Лабораторная работа №4 Линейная алгебра

Статический анализ данных

Коняева Марина Александровна
НФИбд-01-21
Студ. билет: 1032217044

2024

RUDN

Цели лабораторной работы

Изучение возможностей специализированных пакетов Julia для выполнения и оценки эффективности операций над объектами линейной алгебры.

Теоретическое введение

Для вычисления нормы используется `LinearAlgebra.norm(x)`.
Евклидова норма:

$$\|\vec{X}\|_2 = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2};$$

р-норма:

$$\|\vec{A}\|_p = \left(\sum_{i=1}^{n} |a_i|^p\right)^{1/p}.$$

Евклидово расстояние между двумя векторами $\vec{X}$ и $\vec{Y}$ определяется как $||\vec{X} - \vec{Y}||_2$.

Угол между двумя векторами $\vec{X}$ и $\vec{Y}$ определяется как $\cos^{-1}\dfrac{\vec{X}^T\vec{Y}}{||\vec{X}||_2||\vec{Y}||_2}$.

В математике факторизация (или разложение) объекта — его декомпозиция (например, числа, полинома или матрицы) в произведение других объектов или факторов, которые, будучи перемноженными, дают исходный объект.

Задачи лабораторной работы

1. Используя Jupyter Lab, повторите примеры из раздела 4.2.
2. Выполните задания для самостоятельной работы (раздел 4.4).

Выполнение лабораторной работы

Поэлементные операции над многомерными массивами

1. Изучим информацию о поэлементных операциях над многомерными массивами.

2. Повторим примеры с поэлементными операциями над многомерными массивами: Для матрицы 4×3 рассмотрим поэлементные операции сложения её элементов:

```
# Массив 4x3 со случайными целыми числами (от 1 до 20):
a = rand(1:20,(4,3))
```

```
4×3 Matrix{Int64}:
 17  10  17
  1   3  11
 11  10  12
  8   9   4
```

```
# Поэлементная сумма:
sum(a)
```

```
113
```

```
# Поэлементная сумма по столбцам:
sum(a,dims=1)
```

```
1×3 Matrix{Int64}:
 37  32  44
```

```
# Поэлементная сумма по строкам:
sum(a,dims=2)
```

```
4×1 Matrix{Int64}:
 44
 15
 33
 21
```

Рис. 1: Матрица 4×3, сложения её элементов

3. Повторим примеры с поэлементными операциями над многомерными массивами: Для матрицы 4 × 3 рассмотрим поэлементные операции произведения её элементов:

```
# Поэлементное произведение:
prod(a)

36255859200

# Поэлементное произведение по столбцам:
prod(a,dims=1)

1×3 Matrix{Int64}:
 1496  2700  8976

# Поэлементное произведение по строкам:
prod(a,dims=2)

4×1 Matrix{Int64}:
 2890
   33
 1320
  288
```

Рис. 2: Матрица 4×3, произведение её элементов

4. Для работы со средними значениями можно воспользоваться возможностями пакета Statistics.

Поэлементные операции над многомерными массивами

```julia
import Pkg
Pkg.add("Statistics")
```

```
    Updating registry at `C:\Users\User\.julia\registries\General.toml`
   Resolving package versions...
    Updating `C:\Users\User\.julia\environments\v1.10\Project.toml`
  [10745b16] + Statistics v1.10.0
  No Changes to `C:\Users\User\.julia\environments\v1.10\Manifest.toml`
```

Рис. 3: Добавление пакета

5. Повторим примеры с нахождением среднего значения массива, его среднего значения по столбцам и строкам.

```julia
using Statistics
# Вычисление среднего значения массива:
mean(a)
```

```
9.416666666666666
```

```julia
# Среднее по столбцам:
mean(a,dims=1)
```

```
1×3 Matrix{Float64}:
 9.25  8.0  11.0
```

```julia
# Среднее по строкам:
mean(a,dims=2)
```

```
4×1 Matrix{Float64}:
 14.666666666666666
  5.0
 11.0
  7.0
```

Рис. 4: Среднее значение массива

Транспонирование, след, ранг, определитель и инверсия матрицы

6. Для выполнения таких операций над матрицами, как транспонирование, диагонализация, определение следа, ранга, определителя матрицы и т.п. можно воспользоваться библиотекой (пакетом) LinearAlgebra.

```julia
# Подключение пакета LinearAlgebra:
import Pkg
Pkg.add("LinearAlgebra")
using LinearAlgebra
```

```
   Resolving package versions...
    Updating `C:\Users\User\.julia\environments\v1.10\Project.toml`
  [37e2e46d] + LinearAlgebra
  No Changes to `C:\Users\User\.julia\environments\v1.10\Manifest.toml`
```

Рис. 5: Добавление пакета

7. Повторим пример создание массива 4x4 со случайными целыми числами (от 1 до 20).

```julia
# Массив 4x4 со случайными целыми числами (от 1 до 20):
b = rand(1:20,(4,4))
```

```
4×4 Matrix{Int64}:
  7  13  19   6
 16  15  14  12
  1  12  11  14
 19   4   2  12
```

Рис. 6: Массив 4x

8. Повторим примеры с массивом: транспонирование, след матрицы.

```
# Транспонирование:
transpose(b)

4×4 transpose(::Matrix{Int64}) with eltype Int64:
  7  16   1  19
 13  15  12   4
 19  14  11   2
  6  12  14  12

# След матрицы (сумма диагональных элементов):
tr(b)

45
```

Рис. 7: Транспонирование, след матрицы

9. Повторим примеры с массивом: извлечение диагональных элементов как массив, ранг матрицы.

```
# Извлечение диагональных элементов как массив:
diag(b)

4-element Vector{Int64}:
  7
 15
 11
 12

# Ранг матрицы:
rank(b)

4
```

10. Повторим примеры с массивом: инверсия матрицы (определение обратной матрицы), определитель матрицы.

```
# Инверсия матрицы (определение обратной матрицы):
inv(b)

4×4 Matrix{Float64}:
  0.00578035   0.0289017  -0.0520231    0.0289017
 -0.179716     0.283237   -0.0189175   -0.171308
  0.166185    -0.169075    0.00433526   0.0809249
  0.0230557   -0.111994    0.0879532    0.0811876

# Определитель матрицы:
det(b)

15224.000000000002
```

Рис. 9: Инверсия матрицы, определитель матрицы

11. Повторим примеры с массивом: псевдобратная функция для прямоугольных матриц

```
# Псевдобратная функция для прямоугольных матриц:
pinv(a)

3×4 Matrix{Float64}:
  0.108197   -0.114931   -0.0327538   -0.0455152
 -0.117346    0.0514163   0.0703955    0.14614
  0.0156561   0.0804726   0.00337855  -0.0479738
```

Рис. 10: Псевдобратная функция для прямоугольных матриц

Для вычисления нормы используется `LinearAlgebra.norm(x)`.
Евклидова норма:

$$\|\vec{X}\|_2 = \sqrt{x_1^2 + x_2^2 + ... + x_n^2};$$

p-норма:

$$\|\vec{A}\|_p = \left(\sum_{i=1}^{n} |a_i|^p\right)^{1/p}.$$

12. Повторим пример с вычилением нормы, а именно создаем вектор, высчитываем евклидовую норму и p-норму.

```
# Создание вектора X:
X = [2, 4, -5]
```

```
3-element Vector{Int64}:
  2
  4
 -5
```

Рис. 11: Вектор

```
# Вычисление евклидовой нормы:
norm(X)
```

```
6.708203932499369
```

```
# Вычисление p-нормы:
p = 1
norm(X,p)
```

```
11.0
```

13. Повторим примеры с вычислением евклидово расстояния между двумя векторами.

```
# Расстояние между двумя векторами X и Y:
X = [2, 4, -5]
Y = [1,-1,3]
norm(X-Y)
```

```
9.486832980505138
```

```
# Проверка по базовому определению:
sqrt(sum((X-Y).^2))
```

```
9.486832980505138
```

Рис. 13: Расстояние

Угол между двумя векторами $\vec{X}$ и $\vec{Y}$ определяется как $\cos^{-1}\frac{\vec{X}^T\vec{Y}}{||\vec{X}||_2||\vec{Y}||_2}$.

14. Повторим примеры с вычислением угла между двумя веткорами.

```
# Угол между двумя векторами:
acos((transpose(X)*Y)/(norm(X)*norm(Y)))
```

```
2.4404307889469252
```

Рис. 14: Угол

Вычисление нормы векторов и матриц, повороты, вращения

15. Повторим пример с вычилением нормы для двумерной мтарицы, а именно создаем матрицу, высчитываем евклидовую норму и p-норму.

```
# Создание матрицы:
d = [5 -4 2 ; -1 2 3; -2 1 0]

3×3 Matrix{Int64}:
  5  -4  2
 -1   2  3
 -2   1  0

# Вычисление Евклидовой нормы:
opnorm(d)

7.147682841795258

# Вычисление p-нормы:
p=1
opnorm(d,p)

8.0
```

Рис. 15: Матрица и нормы

16. Выполним примеры с поворотом и переворачиваем по строкам и столбцам.

```
# Поворот на 180 градусов:
rot180(d)

3×3 Matrix{Int64}:
 0   1  -2
```

Матричное умножение, единичная матрица, скалярное произведение

17. Повторим примеры: создадим дву матрицы и заполним случайными значения и вычислим произвидение этих матриц.

```
# Матрица 2х3 со случайными целыми значениями от 1 до 10:
A = rand(1:10,(2,3))
# Матрица 3х4 со случайными целыми значениями от 1 до 10:
B = rand(1:10,(3,4))

print(A)
print()
print(B)
```

```
[4 6 8; 7 4 4][4 9 10 7; 5 5 8 4; 5 10 10 8]
```

```
# Произведение матриц A и B:
A*B
```

```
2×4 Matrix{Int64}:
 86  146  168  116
 68  123  142   97
```

Рис. 17: Матрицы и их произведение

18. Повторим пример создания единичной матрицы.

```
# Единичная матрица 3x3:
Matrix{Int}(I, 3, 3)
```

```
3×3 Matrix{Int64}:
 1  0  0
 0  1  0
 0  0  1
```

Рис. 18: Матрица

19. Повторим примеры: вычислим скалярное произведение двух векторов разными способами.

```
# Скалярное произведение векторов X и Y:
X = [2, 4, -5]
Y = [1,-1,3]
dot(X,Y)
```

```
-17
```

```
# тоже скалярное произведение:
X'Y
```

```
-17
```

В математике факторизация (или разложение) объекта — его декомпозиция (например, числа, полинома или матрицы) в произведение других объектов или факторов, которые, будучи перемноженными, дают исходный объект. Матрица может быть факторизована на произведение матриц специального вида для приложений, в которых эта форма удобна. К специальным видам матриц относят ортогональные, унитарные и треугольные матрицы.

Факторизация. Специальные матричные структуры

20. Повторим пример решение систем линейный алгебраических уравнений □□ = □: зададим все начальные условия и найдем решение.

```
# Задаём квадратную матрицу 3x3 со случайными значениями:
A = rand(3, 3)
```

```
3×3 Matrix{Float64}:
 0.323776  0.766627  0.346568
 0.82158   0.154756  0.212603
 0.089916  0.231533  0.572396
```

```
# Задаём единичный вектор:
x = fill(1.0, 3)
```

```
3-element Vector{Float64}:
 1.0
 1.0
 1.0
```

```
# Задаём вектор b:
b = A*x
```

```
3-element Vector{Float64}:
 1.4369700554357259
 1.1889391623864485
 0.8938450054376931
```

Рис. 20: Начальные условия

```
# Решение исходного уравнения получаем с помощью функции \
# (убеждаемся, что x - единичный вектор):
A\b
```

```
3-element Vector{Float64}:
 1.0
 1.0
 1.0
```

Рис. 21: Решение

21. Вычислим факторизацию: Julia позволяет вычислять LU-факторизацию и определяет составной тип факторизации для его хранения:

```
# LU-факторизация:
Alu = lu(A)
```

```
LU{Float64, Matrix{Float64}, Vector{Int64}}
L factor:
3×3 Matrix{Float64}:
 1.0       0.0       0.0
 0.394089  1.0       0.0
 0.109443  0.304115  1.0
U factor:
3×3 Matrix{Float64}:
 0.82158  0.154756  0.212603
 0.0      0.705639  0.262783
 0.0      0.0       0.469212
```

Рис. 22: LU-факторизация

Факторизация. Специальные матричные структуры

Различные части факторизации могут быть извлечены путём доступа к их специальным свойствам: Матрица перестановок: Alu.P Вектор перестановок: Alu.p Матрица L: Alu.L Матрица U: Alu.U

22. Повторим пример: исходная система уравнений □□ = □ может быть решена или с использованием исходной матрицы, или с использованием объекта факторизации:

```
# Решение СЛАУ через матрицу A:
A\b

3-element Vector{Float64}:
 1.0
 1.0
 1.0

# Решение СЛАУ через объект факторизации:
Alu\b

3-element Vector{Float64}:
 1.0
 1.0
 1.0
```

Рис. 23: Решение

23. Повторим пример и найдем детерминат матрицы.

```
# Детерминант матрицы A:
det(A)
```

```
-0.2720205154988499
```

```
# Детерминант матрицы A через объект факторизации:
det(Alu)
```

```
-0.2720205154988499
```

Рис. 24: Детерминат

24. Выполним пример: Julia позволяет вычислять QR-факторизацию и определяет составной тип факторизации для его хранения.

```
# QR-факторизация:
Aqr = qr(A)
```

```
LinearAlgebra.QRCompactWY{Float64, Matrix{Float64}, Matrix{Float64}}
Q factor:
3×3 LinearAlgebra.QRCompactWYQ{Float64, Matrix{Float64}, Matrix{Float64}}:
 -0.274365   0.843755  -0.461304
 -0.677435  -0.510059  -0.530021
 -0.6825     0.167084   0.711532
R factor:
3×3 Matrix{Float64}:
 -0.593653  -0.61946   -0.567117
  0.0        0.630194   0.169954
  0.0        0.0       -0.302095
```

Рис. 25: QR-факторизация

25. По аналогии с LU-факторизацией различные части QR-факторизации могут быть извлечены путём доступа к их специальным свойствам.

```
# Матрица Q:
Aqr.Q
```

```
3×3 LinearAlgebra.QRCompactWYQ{Float64, Matrix{Float64}, Matrix{Float64}}:
 -0.274365   0.843755  -0.461304
 -0.677435  -0.510059  -0.530021
 -0.6825     0.167084   0.711532
```

Рис. 26: Q

```
# Матрица R:
Aqr.R
```

```
3×3 Matrix{Float64}:
 -0.884374  -0.455329  -0.264604
  0.0       -0.792468  -0.521473
  0.0        0.0        0.462976
```

Рис. 27: R

```
# Проверка, что матрица Q - ортогональная:
Aqr.Q'*Aqr.Q
```

```
3×3 Matrix{Float64}:
  1.0          -2.77556e-17  -1.38778e-17
 -5.55112e-17   1.0           0.0
  0.0           0.0           1.0
```

Факторизация. Специальные матричные структуры

26. Выполним примеры собственной декомпозиции матрицы A, а именно симметризация матрицы A, спектральное разложение симметризованной матрицы, посик собственных значений и векторов.

```
# Симметризация матрицы A:
Asym = A + A'
```

```
3×3 Matrix{Float64}:
 1.74614   0.464061  0.211887
 0.464061  1.7004    0.570514
 0.211887  0.570514  0.929872
```

Рис. 29: Симметризация матрицы A

```
# Спектральное разложение симметризованной матрицы:
AsymEig = eigen(Asym)
```

```
Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}
values:
3-element Vector{Float64}:
 0.6257132275305365
 1.3504140615067672
 2.4002822908619654
vectors:
3×3 Matrix{Float64}:
  0.0328487   0.788192  -0.614552
 -0.479667   -0.52701   -0.701555
  0.876835   -0.317825  -0.360758
```

Рис. 30: Спектральное разложение симметризованной матрицы


```
# Собственные значения:
```

27. Далее рассмотрим примеры работы с матрицами большой размерности и специальной структуры: матрица 1000 x 1000.

```
# Матрица 1000 x 1000:
n = 1000
A = randn(n,n)
```

```
1000×1000 Matrix{Float64}:
  0.945564   -0.4136      0.288745   …  -0.0576756   -1.29635   -1.55455
 -0.733987   -0.0813476  -1.8088         0.00406874  -0.146778  -0.543025
  0.416595    0.215597    1.72953        2.10461      2.27       0.493712
 -1.17069    -1.28196     0.172427      -0.281696     0.113201   0.402691
 -0.534953   -0.454402   -0.0904224      0.737858     0.374288   1.03684
 -0.730627    0.79689     0.552524   …  -0.205875     1.10018   -0.537194
 -0.0978003  -0.556665    0.477047      -1.486       -0.628359  -0.915956
  0.562794   -1.07607    -0.526134       0.217834     1.1874    -0.744079
  0.585691    0.901834    0.0827245      1.53281      0.17808   -0.185429
 -0.546395    0.0415964  -0.736421       0.566128    -0.838803   0.172493
  0.413408    0.344019   -0.731707   …  -0.139578     1.23046   -0.406932
 -0.559737   -0.124365   -0.153766       2.37222      1.8644     0.234218
  2.64962    -0.339176    1.10631        1.39374      0.525698  -0.482248
  ⋮                                  ⋱
  1.15559    -1.46166    -1.49758       -0.585228    -1.92706   -0.161073
 -1.14017     0.672569    1.92851       -0.064009     0.101601   0.308163
  0.23013     1.43359     1.09088    …   1.47976      2.01193   -1.21324
 -0.456641   -0.34784    -1.69538        0.216645    -0.678083   0.237115
  0.557833    1.06935     0.690091      -0.104601    -0.676934   2.56679
 -0.0322185  -0.415773    0.477601      -1.52515      0.624246  -0.660277
 -0.810073    0.906215    0.130511       1.3033      -0.350364  -0.445778
 -1.39737    -0.0498869  -1.29807    …  -0.125225    -1.98868   -0.57905
 -0.489871   -1.03055    -0.382777      -0.408173     0.166898   2.47756
  2.38396     0.108788    0.98079       -1.67812     -1.18574    2.19278
  0.407764    0.613379    0.673965      -0.99283     -0.817734  -0.661687
 -1.19675    -0.097588    0.516592       0.694347     1.05551    0.146226
```

Рис. 32: Матрица 1000 x 1000

Факторизация. Специальные матричные структуры

28. Выполним для данной матрицы симметризацию, и проверку на симметрию.

```
# Симметризация матрицы:
Asym = A + A'
```

```
1000×1000 Matrix{Float64}:
  1.89113    -1.14759     0.705339  …   2.32628    -0.888584  -2.7513
 -1.14759    -0.162695   -1.59321       0.112857    0.466601  -0.640613
  0.705339   -1.59321     3.45905       3.0854      2.94397    1.0103
 -1.95993    -1.60121    -0.690125     -0.234089    1.40913    0.396142
 -1.10852     1.63509    -0.371592      1.63391     1.63206    1.65072
 -0.598721    0.935986    0.537715  …  -0.363744    1.30787    0.429677
  1.16067    -0.255707    0.216914     -2.40255    -2.46799   -1.46619
 -0.434411   -0.607108    0.816192      1.19282     1.27309   -0.270205
  0.412727    0.722902    0.424145      2.62941    -0.288722  -0.113701
 -0.434874   -0.659047   -1.59224       0.0749583   0.163534   0.777786
  1.96235     0.363711   -1.27095   …  -0.482602    0.352057  -0.0786477
 -0.814709   -0.445876   -1.83821       1.94458     2.5976     1.03824
  2.16479    -2.58858     2.87103       0.668372   -0.158715   0.53989
  ⋮                                 ⋱
  0.0138219  -1.43221     0.141513      1.13619    -2.71519   -0.368563
 -1.18133    -0.0518368   1.23705       0.923544   -0.336829   0.0998745
  0.789905    3.15444     1.84744   …   3.2034      4.25491   -1.54858
  0.0580671   0.419394   -2.65152      -0.614025   -0.492288  -0.157775
  1.01359    -0.496089    1.09417      -0.0253229   0.173054   2.17623
  2.10788    -0.433621    0.264345     -0.018247    0.350143   0.316145
  0.757072   -1.23472     0.593982      1.89946     1.68289   -1.29059
  0.917399   -0.740756   -2.87718   …  -1.5719     -1.71603   -1.20678
  0.10315    -0.956864   -0.613048      0.528737    0.177616   3.96403
  2.32628     0.112857    3.0854       -3.35623    -2.17857    2.88713
 -0.888584    0.466601    2.94397      -2.17857    -1.63547    0.393826
 -2.7513     -0.640613    1.0103        2.88713     0.393826   0.292451
```

Рис. 33: Симметризация

```
# Проверка, является ли матрица симметричной:
issymmetric(Asym)
```

```
true
```

Рис. 34: Проверка

29. Выполним пример добавления шума в симметричную матрицу (матрица уже не будет симметричной).

```
# Добавление шума:
Asym_noisy = copy(Asym)
Asym_noisy[1,2] += 5eps()
```

```
-1.147586931664726
```

```
# Проверка, является ли матрица симметричной:
issymmetric(Asym_noisy)
```

```
false
```

Рис. 35: Добавление шума и проверка

30. В Julia можно объявить структуру матрица явно, например, используя Diagonal, Triangular, Symmetric, Hermitian, Tridiagonal и SymTridiagonal.

```julia
# Явно указываем, что матрица является симметричной:
Asym_explicit = Symmetric(Asym_noisy)
```

```
1000×1000 Symmetric{Float64, Matrix{Float64}}:
  1.89113    -1.14759     0.705339  …   2.32628    -0.888584  -2.7513
 -1.14759    -0.162695   -1.59321       0.112857    0.466601  -0.640613
  0.705339   -1.59321     3.45905       3.0854      2.94397    1.0103
 -1.95993    -1.60121    -0.690125     -0.234089    1.40913    0.396142
 -1.10852     1.63509    -0.371592      1.63391     1.63206    1.65072
 -0.598721    0.935986    0.537715  …  -0.363744    1.30787    0.429677
  1.16067    -0.255707    0.216914     -2.40255    -2.46799   -1.46619
 -0.434411   -0.607108    0.816192      1.19282     1.27309   -0.270205
  0.412727    0.722902    0.424145      2.62941    -0.288722  -0.113701
 -0.434874   -0.659047   -1.59224       0.0749583   0.163534   0.777786
  1.96235     0.363711   -1.27095   …  -0.482602    0.352057  -0.0786477
 -0.814709   -0.445876   -1.83821       1.94458     2.5976     1.03824
  2.16479    -2.58858     2.87103       0.668372   -0.158715   0.53989
  ⋮                                 ⋱
  0.0138219  -1.43221     0.141513      1.13619    -2.71519   -0.368563
 -1.18133    -0.0518368   1.23705       0.923544   -0.336829   0.0998745
  0.789905    3.15444     1.84744   …   3.2034      4.25491   -1.54858
  0.0580671   0.419394   -2.65152      -0.614025   -0.492288  -0.157775
  1.01359    -0.496089    1.09417      -0.0253229   0.173054   2.17623
  2.10788    -0.433621    0.264345     -0.018247    0.350143   0.316145
  0.757072   -1.23472     0.593982      1.89946     1.68289   -1.29059
  0.917399   -0.740756   -2.87718   …  -1.5719     -1.71603   -1.20678
  0.10315    -0.956864   -0.613048      0.528737    0.177616   3.96403
  2.32628     0.112857    3.0854       -3.35623    -2.17857    2.88713
 -0.888584    0.466601    2.94397      -2.17857    -1.63547    0.393826
 -2.7513     -0.640613    1.0103        2.88713     0.393826   0.292451
```

Рис. 36: Указание на симметричность

31. Далее для оценки эффективности выполнения операций над матрицами большой размерности и специальной структуры воспользуемся пакетом BenchmarkTools, добавим его.

```
import Pkg
Pkg.add("BenchmarkTools")
using BenchmarkTools
```

```
   Resolving package versions...
   Installed BenchmarkTools ─ v1.5.0
    Updating `C:\Users\User\.julia\environments\v1.10\Project.toml`
  [6e4b80f9] + BenchmarkTools v1.5.0
    Updating `C:\Users\User\.julia\environments\v1.10\Manifest.toml`
  [6e4b80f9] ↑ BenchmarkTools v1.4.0 ⇒ v1.5.0
Precompiling project...
  ✓ BenchmarkTools
  ✓ MathOptInterface
  ✓ Optim
  ✓ DiffEqNoiseProcess
  ✓ StochasticDiffEq
  ✓ DifferentialEquations
  6 dependencies successfully precompiled in 144 seconds. 322 already preco
```

Рис. 37: Добавление пакета

32. Выполним оценку эффективности выполнения операции по нахождению собственных значений для различных матриц.

```
# Оценка эффективности выполнения операции по нахождению
# собственных значений симметризованной матрицы:
@btime eigvals(Asym)
```

```
  140.190 ms (11 allocations: 7.99 MiB)
1000-element Vector{Float64}:
 -89.90745697267798
 -89.28462914513722
 -87.7618794507008
 -86.82276458104195
 -86.17545419337942
 -85.50230407876174
 -85.06852580328531
 -84.70714389011076
 -84.5030609471845
 -84.18620811687069
 -83.58445180670451
 -83.05188255007515
 -82.88233698029482
   ⋮
  82.50731589150371
  83.0077687026647
  83.13600831985721
  83.28420721839369
  84.08896605208413
  84.6386358303351
  85.29488028420883
  85.65940396658235
  86.54358105900604
  87.12252883775672
  87.33499654466465
  87.5091798420538
```

Рис. 38: Оценка эффективности 1

```julia
# Оценка эффективности выполнения операции по нахождению
# собственных значений зашумлённой матрицы:
@btime eigvals(Asym_noisy)
```

```
  736.767 ms (14 allocations: 7.93 MiB)
1000-element Vector{Float64}:
 -89.90745697267911
 -89.28462914513783
 -87.76187945070113
 -86.8227645810423
 -86.17545419337884
 -85.50230407876121
 -85.06852580328572
 -84.70714389011033
 -84.50306094718434
 -84.18620811687012
 -83.58445180670492
 -83.05188255007515
 -82.8823369802946
   ⋮
  82.50731589150357
  83.00776870266446
  83.1360083198572
  83.28420721839342
  84.08896605208409
  84.63863583033428
  85.29488028420847
  85.65940396658272
  86.54358105900539
  87.12252883775666
  87.33499654466515
  87.50917984205434
```

Рис. 39: Оценка эффективности 2

```
# Оценка эффективности выполнения операции по нахождению
# собственных значений зашумлённой матрицы,
# для которой явно указано, что она симметричная:
@btime eigvals(Asym_explicit)
```

```
  153.165 ms (11 allocations: 7.99 MiB)
1000-element Vector{Float64}:
 -89.90745697267839
 -89.28462914513725
 -87.76187945070029
 -86.82276458104262
 -86.1754541933794
 -85.50230407876178
 -85.06852580328525
 -84.70714389011067
 -84.50306094718444
 -84.18620811687055
 -83.5844518067044
 -83.05188255007535
 -82.88233698029536
   ⋮
  82.5073158915038
  83.00776870266455
  83.13600831985687
  83.2842072183936
  84.08896605208412
  84.6386358303352
  85.29488028420896
  85.65940396658252
  86.54358105900596
  87.12252883775663
  87.33499654466463
  87.50917984205351
```

Рис. 40: Оценка эффективности 3

33. Далее рассмотрим примеры работы с разряженными матрицами большой размерности. Использование типов Tridiagonal и SymTridiagonal для хранения трёхдиагональных матриц позволяет работать с потенциально очень большими трёхдиагональными матрицами. Выполним оценку эффективности.

```julia
# Трёхдиагональная матрица 1000000 x 1000000:
n = 1000000
A = SymTridiagonal(randn(n), randn(n-1))
```

```
1000000×1000000 SymTridiagonal{Float64, Vector{Float64}}:
 0.540287   0.7954     ⋅          ⋅        …   ⋅          ⋅         ⋅
 0.7954     1.1026   -1.52548    ⋅              ⋅          ⋅         ⋅
  ⋅        -1.52548   0.197984  1.57573         ⋅          ⋅         ⋅
  ⋅          ⋅        1.57573   0.214912        ⋅          ⋅         ⋅
  ⋅          ⋅         ⋅        1.4176          ⋅          ⋅         ⋅
  ⋅          ⋅         ⋅         ⋅         …    ⋅          ⋅         ⋅
  ⋅          ⋅         ⋅         ⋅              ⋅          ⋅         ⋅
  ⋅          ⋅         ⋅         ⋅              ⋅          ⋅         ⋅
  ⋅          ⋅         ⋅         ⋅              ⋅          ⋅         ⋅
  ⋅          ⋅         ⋅         ⋅              ⋅          ⋅         ⋅
  ⋅          ⋅         ⋅         ⋅         …    ⋅          ⋅         ⋅
  ⋅          ⋅         ⋅         ⋅              ⋅          ⋅         ⋅
  ⋅          ⋅         ⋅         ⋅              ⋅          ⋅         ⋅
 ⋮                                         ⋱
  ⋅          ⋅         ⋅         ⋅              ⋅          ⋅         ⋅
  ⋅          ⋅         ⋅         ⋅              ⋅          ⋅         ⋅
  ⋅          ⋅         ⋅         ⋅         …    ⋅          ⋅         ⋅
  ⋅          ⋅         ⋅         ⋅              ⋅          ⋅         ⋅
  ⋅          ⋅         ⋅         ⋅              ⋅          ⋅         ⋅
  ⋅          ⋅         ⋅         ⋅              ⋅          ⋅         ⋅
  ⋅          ⋅         ⋅         ⋅              ⋅          ⋅         ⋅
  ⋅          ⋅         ⋅         ⋅         …    ⋅          ⋅         ⋅
  ⋅          ⋅         ⋅         ⋅            -0.169001    ⋅         ⋅
  ⋅          ⋅         ⋅         ⋅             0.546604   0.357314   ⋅
  ⋅          ⋅         ⋅         ⋅             0.357314  -0.4732    0.106754
  ⋅          ⋅         ⋅         ⋅              ⋅         0.106754  0.213239
```

Рис. 41: Матрица 1000000x1000000

```
# Оценка эффективности выполнения операции по нахождению
# собственных значений:
@btime eigmax(A)
```

```
  524.792 ms (17 allocations: 183.11 MiB)
6.275049715720039
```

Рис. 42: Оценка эффективности

34. При попытке задать подобную матрицу обычным способом и посчитать её собственные значения, мы получим ошибку переполнения памяти.

```
B = Matrix(A)
```

```
OutOfMemoryError()

Stacktrace:
 [1] Array
   @ .\boot.jl:479 [inlined]
 [2] Matrix{Float64}(M::SymTridiagonal{Float64, Vector{Float64}})
   @ LinearAlgebra C:\Users\User\AppData\Local\Programs\Julia-1.10.0\share\julia
diag.jl:127
 [3] (Matrix)(M::SymTridiagonal{Float64, Vector{Float64}})
   @ LinearAlgebra C:\Users\User\AppData\Local\Programs\Julia-1.10.0\share\julia
diag.jl:138
 [4] top-level scope
   @ In[108]:1
```

Рис. 43: Ошибка

Общая линейная алгебра

Обычный способ добавить поддержку числовой линейной алгебры - это обернуть подпрограммы BLAS и LAPACK. Собственно, для матриц с элементами Float32,Float64, Complex {Float32} или Complex {Float64} разработчики Julia использовали такое же решение. Однако Julia также поддерживает общую линейную алгебру, что позволяет, например, работать с матрицами и векторами рациональных чисел.

35. В следующем примере показано, как можно решить систему линейных уравнений с рациональными элементами без преобразования в типы элементов с плавающей запятой (для избежания проблемы с переполнением используем BigInt).

```
# Матрица с рациональными элементами:
Arational = Matrix{Rational{BigInt}}(rand(1:10, 3, 3))/10

3×3 Matrix{Rational{BigInt}}:
 1//10  1//2  2//5
 3//10  4//5  7//10
  1     3//5  4//5
```

```
# Единичный вектор:
x = fill(1, 3)
# Задаём вектор b:
b = Arational*x

3-element Vector{Rational{BigInt}}:
   1
  9//5
 12//5
```

Рис. 44: Матрица с рациональными элементами

Общая линейная алгебра

```julia
# Решение исходного уравнения получаем с помощью функции \
# (убеждаемся, что x - единичный вектор):
Arational\b
```

```
3-element Vector{Rational{BigInt}}:
 1
 1
 1
```

```julia
# LU-разложение:
lu(Arational)
```

```
LU{Rational{BigInt}, Matrix{Rational{BigInt}}, Vector{Int64}}
L factor:
3×3 Matrix{Rational{BigInt}}:
  1       0     0
 3//10    1     0
 1//10  22//31  1
U factor:
3×3 Matrix{Rational{BigInt}}:
 1   3//5    4//5
 0  31//50  23//50
 0    0     -1//155
```

Рис. 45: Решение и LU-разложение

Задания для самостоятельного выполнения

Произведение векторов

36. Выполним 1.1 задание: задайте вектор v. Умножьте вектор v скалярно сам на себя и сохраните результат в dot_v.

```
v = rand(1:100, 3); display(v)
dot_v = v'v
```

```
3-element Vector{Int64}:
 16
 69
 25
5642
```

Рис. 46: 1.1 задание

37. Выполним 1.2 задание: умножьте v матрично на себя (внешнее произведение), присвоив результат переменной outer_v.

```
outer_v = v*v'
```

```
3×3 Matrix{Int64}:
  256  1104   400
 1104  4761  1725
  400  1725   625
```


Рис. 47: 1.2 задание

38. Выполним 2 задание: Решить СЛАУ с двумя/тремя неизвестными.

```julia
function LinearDep(mtrx::Matrix, vec::Vector)
    # returns isSolvable::Bool, ind::Vector{Int64} -- вектор
    A = hcat(mtrx, vec)
    Ac = copy(mtrx); bc = copy(vec)
    s1 = size(A)[1]; s2 = size(A)[2]-1
    t = [false for i in 1:size(A)[1]]
    poss_j = collect(2:s2)
    for i in 1:s2
        for j in i+1:s1
            mbool = true
            temp = A[j, :]./A[i, :]
            if length(unique(temp[1:s2])) == 1
                if temp[s2+1] == temp[1]
                    t[j] = true
                else
                    return false, []
                end
            end
            tii = i
            if Ac[i, i] == 0
                tii = sortperm(abs.(Ac[i, :]))[s2]
                if Ac[i, tii] == 0
                    mbool = false
                end
            end
            if mbool
                c = -Ac[j, tii] / Ac[i, tii]
                if isequal(Ac[j, :].+(c*Ac[i, :]), zeros(Float64, s2))
                    if bc[j] + c*bc[i] != 0
                        return false, []
                    else
                        t[j] = true
                        Ac[j, :] = Ac[j, :].+(c*Ac[i, :])
                        bc[j] += c*bc[i]
                    end
                else
                    Ac[j, :].+= (c*Ac[i, :])
                    bc[j] += c*bc[i]
                end
            end
        end
    end
```

Рис. 48: Функция для решения

Системы линейных уравнений

```julia
    for i in 1:s1
        if isequal(Ac[i, :], zeros(Float64, s2))
            t[i] = true
        end
    end
    answ = deleteat!(collect(1:s1), t)
    if length(answ) >= s2
        return true, answ
    else
        return false, [pi]
    end
end

function SLAU_solver(A::Matrix, b::Vector)
    if ndims(A) != 2 || size(A)[1] != length(b)
        println("Не совпадают размерности!")
        return
    end
    s1 = size(A)[1]; s2 = size(A)[2]
    if s1 == s2 && det(A) != 0
        return A\b
    elseif s1 < s2
        println("Уравнений меньше, чем переменных")
        return
    else # s1 > s2 || (s1 == s2 && det(A) == 0)
        isSolvable, indNonLinear = LinearDep(A, b)
        if !isSolvable && isequal(indNonLinear, [])
            println("Нет решений")
            return
        elseif !isSolvable && isequal(indNonLinear, [pi])
            println("Бесконечное количество решений")
            return
        else
            length(indNonLinear) > s2 ? indNonLinear = indNonLinear[1:s2] :
            return A[indNonLinear, :]\b[indNonLinear]
        end
    end
end
A = Float64[1 2 3; 1/3 2 1; 2 3 6; 3 4 5]
b = Float64[1, 1, 4, 5]
SLAU_solver(A, b)
```

Рис. 49: Функция для решения

а) Решение существует (система линейно независима)

$$\begin{cases} x + y = 2, \\ x - y = 3. \end{cases}$$

```
A = Float64[1 1; 1 -1]
b = Float64[2, 3]
SLAU_solver(A, b)
```

```
2-element Vector{Float64}:
  2.5
 -0.5
```

Рис. 50: а)

b) Бесконечное количество решений (вся система линейно зависима и коэффициенты и ветокры ответов)

$$\begin{cases} x + y = 2, \\ 2x + 2y = 4. \end{cases}$$

```
A = Float64[1 1; 2 2]
b = Float64[2, 4]
SLAU_solver(A, b)
```

```
Бесконечное количество решений
```

Рис. 51: b)

c) Нет решений (матрица коэффициентов линейно зависима, при этом векторы нет)

$$\begin{cases} x + y = 2, \\ 2x + 2y = 5. \end{cases}$$

```
A = Float64[1 1; 2 2]
b = Float64[2, 5]
SLAU_solver(A, b)
```

```
Нет решений
```

Рис. 52: c)

d) Бесконечное количество решений (вся система линейно зависима)

$$\begin{cases} x + y = 1, \\ 2x + 2y = 2, \\ 3x + 3y = 3. \end{cases}$$

```
A = Float64[1 1; 2 2; 3 3]
b = Float64[1, 2, 3]
SLAU_solver(A, b)
```

```
Бесконечное количество решений
```

Рис. 53: d)

e) Нет решений

$$\begin{cases} x + y = 2, \\ 2x + y = 1, \\ x - y = 3. \end{cases}$$

```
A = Float64[1 1; 2 1; 1 -1]
b = Float64[2, 1, 3]
SLAU_solver(A, b)
```

```
Нет решений
```

Рис. 54: e)

f) Решение существует

$$\begin{cases} x + y = 2, \\ 2x + y = 1, \\ 3x + 2y = 3. \end{cases}$$

```
A = Float64[1 1; 2 1; 3 2]
b = Float64[2, 1, 3]
SLAU_solver(A, b)
```

```
2-element Vector{Float64}:
 -1.0
  3.0
```

Рис. 55: f)

- Решить СЛАУ с тремя неизвестными:

а)

$$\begin{cases} x + y + z = 2, \\ x - y - 2z = 3. \end{cases}$$

```
A = Float64[1 1 1; 1 -1 -2]
b = Float64[2, 3]
SLAU_solver(A, b)

Уравнений меньше, чем переменных
```

Рис. 56: а)

b)

$$\begin{cases} x + y + z = 2, \\ 2x + 2y - 3z = 4, \\ 3x + y + z = 1. \end{cases}$$

```
A = Float64[1 1 1; 2 2 -3; 3 1 1]
b = Float64[2, 4, 1]
SLAU_solver(A, b)
```

```
3-element Vector{Float64}:
 -0.5
  2.5
  0.0
```

Рис. 57: b)

Системы линейных уравнений

c)

$$\begin{cases} x + y + z = 1, \\ x + y + 2z = 0, \\ 2x + 2y + 3z = 1. \end{cases}$$

```
A = Float64[1 1 1; 1 1 2; 2 2 3]
b = Float64[1, 0, 1]
SLAU_solver(A, b)
```

```
Бесконечное количество решений
```

Рис. 58: c)

Системы линейных уравнений

d)

$$\begin{cases} x + y + z = 1, \\ x + y + 2z = 0, \\ 2x + 2y + 3z = 0. \end{cases}$$

```
A = Float64[1 1 1; 1 1 2; 2 2 3]
b = Float64[1, 0, 0]
SLAU_solver(A, b)
```

```
Нет решений
```

Рис. 59: d)

39. Приведите приведённые ниже матрицы к диагональному виду:

```julia
function to_Diagonal(mtrx)
    s = size(mtrx)[1]
    ordDown = vcat([[if i == s; [i, j-1] else [i, j] end for j in i+1:s] for i in 1:s]...)
    ordUP = vcat([[if i == s; [j-1, i] else [j, i] end for j in i+1:s] for i in s:-1:1]...)
    answ = [[] for _ in 1:s]
    for i in ordDown
        if mtrx[i[2], i[1]] != 0
            k = mtrx[i[2], i[1]] / mtrx[i[1], i[1]]
            answ[i[2]] = [mtrx[i[2], j] - mtrx[i[1], j] * k for j in 1:s]
        end
    end
    for i in ordUP
        if mtrx[i[2], i[1]] != 0
            k = mtrx[i[2], i[1]] / mtrx[i[1], i[1]]
            answ[i[2]] = [mtrx[i[2], j] - mtrx[i[1], j] * k for j in 1:s]
        end
    end
    return copy(hcat(answ...)')
end
```

```
to_Diagonal (generic function with 1 method)
```

Рис. 60: Функция для решения

Операции с матрицами

```
# a)
A = Float64[1 -2; -2 1]
to_Diagonal(A)
```

```
2×2 Matrix{Float64}:
 -3.0   0.0
  0.0  -3.0
```

```
# b)
A = Float64[1 -2; -2 3]
to_Diagonal(A)
```

```
2×2 Matrix{Float64}:
 -0.333333   0.0
  0.0       -1.0
```

```
# c)
A = Float64[1 -2 0; -2 1 2; 0 2 0]
to_Diagonal(A)
```

```
3×3 Matrix{Float64}:
  -3.0    0.0    4.0
 NaN    -Inf   NaN
   4.0    0.0   -4.0
```

Рис. 61: a,b,c)

Операции с матрицами

- Вычислите:

```
function mtrx_Function(A::Matrix, op)
    X = eigvecs(A)
    lamb = diagm(eigvals(A))
    lambfunc = [op(l) for l in lamb]
    answ = X^(-1)*lambfunc*X
    return answ
end

mtrx_Function (generic function with 1 method)
```

Рис. 62: Функция для решения

Операции с матрицами

```
# a)
A = [1 -2; -2 1]; display(A^10)
mtrx_Function(A, x -> x^10)
```

```
2×2 Matrix{Int64}:
  29525  -29524
 -29524   29525
2×2 Matrix{Float64}:
  29525.0  -29524.0
 -29524.0   29525.0
```

```
# b)
A = [5 -2; -2 5]
mtrx_Function(A, x -> sqrt(x))
```

```
2×2 Matrix{Float64}:
  2.1889   -0.45685
 -0.45685   2.1889
```

```
# c)
A = [1 -2; -2 1]
mtrx_Function(A, x -> cbrt(x))
```

```
2×2 Matrix{Float64}:
  0.221125  -1.22112
 -1.22112    0.221125
```

```
# d)
A = ComplexF64[1 2; 3 4]
mtrx_Function(A, x -> sqrt(x))
```

```
2×2 Matrix{ComplexF64}:
 0.553689+0.464394im  -0.889962+0.234276im
 -1.09755+0.288922im    1.76413+0.145754im
```

Рис. 63: a,b,c,d)

Операции с матрицами

- Найдите собственные значения матрицы □, если:

```
A = [140 97 74 168 131; 97 106 89 131 36; 74 89 152 144 71; 168 131 144 52 142; 131 36 71 142 36]
@btime eigvals(A)

  2.167 μs (10 allocations: 2.59 KiB)

5-element Vector{Float64}:
 -129.84037845927043
  -56.008181312078634
   42.75068638743729
   87.15844501190598
  541.9394283720058
```

Рис. 64: A

- Создайте диагональную матрицу из собственных значений матрицы □. Создайте нижнедиагональную матрицу из матрица □. Оцените эффективность выполняемых операций.

```
@btime diagm(eigvals(A))

  2.200 μs (11 allocations: 2.84 KiB)

5×5 Matrix{Float64}:
 -129.84     0.0       0.0      0.0      0.0
    0.0    -56.0082    0.0      0.0      0.0
    0.0      0.0      42.7507   0.0      0.0
    0.0      0.0       0.0     87.1584   0.0
    0.0      0.0       0.0      0.0    541.939

@btime blA = Bidiagonal(A, :L)

  313.248 ns (3 allocations: 224 bytes)

5×5 Bidiagonal{Int64, Vector{Int64}}:
 140   ⋅    ⋅    ⋅   ⋅
  97  106   ⋅    ⋅   ⋅
   ⋅   89  152   ⋅   ⋅
   ⋅    ⋅  144   52  ⋅
   ⋅    ⋅    ⋅  142  36
```

Рис. 65: A из собственных значений

Линейные модели экономики

40. Выполним задание: 1. Матрица □ называется продуктивной, если решение □ системы при любой неотрицательной правой части □ имеет только неотрицательные элементы □□. Используя это определение, проверьте, являются ли матрицы продуктивными.

```
function economicModel(M, y)
    x = (Diagonal(fill(1, 2)) - M)^(-1) * y
    return x
end
```

```
economicModel (generic function with 1 method)
```

Рис. 66: Функция для решения

Линейные модели экономики

```julia
# a)
A = [1 2; 3 4]
Y = [2; 1]
X = economicModel(A,Y); display(X)
if mapreduce(z -> if z < 0 1 else 0 end, +, X) > 0
    println("Непродуктивный")
else
    println("Продуктивный")
end
```

```
2-element Vector{Float64}:
  0.6666666666666667
 -1.0
Непродуктивный
```

```julia
# b)
A = [1 2; 3 4]*0.5
Y = [2; 1]
X = economicModel(A,Y); display(X)
if mapreduce(z -> if z < 0 1 else 0 end, +, X) > 0
    println("Непродуктивный")
else
    println("Продуктивный")
end
```

```
2-element Vector{Float64}:
  0.5
 -1.75
Непродуктивный
```

Рис. 67: a,b)

```julia
# c)
A = [1 2; 3 4]*0.1
Y = [2; 1]
X = economicModel(A,Y); display(X)
if mapreduce(z -> if z < 0 1 else 0 end, +, X) > 0
    println("Непродуктивный")
else
    println("Продуктивный")
end
```

```
2-element Vector{Float64}:
 2.9166666666666665
 3.125
Продуктивный
```

Рис. 68: c)

41. Выполним задание: 2. Критерий продуктивности: матрица □ является продуктивной тогда и только тогда, когда все элементы матрица (□ − □)−1 являются неотрицательными числами. Используя этот критерий, проверьте, являются ли матрицы продуктивными.

```
function OnesModel(M)
    x = (Diagonal(fill(1, size(M,1))) - M)^(-1)
    return x
end
```

```
OnesModel (generic function with 1 method)
```

Рис. 69: Функция для решения

```
# a)
A = [1 2; 3 1]
X = OnesModel(A); display(X)
if mapreduce(z -> if z < 0 1 else 0 end, +, X) > 0
    println("Непродуктивный")
else
    println("Продуктивный")
end
```

```
2×2 Matrix{Float64}:
 -0.0  -0.333333
 -0.5   0.0
Непродуктивный
```

```
# b)
A = [1 2; 3 1]*0.5
X = OnesModel(A); display(X)
if mapreduce(z -> if z < 0 1 else 0 end, +, X) > 0
    println("Непродуктивный")
else
    println("Продуктивный")
end
```

```
2×2 Matrix{Float64}:
 -0.4  -0.8
 -1.2  -0.4
Непродуктивный
```

Рис. 70: a,b)

Линейные модели экономики

```
# c)
A = [1 2; 3 1]*0.1
X = OnesModel(A); display(X)
if mapreduce(z -> if z < 0 1 else 0 end, +, X) > 0
    println("Непродуктивный")
else
    println("Продуктивный")
end
```

```
2×2 Matrix{Float64}:
 1.2  0.266667
 0.4  1.2
Продуктивный
```

Рис. 71: c)

42. Выполним задание: Спектральный критерий продуктивности: матрица □ является продуктивной тогда и только тогда, когда все её собственные значения по модулю меньше 1. Используя этот критерий, проверьте, являются ли матрицы продуктивными.

```
# a)
A = [1 2; 3 1]
X = eigenvalues(A); display(X)
if mapreduce(z -> if z < 0 1 else 0 end, +, X) > 0
    println("Непродуктивный")
else
    println("Продуктивный")
end
```

```
2-element Vector{Float64}:
 -1.4494897427831779
  3.4494897427831783
Непродуктивный
```

```
# b)
A = [1 2; 3 1]*0.5
X = eigenvalues(A); display(X)
if mapreduce(z -> if z < 0 1 else 0 end, +, X) > 0
    println("Непродуктивный")
else
    println("Продуктивный")
end
```

```
2-element Vector{Float64}:
 -0.7247448713915892
  1.724744871391589
Непродуктивный
```

Рис. 72: a,b)

Линейные модели экономики

```julia
# c)
A = [1 2; 3 1]*0.1
X = eigenvalues(A); display(X)
if mapreduce(z -> if z < 0 1 else 0 end, +, X) > 0
    println("Непродуктивный")
else
    println("Продуктивный")
end
```

```
2-element Vector{Float64}:
 -0.14494897427831785
  0.34494897427831783
Непродуктивный
```

```julia
# d)
A = [0.1 0.2 0.3; 0.0 0.1 0.2; 0.0 0.1 0.3]
X = eigenvalues(A); display(X)
if mapreduce(z -> if z < 0 1 else 0 end, +, X) > 0
    println("Непродуктивный")
else
    println("Продуктивный")
end
```

```
3-element Vector{Float64}:
 0.02679491924311228
 0.1
 0.37320508075688774
Продуктивный
```

Рис. 73: c,d)

Выводы по проделанной работе

В результате выполнения работы мы изучили возможностей специализированных пакетов Julia для выполнения и оценки эффективности операций над объектами линейной алгебры. Были записаны скринкасты выполнения , создания отчета, презентации и защиты лабораторной работы.

Список литературы

- Julia: https://ru.wikipedia.org/wiki/Julia
- https://julialang.org/packages/
- https://juliahub.com/ui/Home
- https://juliaobserver.com/
- https://github.com/svaksha/Julia.jl