераций:
К пятой итерации ведущие сингулярные числа уже приблизились почти к своему «истинному» значению (полный SVD). Ошибки упали заметно по сравнению с N=1.

N = 10 итераций:
Значения σ и соответствующие ошибки практически не изменились по сравнению с N=5. Это говорит о том, что сходимость по ведущему сингулярному числу завершилась уже к N≈5 дальнейшие итерации дают лишь незначительные численные изменения.

N = 30 итераций:
При N=30 цифры полностью совпадают с тем, что мы видим при N=10, с точностью до порядка машинных погрешностей. Это означает, что после 10 итераций степенной метод уже практически достиг стационарного значения σ_1 и «векторная» составляющая перестала существенно меняться.

```
U_R, S_R, Vt_R = np.linalg.svd(R, full_matrices=False)
U_G, S_G, Vt_G = np.linalg.svd(G, full_matrices=False)
U_B, S_B, Vt_B = np.linalg.svd(B, full_matrices=False)
```

Полный SVD и распределение сингулярных чисел

Пусть A — матрица размера $m \times n$. Её сингулярное разложение:

$$A = U \Sigma V^T, \text{ где}$$

- $U(m \times m)$ — ортогональная матрица ($U^T U = I$), столбцы u_i — левые сингулярные векторы.
- Σ — $m \times n$ диагональная матрица, $\Sigma_{ii} = \sigma_i \geq 0, \sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r,\ r = \min(m, n)$.
- $V(n \times n)$ — ортогональная матрица ($V^T V = I$), столбцы v_i — правые сингулярные векторы.

Свойства:

$$A^T A = V \Sigma^T \Sigma V^T,\ A A^T = U \Sigma \Sigma^T U^T,$$

откуда $\lambda_i(A^T A) = \sigma_i^2$.

Усечённое разложение ранга k:

$$A_k = \sum_{i=1}^{k} \sigma_i\ u_i v_i^T = U_k \Sigma_k V_k^T,$$

где $U_k(m \times k),\ \Sigma_k = (\sigma_1, \dots, \sigma_k), V_k(n \times k)$

По теореме Эккарта–Канторовича это минимизирует $\parallel A - B \parallel_F$ среди матриц ранга $\leq k$.

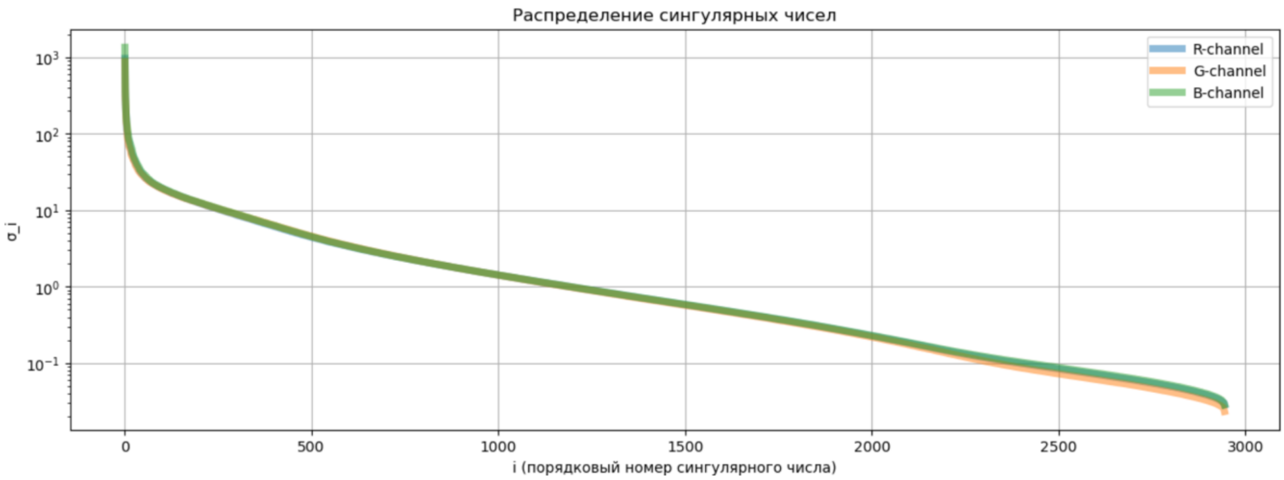

*Рисунок 4***.** Распределение сингулярных чисел (ось Y в лог-шкале)

Из графика видно, что σ_i быстро убывают: большинство энергии сосредоточено в первых ~50–100 компонентах.

Усечённый SVD: восстановление изображения

```
def low_rank_approximation(U, S, Vt, k: int):

    # Берём первые k сингулярных чисел и векторов
    U_k = U[:, :k]
    Sigma_k = np.diag(S[:k])
    Vt_k = Vt[:k, :]

    # Умножаем обратно, получаем A_k
    A_k = U_k @ Sigma_k @ Vt_k
    return A_k
```

```
def reconstruct_image(R_approx, G_approx, B_approx):

    # Стекуем каналы по третьей оси
    img_approx = np.dstack((R_approx, G_approx, B_approx))

    # Ограничим значения, чтобы не выходили за [0,1] из-за численных погрешностей
    img_approx = np.clip(img_approx, 0.0, 1.0)

    return img_approx
```

Визуализация эксперимента:

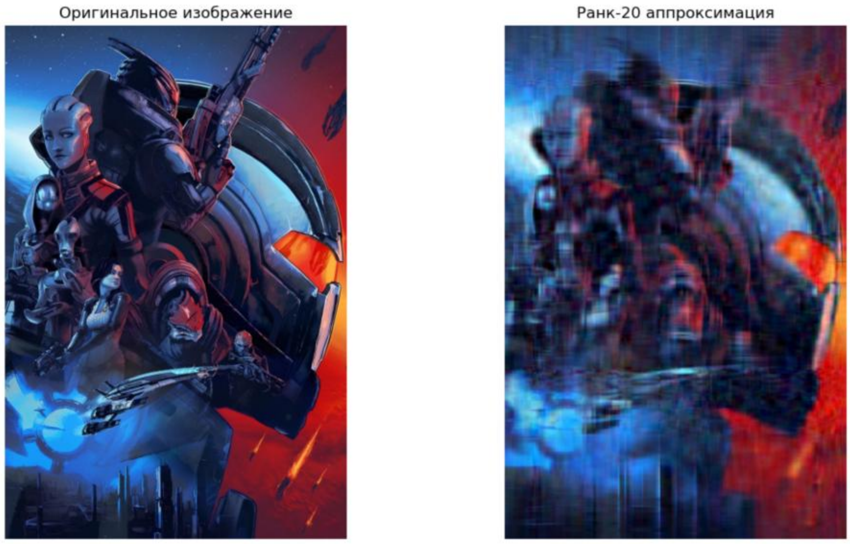

*Рисунок 5***.** Сравнение оригинального и восстановленного изображения (при k = 20)

*Рисунок 6***.** Сравнение оригинального и восстановленного изображения (при k = 50)

*Рисунок 6***.** Сравнение оригинального и восстановленного изображения (при k = 100)

*Рисунок 7***.** Сравнение оригинального и восстановленного изображения (при k = 500)

Мы видим, что при k = 500 мы получаем картинку неотличимую от оригинала

Подсчет ошибок при k = [1, 10, 20, 50, 100, 150, 300, 500, 750, 1000]

```
ks = [1, 10, 20, 50, 100, 150, 300, 500, 750, 1000]
errors = []
for k_val in ks:
    R_k_temp = low_rank_approximation(U_R, S_R, Vt_R, k_val)
    G_k_temp = low_rank_approximation(U_G, S_G, Vt_G, k_val)
    B_k_temp = low_rank_approximation(U_B, S_B, Vt_B, k_val)
    err_R_temp = np.linalg.norm(R - R_k_temp, ord='fro')
    err_G_temp = np.linalg.norm(G - G_k_temp, ord='fro')
    err_B_temp = np.linalg.norm(B - B_k_temp, ord='fro')

    err_total_temp = np.sqrt(err_R_temp**2 + err_G_temp**2 + err_B_temp**2)
    errors.append(err_total_temp)
```

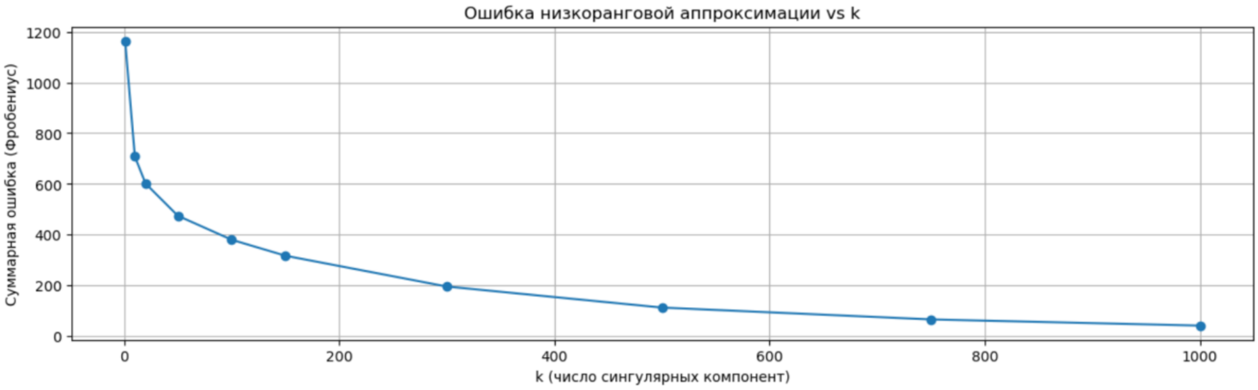

Рисунок 8. График отображающий ошибку низкоранговой аппроксимации от k сингулярных компонент.

Ошибка Фробениуса:

$$\mathrm{err}_{\mathrm{F}}^{(\mathrm{N})} = |\, A - A_{\text{rank-1}}^{(N)}|_F$$

где $A_{\text{rank-1}}^{(N)} = \sigma^{(N)}\, u^{(N)} \left(v^{(N)}\right)^T$

Сравнение с методом Якоби

```
def jacobi_eigenvalues(M, tol=1e-8, max_iter=500):
    A = M.copy()
    n = A.shape[0]
    dominant_eig_history = []
    def max_offdiag(A):
        max_val = 0.0
        p = 0; q = 1
        for i in range(n):
            for j in range(i+1, n):
                if abs(A[i, j]) > max_val:
                    max_val = abs(A[i, j]); p, q = i, j
        return p, q, max_val

    for iteration in range(max_iter):
        p, q, max_val = max_offdiag(A)
        if max_val < tol:
            break
        if A[p, p] == A[q, q]:
            phi = np.pi / 4
        else:
            phi = 0.5 * np.arctan2(2*A[p, q], (A[q, q] - A[p, p]))
        c = np.cos(phi); s = np.sin(phi)

        Ap = A.copy()
        for i in range(n):
            if i not in (p, q):
                Ap[i, p] = c*A[i, p] - s*A[i, q]; Ap[p, i] = Ap[i, p]
                Ap[i, q] = s*A[i, p] + c*A[i, q]; Ap[q, i] = Ap[i, q]
        Ap[p, p] = c*c*A[p, p] - 2*s*c*A[p, q] + s*s*A[q, q]
        Ap[q, q] = s*s*A[p, p] + 2*s*c*A[p, q] + c*c*A[q, q]
        Ap[p, q] = 0.0; Ap[q, p] = 0.0
        A = Ap.copy()

        dominant_eig_history.append(np.max(np.diag(A)))

    eigenvalues = np.diag(A)
    return eigenvalues, dominant_eig_history
```

Метод вращений Якоби

Предназначен для нахождения всех собственных значений симметричной матрицы $M \in \mathbb{R}^{n\times n}$.

Алгоритм:

1. Находят максимальный по модулю вне-диагональный элемент $M_{pq}, p < q$.
2. Вычисляют угол φ:

$$\varphi = \frac{\pi}{4}, \text{ если } M_{pp} = M_{qq},$$

$$\varphi = \left(\frac{1}{2}\right) \arctan\left(\frac{2M_{pq}}{(M_{qq}-M_{pp})}\right), \text{ иначе.}$$

3. Формируют матрицу поворота $J(p, q, \varphi)$, обновляют $M \coloneqq J^T M J$ — этот шаг «обнуляет» элемент M_{pq}.
4. Повторяют, пока $\max_{i<j}|M_{ij}| <$ tol. В конце $M \sim \mathrm{diag}(\lambda_1, \dots, \lambda_n)$.

Сложность шага: поиск max вне-диагонального $O(n^2)$,обновление строк и столбцов O(n).

Обычно требуется $O(n^2)$, итераций $\rightarrow O(n^4)$, но на практике $\sim O(n^3)$.

Применение к каналу R

```
M_R = R.T @ R
eigvals_R_jacobi, history_R_jacobi = jacobi_eigenvalues(M_R, tol=1e-10,
max_iter=500)
singulars_R_jacobi = np.sort(np.sqrt(np.abs(eigvals_R_jacobi)))[::-1]
```

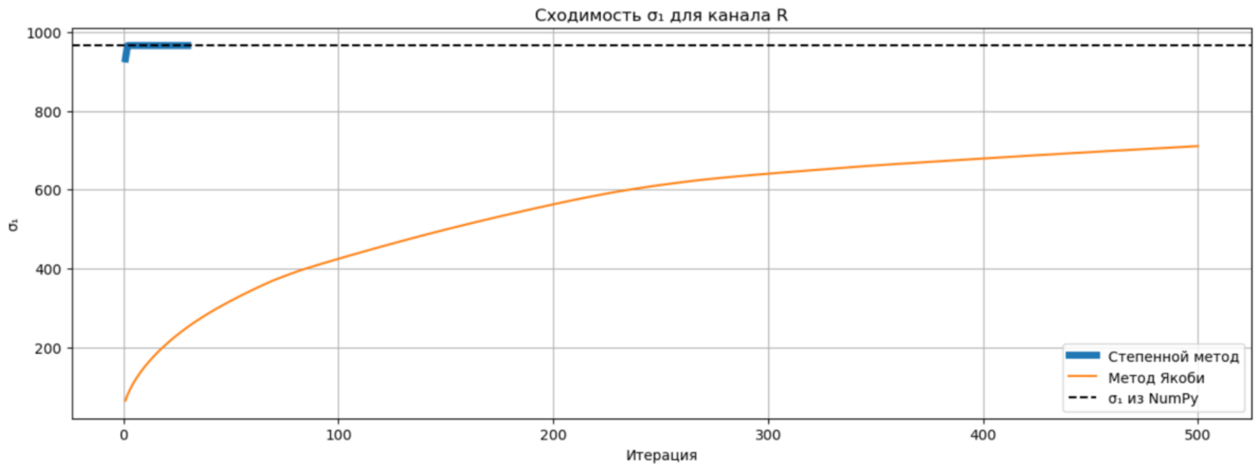

Рисунок 8. График сходимости σ_1 для канала R. Сравнение Степенного метода и метода Якоби

Мы видим, что Степенной метод на 30 итерации сравнялся с σ_1 из NumPy, когда метод Якоби не достиг этого даже на 500 итерации. Следовательно Степенной метод эффективнее для нахождения σ_1.

Гистограммы глубины цвета:

```
R_255 = (R * 255).astype(np.uint8)
G_255 = (G * 255).astype(np.uint8)
B_255 = (B * 255).astype(np.uint8)

fig, axes = plt.subplots(1, 3, figsize=(15,4))
axes[0].hist(R_255.ravel(), bins=256, color='red', alpha=0.7)
axes[0].set_title(f'R: mean={R_255.mean():.1f}, std={R_255.std():.1f}')
axes[0].set_xlim(0,255)

axes[1].hist(G_255.ravel(), bins=256, color='green', alpha=0.7)
axes[1].set_title(f'G: mean={G_255.mean():.1f}, std={G_255.std():.1f}')
axes[1].set_xlim(0,255)

axes[2].hist(B_255.ravel(), bins=256, color='blue', alpha=0.7)
axes[2].set_title(f'B: mean={B_255.mean():.1f}, std={B_255.std():.1f}')
axes[2].set_xlim(0,255)

plt.tight_layout()
plt.show()
```

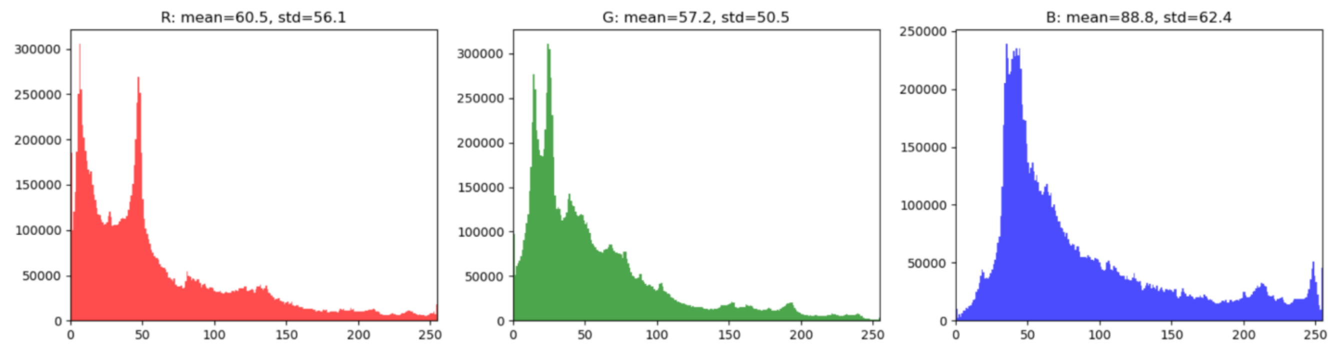

Рисунок 9. Гистограммы глубины цвета

Широкие гистограммы соответствуют большому разбросу яркостей, что влияет на выбор числа k в усечённом SVD.

В данной работе были реализованы и исследованы следующие методы:

1. Степенной метод для вычисления наибольшего сингулярного числа $\sigma 1$ и векторов $u1$, $v1$.
   • Показано, что уже при $N \approx 10$ итерациях ошибка rank-1 приближения близка к оптимальной.
   • Основная затрата: $O(K \cdot m \cdot n)$, где K — число итераций.

2. Метод вращений Якоби для нахождения всех собственных значений $M = A^T A \rightarrow$ всех. σ_i
   • Сравнение первых 15 σ_i не показало точного совпадения с NumPy
   • Якоби требует порядка $O(n^3)$ операций, что существенно больше, чем степенной метод для одного $\sigma 1$.

3. Полный SVD (NumPy) для каналов R, G, B:
   • Анализ распределения σ_i (рис. 4) показал, что σ_iбыстро убывают.
   • Усечённый SVD с k = 100 обеспечивает визуально приемлемое восстановление (рис. 6), а с k = 500 – даёт неотличимый от оригинала результат (рис. 7).
При этом суммарная ошибка $\| A - A_k \|_F$ уменьшается с ростом k.

4. Построены гистограммы глубины цвета (рис. 9), вычислены средние и стандартные отклонения.
   • Гистограммы показывают разброс яркостей в каждом канале, что позволяет обосновать количество необходимых σ_i для низкорангового приближения.