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 «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ

по рубежному контролю
по курсу «Анализ алгоритмов»
на тему: «Сингулярное разложение»

Студент ИУ7-54Б (Группа) _______ (Подпись, дата) Разин А. (И. О. Фамилия)

Преподаватель _______ (Подпись, дата) Волкова Л. Л. (И. О. Фамилия)

2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Во многих областях человеческой деятельности информацию часто представляют в форме матриц. Матрица — это регулярный числовой массив. В некоторых задачах необходимо уменьшить число хранимых данных, не потеряв важную информацию из матрицы. При анализ данных для снижения размерности хранимых данных и понижения «шума» возможно использование сингулярного разложения матриц (SVD, от англ. Singular Value Decomposition) [1].

Целью данной лабораторной работы является описание алгоритма сингулярного разложения и исследование его трудоемкости.

Для поставленной цели необходимо выполнить следующие задачи.

1. Описать алгоритм сингулярного разложения.
2. Разработать алгоритм сингулярного разложения.
3. Создать программное обеспечение, реализующее алгоритм сингулярного разложения.
4. Оценить трудоемкость реализации сингулярного разложения.

1 Аналитическая часть

В данной части работы будет описан алгоритм сингулярного разложения матриц.

1.1 Сингулярное разложение матриц

Сингулярное разложение обозначают $SVD(A)$, опишем его в виде равенства [1]:

$$A = USV^T. \tag{1.1}$$

Матрица S всегда диагональная, ее коэффициенты —- неотрицательные вещественные числа $\sigma_1, \cdots, \sigma_n$, расположенные на главной диагонали матрицы, называются сингулярными числами, сингулярные числа являются корнем из собственных чисел матриц. Столбцы матрицы V называются правыми сингулярными векторами и всегда ортогональны друг другу. Столбцы матрицы U называются левыми сингулярными векторами и также ортогональны друг другу. Матрицы U и V являются унитарными, т. е сумма квадратов значений каждого столбца матриц равняется единице. Их можно использовать в качестве нового базиса системы координат для представления данных, записанных в матрице A [1].

1.2 Алгоритм сингулярного разложения матриц

, Необходимо получить сингулярные числа некоторый матрицы, являющейся произведением исходной матрицы A и ее транспонированной копии A^T. После чего необходимо найти собственные числа результата произведения, из которых будут получены сингулярные числа [1; 2].

После того как сингулярные числа получены возможно получение матрицы S, путем расстановки сингулярных чисел по диагонали матрицы в порядке убывания [3].

Правые сингулярные векторы будут получены с помощью полученных сингулярных чисел и матрицы AA^T, пусть дано сингулярное число σ_1 и матрица AA^T см. выражение (1.2). После вычисления значения выражения (1.3), будет получена матрица коэффициентов системы для 3 неизвестных (1.4). Поле ее решения будет получен правый сингулярный вектор для сингулярного числа σ_1 ([3]).

$$\sigma_1 = 15.4, A^T A = \begin{pmatrix} 10 & 5 & 2 \\ 5 & 6 & 5 \\ 2 & 5 & 5 \end{pmatrix} \tag{1.2}$$

$$A^T A - \sigma_1 I = \begin{pmatrix} 10 - 15.4 & 5 & 2 \\ 5 & 6 - 15.4 & 5 \\ 2 & 5 & 5 - 15.4 \end{pmatrix} \tag{1.3}$$

$$\begin{pmatrix} -5.43 - 15.4 & 5 & 2 \\ 5 & 6 - 9.43 & 5 \\ 2 & 5 & 5 - 10.43 \end{pmatrix} = 0 \tag{1.4}$$

После получения сингулярных векторов, необходимо их нормализовать и построчно записать в матрицу, таким образом получая матрицу V.

После получения S,V, получение левых сингулярных векторов возможно с использованием выражения (1.5). Таким образом все составляющие сингулярного разложения были получены [3].

$$U = AV^T S^{-1} \tag{1.5}$$

Вывод В данной части было рассмотрено представление сингулярного разложения и алгоритм его получения.

2 Конструкторская часть

В данной части работы будут рассмотрены схемы алгоритмов сортировок, а также приведен расчет их трудоемкости.

2.1 Схемы алгоритмов

На рисунке 2.1 приведены схемы алгоритма сингулярного разложения.

2.2 Оценка трудоемкости реализации сингулярного разложения

2.2.1 Расчет транспонированной матрицы

Матрица размерами $N \times M$.

$$f_{matrix\_transpose} = f_{resize} + f_{outer\_for}. \tag{2.1}$$

$$f_{outer\_for} = 3 + N \cdot (f_{resize} + 3 + M \cdot 5)). \tag{2.2}$$

Тогда:

$$\begin{gathered} f_{matrix\_transpose} = f_{resize} + 3 + N \cdot (f_{resize} + 3 + 5M) = f_{resize} + 3N \cdot f_{resize} + \\ +9N + 15MN = f_{resize} \cdot (3N + 1) + 9N + 15MN. \end{gathered} \tag{2.3}$$

2.2.2 Расчет произведения матриц

Матрицы размерами $N \times M$ и $M \times K$.

$$f_{matrix\_by\_matrix} = f_{resize} + f_{1\_outer\_for}. \tag{2.4}$$

$$f_{1\_outer\_for} = 3 + N \cdot (f_{resize} + f_{2\_outer\_for}). \tag{2.5}$$

$$f_{2\_outer\_for} = 3 + M \cdot (3 + f_{inner\_for}). \tag{2.6}$$

$$f_{inner\_for} = 3 + M \cdot (2 + 1 + 2 + 2 + 2) = 3 + 9M. \tag{2.7}$$

Тогда:

$$f_{matrix\_by\_matrix} = f_{resize} + f_{1\_outer\_for} = f_{resize} + 3\cdot$$
$$\cdot N(f_{resize} + 3 + M \cdot (3 + 3 + 9M)) = f_{resize} + 3 \cdot N(f_{resize} + 3+$$
$$+6M + 9M^2) = f_{resize} + 3N \cdot f_{resize} + 9N + 18NM + 27NM^2 =$$
$$= f_{resize} \cdot (3N + 1) + 9N + 18NM + 27NM^2. \tag{2.8}$$

2.3 Получение эрмитовой матрицы

Размер вектора равен N;

$$f_{get\_hermitian\_matrix} = f_{resize} + f_{for\_1} + (2 + 1 + 2 + 1) + f_{for\_2} + f_{for\_3}. \tag{2.9}$$

$$f_{for\_1} = 3 + N \cdot (1 + f_{resize}) = 3 + N + N \cdot f_{resize}. \tag{2.10}$$

$$f_{for\_2} = 3 + (N - 1) \cdot (2 + 1 + 1 + 2 + 1) = 3 + (N - 1) \cdot 7 =$$
$$= 3 + 7N - 7 = 7N - 4. \tag{2.11}$$

$$f_{for\_3} = 3 + N \cdot (2 + 1) = 3 + 3N. \tag{2.12}$$

Тогда:

$$f_{get\_hermitian\_matrix} = f_{resize} + f_{for\_1} + (2 + 1 + 2 + 1)+$$
$$+f_{for\_2} + f_{for\_3} =$$
$$= f_{resize} + 3 + N + N \cdot f_{resize} + 6 + 7N - 4 + 3 + 3N =$$
$$= f_{resize} \cdot (1 + N) + 8 + 11N. \tag{2.13}$$

2.4 Получение обратной эрмитовой матрицы

Имеет аналогичную трудоемкость, что и $get\_hermitian\_matrix$ в силу смены порядка операций.

2.5 Решение системы с помощью метода Гаусса-Жордана

Размер матрицы $N \times M$, размер вектора K.

$$f_{jordan\_gaussian\_transform} = f_{for} + f_{while} + 3. \tag{2.14}$$

$$f_{for} = 3 + (N-1) \cdot (6 + f_{inner\_for\_1} + f_{inner\_for\_2}). \tag{2.15}$$

$$f_{inner\_for\_1} = 3 + M \cdot (2 + 2 + 2 + f_{swap}) = 3 + 6M + M \cdot f_{swap}. \tag{2.16}$$

$$\begin{aligned} f_{inner\_for\_2} = 3 + N \cdot (1 + 2 + 3 + M \cdot (2 + 2 + 1 + 2 + 1 + 2 + 2)) = \\ = 3 + N \cdot (6 + 12M) = 3 + 6N + 12NM. \end{aligned} \tag{2.17}$$

$$f_{while} = N \cdot (2 + 1 + 2 + 1) = 6N. \tag{2.18}$$

Тогда:

$$\begin{gathered} f_{jordan\_gaussian\_transform} = 3 + (N-1) \cdot (6 + 3 + 6M + M \cdot f_{swap} + \\ +3 + 6N + 12NM) + 6N + 3 = 6 + 6N + (N-1) \cdot (12 + 6M + \\ +M \cdot f_{swap} + 6N + 12NM) = 6 + 6N + 12N - 12 + 6NM - 6M + \\ +M \cdot f_{swap} \cdot (N-1) + 6N^2 - 6N + 12N^2M - 12NM = \\ = -6 + 12N - 6M - 6NM + 6N^2 + 12N^2M + M \cdot f_{swap} \cdot (N-1). \end{gathered} \tag{2.19}$$

2.6 Получение обратной диагональной матрицы

Размер матрицы $N \times M$.

$$
\begin{gathered}
f_{get\_inverse\_diagonal\_matrix} = \\
= f_{resize} + 3 + N \cdot (2 + f_{resize} + 2 + 1 + 1 + 2) = \\
= f_{resize} + 3 + N \cdot (8 + f_{resize}) = \\
= f_{resize} \cdot (1 + N) + 3 + 8N.
\end{gathered} \tag{2.20}
$$

2.7 Получение матриц меньшей размерности

Размер матрицы $N \times M$.

$$
f_{get\_reduced\_matrix\_best} = 1 + 2 \cdot f_{resize} + 1 + 2 + 1 + 2 = 7 + 2 \cdot f_{resize}. \tag{2.21}
$$

$$
\begin{gathered}
f_{get\_reduced\_matrix\_worst} = 1 + f_{resize} + 2 + 2 + N \cdot (1 + f_{resize} + 2+ \\
+M \cdot (2 + 1 + 1 + 2 + 1)) = 1 + f_{resize} + 4 + N \cdot (3 + f_{resize}+ \\
+7M) = 5 + f_{resize} + 3N + N \cdot f_{resize}. + 7NM = f_{resize} \cdot (1 + N) + 5+ \\
+3N + 7NM.
\end{gathered} \tag{2.22}
$$

2.8 Получение сингулярных чисел и векторов

Размер матрицы $N \times M$, размер вектора K.

$$
\begin{gathered}
f_{compute\_evd} = f_{first\_if} + 3 + N + 1 + 2 + J \cdot (11 + N + N \cdot (3 + 2+ \\
+N \cdot 8) + 2 + N \cdot 4 + 1 + f_{sec\_if} + 1) + f_{third\_if} + f_{fourth\_if} = \\
= f_{first_if} + 6 + N + J \cdot (11 + N + N \cdot (5 + 8N) + 4 + 4N) + f_{third\_if}+ \\
+f_{fourth\_if}.
\end{gathered} \tag{2.23}
$$

$$
\begin{gathered}
f_{first\_if\_best} = 2; \\
f_{first\_if\_worst} = 2 + 1 = 3.
\end{gathered} \tag{2.24}
$$

$$
\begin{gathered}
f_{sec\_if\_best} = 1; \\
f_{sec\_if\_worst} = 1 + 8 + 4 = 13.
\end{gathered} \tag{2.25}
$$

$$
\begin{gathered}
f_{third\_if\_best} = 1;\\
f_{third\_if\_worst} = 1 + 2 + N \cdot (1 + 2 + 8M) +\\
+ f_{jordan\_gaussian\_transform} + f_{get\_hermitian\_matrix} +\\
+2 \cdot f_{matrix\_by\_matrix} + f_{get\_hermitian\_matrix\_inverse} +\\
+ f_{get\_reduced\_matrix} = 3 + 3N + 8NM - 6 + 12N - 6M - 6NM +\\
+6N^2 + 12N^2M + M \cdot f_{swap} \cdot (N-1) + f_{resize} \cdot (1+N) + 8 + 11N +\\
+2 \cdot f_{resize} * (3N+1) + 18N + 36NM + 54NM^2 +\\
+ f_{resize} \cdot (1+N) + 8 + 11N + f_{resize} \cdot (1+N) + 5 + 3N + 7NM =\\
= 18 + 58N - 6M + 45NM + 6N^2 + 12N^2M + 54NM^2 +\\
+M \cdot f_{swap} \cdot (N-1) + f_{resize} \cdot (9N+5).
\end{gathered} \tag{2.26}
$$

$$
\begin{gathered}
f_{fourth\_if\_best} = 1;\\
f_{fourth\_if\_worst} = 2 + K \cdot (4 + N \cdot (3 + 6N) +\\
+ f_{jordan\_gaussian\_transform} + 9 + 7N) = 2 + K \cdot (13 + 10N + 6N^2 -\\
-6 + 12N - 6M - 6NM + 6N^2 + 12N^2M + M \cdot f_{swap} \cdot (N-1)) =\\
= 2 + K(7 + 22N - 6M - 6NM + 12N^2 + 12N^2M + M \cdot f_{swap} \cdot\\
\cdot (N-1)) = 2 + 7K + 22KN - 6KM - 6KNM = 12KN^2 +\\
+12KN^2M + KM \cdot f_{swap} \cdot (N-1).
\end{gathered} \tag{2.27}
$$

Т. о. в худшем и лучшем случаях имеем:

$$
\begin{gathered}
f_{compute\_evd\_best} = f_{first\_if} + 6 + N + J \cdot (11 + N + N \cdot (5 + 8N) +\\
+4 + 4N + f_{sec\_if}) + f_{third\_if} +\\
+ f_{fourth\_if} = 2 + 6 + N + J \cdot (15 + 10N + 8N^2 + 1) + 1 + 1 =\\
= 10 + N + 16J + 10JN + 8JN^2.
\end{gathered} \tag{2.28}
$$

$$
\begin{gathered}
f_{compute\_evd\_worst} = f_{first\_if} + 6 + N + J \cdot (11 + N + N \cdot (5 + 8N) + \\
+ 4 + 4N + f_{sec\_if}) + f_{third\_if} + f_{fourth\_if} = 3 + 6 + N + J \cdot (15 + \\
+ 10N + 8N^2 + 13) + 18 + 58N - 6M + 45NM + 6N^2 + 12N^2M + \\
+ 54NM^2 + M \cdot f_{swap} \cdot (N - 1) + f_{resize} \cdot (9N + 5) + 12KN^2 + \\
+ 12KN^2M + KM \cdot f_{swap} \cdot (N - 1) = 27 + 59N - 6M + \\
+ 12KN^2M + KM \cdot f_{swap} \cdot (N - 1) + 45NM + 54NM^2 + 12KN^2 + \\
+ f_{resize} \cdot (9N + 5).
\end{gathered} \tag{2.29}
$$

2.9 Сингулярное разложение

Размер матрицы $N \times M$, размер вектора K.

$$
\begin{gathered}
f_{svd} = 2 \cdot f_{matrix\_transpose} + 4 \cdot f_{matrix\_by\_matrix} + \\
+ f_{compute\_evd} + 2 + 5K + f_{get\_inverse\_diagonal\_matrix}.
\end{gathered} \tag{2.30}
$$

$$
\begin{gathered}
f_{svd\_best} = 2 \cdot f_{resize} \cdot (3N + 1) + 18N + 30MN + 4 \cdot f_{resize} \cdot \\
\cdot (3N + 1) + 36N + 72NM + 108NM^2 + 10 + N + \\
+ 16J + 10JN + 8JN^2 + 2 + 5K + f_{resize} \cdot (1 + N) + 3 + 8N = \\
= 15 + 63N + 102NM + 16J + 10JN + 8JN^2 + 5K + \\
+ f_{resize} \cdot (7 + 19N).
\end{gathered} \tag{2.31}
$$

$$
\begin{gathered}
f_{svd\_worst} = 2 \cdot f_{resize} \cdot (3N + 1) + 18N + 30MN + 4 \cdot \\
\cdot f_{resize} \cdot (3N + 1) + 36N + 72NM + 108NM^2 + \\
+ 27 + 59N - 6M + 12KN^2M + \\
+ KM \cdot f_{swap} \cdot (N - 1) + 45NM + 54NM^2 + 12KN^2 + \\
+ f_{resize} \cdot (9N + 5) + 2 + 5K + f_{resize} \cdot (1 + N) + 3 + \\
+ 8N = f_{resize} \cdot (7 + 19N) + 32 + 113N + 147NM + 162NM^2 + \\
+ 12KN^2 + 5K + KM \cdot f_{swap} \cdot (N - 1) + 12KN^2M.
\end{gathered} \tag{2.32}
$$

Вывод

В данном разделе была построена схема алгоритма расчета сингулярного разложения и была выведена трудоемкость для лучшего и худшего случаев сингулярного разложения.

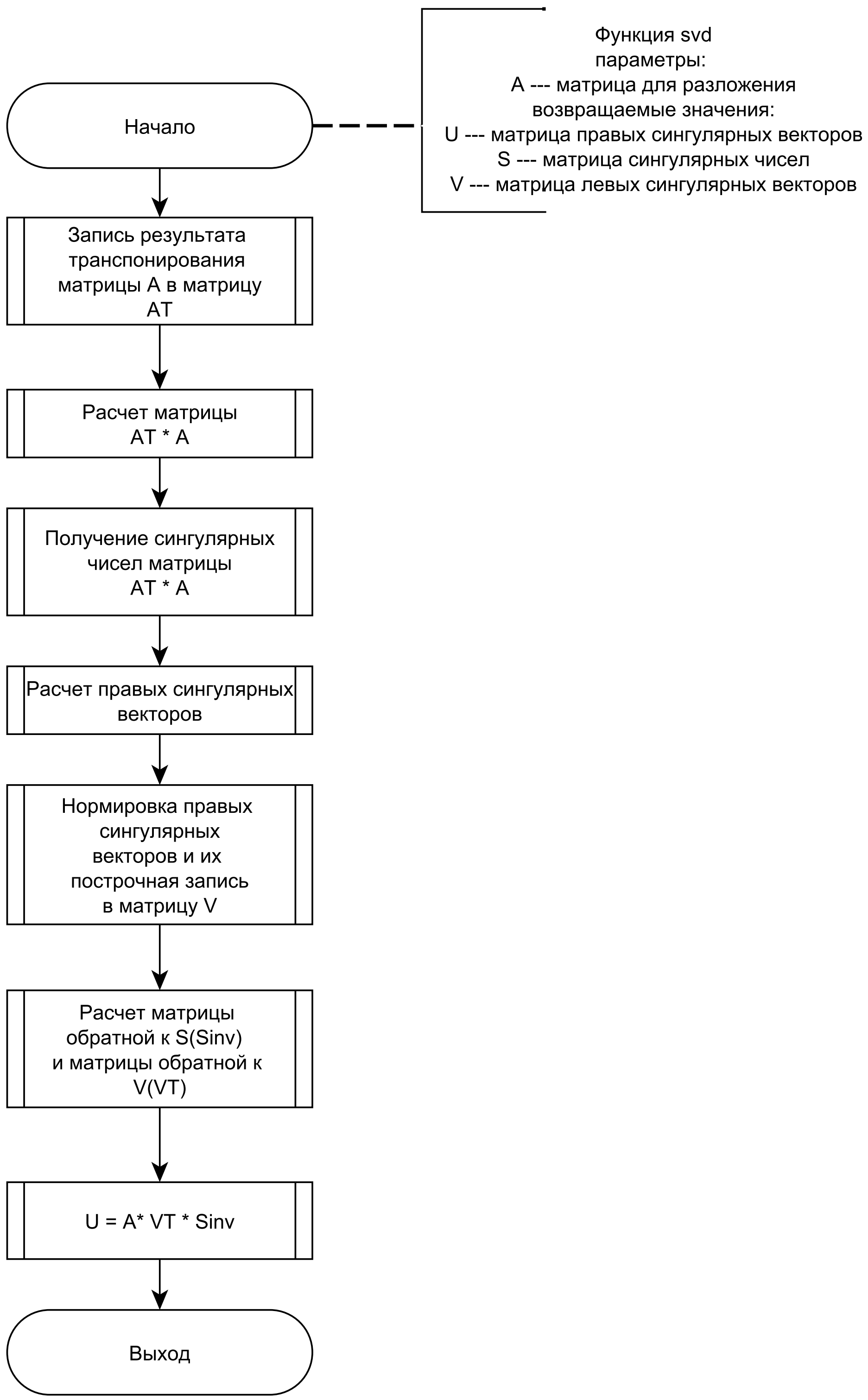

Рисунок 2.1 – Схема алгоритма сингулярного разложения

3 Технологический раздел

В данной части работы будут описаны средства реализации программы, а также листинги, модульные и функциональные тесты.

3.1 Средства реализации

Алгоритмы для данной лабораторной работы были реализованы на языке C++, при использовании компилятора gcc версии 10.5.0, так как в стандартной библиотеке приведенного языка присутствует класс `vector` упрощающий реализацию матриц [4].

3.2 Реализация алгоритмов

Листинги исходных кодов программ А.1–А.4 приведены в приложении.

3.3 Тестирование

Функциональные тесты рассмотрены в таблице 3.1. Столбцы A,S,V,D обозначают соответствующие матрицы в формуле сингулярного разложения, все тесты были пройдены успешно.

Таблица 3.1 – Функциональные тесты для сингулярного разложения матрицы

A	**S**	**V**	**D**
$\begin{pmatrix}1.0\end{pmatrix}$	$\begin{pmatrix}1.0\end{pmatrix}$	$\begin{pmatrix}1.0\end{pmatrix}$	$\begin{pmatrix}1.0\end{pmatrix}$
$\begin{pmatrix}1.0 & 2.0\\ 3.0 & 4.0\end{pmatrix}$	$\begin{pmatrix}5.4 & 0.0\\ 0.0 & 4.0\end{pmatrix}$	$\begin{pmatrix}0.0 & 0.9\\ 0.9 & -0.4\end{pmatrix}$	$\begin{pmatrix}0.5 & -0.8\\ 0.8 & 0.5\end{pmatrix}$
$\begin{pmatrix}-1.0 & 2.0 & 3.0\\ -4.0 & 5.0 & 6.0\\ 7.0 & 8.0 & 9.0\end{pmatrix}$	$\begin{pmatrix}15.2 & 0.0 & 0.0\\ 0.0 & 7.1 & 0.0\\ 0.0 & 0.0 & 0.3\end{pmatrix}$	$\begin{pmatrix}0.2 & 0.2 & 0.9\\ 0.4 & 0.8 & -0.3\\ 0.8 & -0.4 & -0.1\end{pmatrix}$	$\begin{pmatrix}0.2 & -0.9 & 0.1\\ 0.6 & 0.1 & -0.7\\ 0.7 & 0.2 & 0.6\end{pmatrix}$
$\begin{pmatrix}-1.0\end{pmatrix}$	$\begin{pmatrix}1.0\end{pmatrix}$	$\begin{pmatrix}-1.0\end{pmatrix}$	$\begin{pmatrix}1.0\end{pmatrix}$

Вывод

В данной части работы были представлены листинги реализованных алгоритмов и тесты, успешно пройденные программой.

4 Исследовательская часть

В данном разделе приведены примеры работы реализации алгоритма сингулярного разложения.

4.1 Демонстрация работы программы

```
Введите размерность квадартной матрицы:
3
Введите элементы матрицы:
1 2 3
4 -1 2
3 0 0

S =
5.6815 0 0
0 3.2262 0
0 0 1.1457



U =
0.41842 0.89379 -0.16148
0.78422 -0.26583 0.56065
0.45817 -0.36122 -0.81216



V =
0.8677 -0.38845 -0.31016
0.0092597 0.63648 -0.77123
0.497 0.66633 0.55588
```

Рисунок 4.1 – Разложение матрицы 3×3

```
Введите размерность квадартной матрицы:2
Введите элементы матрицы:
1 2
3 4

S =
5.465 0
0 0.36597


U =
0.40455 0.91451
0.91451 -0.40455


V =
0.57605 -0.81742
0.81742 0.57605
```

Рисунок 4.2 – Разложение матрицы 2×2

```
Введите размерность квадартной матрицы:2
Введите элементы матрицы:
-1000 2
11000 3

S =
11045 0
0 2.2361


U =
-0.090536 1.0081
0.99589 0.091642


V =
1 -0.0002541
0.00025806 1
```

Рисунок 4.3 – Разложение матрицы 2×2 с отрицательными числами

ЗАКЛЮЧЕНИЕ

Поставленная цель: описание алгоритма сингулярного разложения и исследование его трудоемкости, была достигнута.

Для поставленной цели были выполнены все поставленные задачи.

1. Описать алгоритм сингулярного разложения.
2. Разработать алгоритм сингулярного разложения.
3. Создать программное обеспечение, реализующее алгоритм сингулярного разложения.
4. Оценить трудоемкость реализации сингулярного разложения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Сингулярное разложение матриц [Электронный ресурс]. — Режим доступа: https://inf.grid.by/jour/issue/viewFile/19/20 (дата обращения: 28.09.2023).
2. Сингулярное разложение [Электронный ресурс]. — Режим доступа: http://www.machinelearning.ru/wiki/index.php (дата обращения: 24.12.2023).
3. Singular Values Decomposition [Электронный ресурс]. — Режим доступа: https://www.codeproject.com/Articles/1268576/Singular-Values-Decomposition-SVD-in-Cplusplus11-b (дата обращения: 28.09.2023).
4. std::vector [Электронный ресурс]. — Режим доступа: https://en.cppreference.com/w/cpp/container/vector (дата обращения: 28.09.2023).

ПРИЛОЖЕНИЕ А

Листинг А.1 – Реализация алгоритма сингулярного разложения

```
void svd(std::vector<std::vector<float>> matrix, std::vector<std::vector<float>>& s,
    std::vector<std::vector<float>>& u, std::vector<std::vector<float>>& v)
{
    std::vector<std::vector<float>> matrix_t;
    matrix_transpose(matrix, matrix_t);

    std::vector<std::vector<float>> matrix_product1;
    matrix_by_matrix(matrix, matrix_t, matrix_product1);

    std::vector<std::vector<float>> matrix_product2;
    matrix_by_matrix(matrix_t, matrix, matrix_product2);

    std::vector<std::vector<float>> u_1;
    std::vector<std::vector<float>> v_1;

    std::vector<float> eigenvalues;
    compute_evd(matrix_product2, eigenvalues, v_1, 0);

    matrix_transpose(v_1, v);

    s.resize(matrix.size());
    for (std::size_t index = 0; index < eigenvalues.size(); index++)
    {
        s[index].resize(eigenvalues.size());
        s[index][index] = eigenvalues[index];
    }

    std::vector<std::vector<float>> s_inverse;
    get_inverse_diagonal_matrix(s, s_inverse);

    std::vector<std::vector<float>> av_matrix;
    matrix_by_matrix(matrix, v, av_matrix);
    matrix_by_matrix(av_matrix, s_inverse, u);
}
```

Листинг А.2 – Реализация алгоритма решения систем уравнений методом Гаусса

```
void jordan_gaussian_transform(
    std::vector<std::vector<float>> matrix, std::vector<float>&
       eigenvector)
{
    const float eps = 10e-6; bool eigenv_found = false;
    for (std::size_t s = 0; s < matrix.size() - 1 &&
       !eigenv_found; s++)
    {
        std::size_t col = s; float alpha = matrix[s][s];
        while (col < matrix[s].size() && alpha != 0 && alpha !=
           1)
            matrix[s][col++] /= alpha;

        for (std::size_t c = s; col < matrix[s].size() &&
           !alpha; col++)
            std::swap(matrix[s][col], matrix[s + 1][col]);

        for (std::size_t row = 0; row < matrix.size(); row++)
        {
            float gamma = matrix[row][s];
            for (std::size_t c = s; c < matrix[row].size() &&
               row != s; c++)
                matrix[row][c] = matrix[row][c] - matrix[s][c] *
                   gamma;
        }
    }

    std::size_t row = 0;
    while (row < matrix.size())
        eigenvector.push_back(-matrix[row++][matrix.size() - 1]);

    eigenvector[eigenvector.size() - 1] = 1;
}
```

Листинг А.3 – Реализация поиска сингулярных векторов и чисел матрицы

```
std::vector<std::vector<float>> matrix_i;


void compute_evd(std::vector<std::vector<float>> matrix,
```

```
    std::vector<float>& eigenvalues,
       std::vector<std::vector<float>>& eigenvectors,
       std::size_t eig_count)
{
    std::size_t m_size = matrix.size();
    std::vector<float> vec; vec.resize(m_size);
    std::generate(vec.begin(), vec.end(), []() {
      return rand() % 100;
    });

    if (eigenvalues.size() == 0 && eigenvectors.size() == 0)
    {
        eigenvalues.resize(m_size);
        eigenvectors.resize(eigenvalues.size());
        matrix_i = matrix;
    }

    std::vector<std::vector<float>> m; m.resize(m_size);
    for (std::size_t row = 0; row < m_size; row++)
        m[row].resize(100);

    float lambda_old = 0;

    std::size_t index = 0; bool is_eval = false;
    while (is_eval == false)
    {
        for (std::size_t row = 0; row < m_size && (index % 100)
           == 0; row++)
            m[row].resize(m[row].size() + 100);

        for (std::size_t row = 0; row < m_size; row++)
        {
            m[row][index] = 0;
            for (std::size_t col = 0; col < m_size; col++)
                m[row][index] += matrix[row][col] * vec[col];
        }

        for (std::size_t col = 0; col < m_size; col++)
            vec[col] = m[col][index];

        if (index > 0)
```

```
        {
            float lambda = ((index - 1) > 0) ? \
                (m[0][index] / m[0][index - 1]) : m[0][index];
            is_eval = (fabs(lambda - lambda_old) <
               /*10e-15*/10e-10);

            eigenvalues[eig_count] = lambda; lambda_old = lambda;
        }

        index++;
    }

    std::vector<std::vector<float>> matrix_new;

    if (m_size > 1)
    {
        std::vector<std::vector<float>> matrix_tfloatet;
        matrix_tfloatet.resize(m_size);

        for (std::size_t row = 0; row < m_size; row++)
        {
            matrix_tfloatet[row].resize(m_size);
            for (std::size_t col = 0; col < m_size; col++)
                matrix_tfloatet[row][col] = (row == col) ? \
                (matrix[row][col] - eigenvalues[eig_count]) :
                   matrix[row][col];
        }

        std::vector<float> eigenvector;
        jordan_gaussian_transform(matrix_tfloatet, eigenvector);

        std::vector<std::vector<float>> hermitian_matrix;
        get_hermitian_matrix(eigenvector, hermitian_matrix);

        std::vector<std::vector<float>> ha_matrix_product;
        matrix_by_matrix(hermitian_matrix, matrix,
           ha_matrix_product);

        std::vector<std::vector<float>> inverse_hermitian_matrix;
        get_hermitian_matrix_inverse(eigenvector,
           inverse_hermitian_matrix);
```

```

        std::vector<std::vector<float>> iha_matrix_product;
        matrix_by_matrix(ha_matrix_product, inverse_hermitian_matrix, iha_matrix_product);

        get_reduced_matrix(iha_matrix_product, matrix_new, m_size - 1);
    }

    if (m_size <= 1)
    {
        for (std::size_t i = 0; i < eigenvalues.size(); i++)
        {
            float lambda = eigenvalues[i];
            std::vector<std::vector<float>> matrix_tfloatet;
            matrix_tfloatet.resize(matrix_i.size());

            for (std::size_t row = 0; row < matrix_i.size(); row++)
            {
                matrix_tfloatet[row].resize(matrix_i.size());
                for (std::size_t col = 0; col < matrix_i.size(); col++)
                    matrix_tfloatet[row][col] = (row == col) ? \
                    (matrix_i[row][col] - lambda) : matrix_i[row][col];
            }

            eigenvectors.resize(matrix_i.size());
            jordan_gaussian_transform(matrix_tfloatet, eigenvectors[i]);

            float eigsum_sq = 0;
            for (std::size_t v = 0; v < eigenvectors[i].size(); v++)
                eigsum_sq += pow(eigenvectors[i][v], 2.0);

            for (std::size_t v = 0; v < eigenvectors[i].size(); v++)
                eigenvectors[i][v] /= sqrt(eigsum_sq);

```

```
            eigenvalues[i] = sqrt(eigenvalues[i]);
        }

        return;
    }

    compute_evd(matrix_new, eigenvalues, eigenvectors, eig_count
       + 1);

    return;
}
```

Листинг А.4 – Реализация расчета произведения матриц и обратных матриц

```
void matrix_transpose(std::vector<std::vector<float>> matrix1,
    std::vector<std::vector<float>>& matrix2)
{
    matrix2.resize(matrix1.size());
    for (std::size_t row = 0; row < matrix1.size(); row++)
    {
        matrix2[row].resize(matrix1[row].size());
        for (std::size_t col = 0; col < matrix1[row].size();
           col++)
            matrix2[row][col] = matrix1[col][row];
    }
}

void get_hermitian_matrix(std::vector<float> eigenvector,
    std::vector<std::vector<float>>& h_matrix)
{
    h_matrix.resize(eigenvector.size());
    for (std::size_t row = 0; row < eigenvector.size(); row++)
        h_matrix[row].resize(eigenvector.size());

    h_matrix[0][0] = 1.0 / eigenvector[0];
    for (std::size_t row = 1; row < eigenvector.size(); row++)
        h_matrix[row][0] = -eigenvector[row] / eigenvector[0];

    for (std::size_t row = 1; row < eigenvector.size(); row++)
        h_matrix[row][row] = 1;
}


```

```
void get_hermitian_matrix_inverse(std::vector<float> eigenvector,
    std::vector<std::vector<float>>& ih_matrix)
{
    ih_matrix.resize(eigenvector.size());
    for (std::size_t row = 0; row < eigenvector.size(); row++)
        ih_matrix[row].resize(eigenvector.size());

    ih_matrix[0][0] = eigenvector[0];
    for (std::size_t row = 1; row < eigenvector.size(); row++)
        ih_matrix[row][0] = -eigenvector[row];

    for (std::size_t row = 1; row < eigenvector.size(); row++)
        ih_matrix[row][row] = 1;
}

void get_reduced_matrix(std::vector<std::vector<float>> matrix,
    std::vector<std::vector<float>>& r_matrix, std::size_t
       new_size)
{
    if (new_size > 1)
    {
        r_matrix.resize(new_size);
        std::size_t index_d = matrix.size() - new_size;
        std::size_t row = index_d, row_n = 0;
        while (row < matrix.size())
        {
            r_matrix[row_n].resize(new_size);
            std::size_t col = index_d, col_n = 0;
            while (col < matrix.size())
                r_matrix[row_n][col_n++] = matrix[row][col++];

            row++; row_n++;
        }
    }

    else if (new_size == 1)
    {
        r_matrix.resize(new_size);
        r_matrix[0].resize(new_size);
        r_matrix[0][0] = matrix[1][1];
    }
```

```
}
```