

Лабораторная работа №4

Линейная алгебра

Чемоданова Ангелина Александровна

15 сентября 2025

Российский университет дружбы народов имени Патриса Лумумбы, Москва, Россия

- Чемоданова Ангелина Александровна
- Студентка НФИбд-02-22
- Российский университет дружбы народов имени Патриса Лумумбы
- 1132226443@pfur.ru
- <https://github.com/aachemodanova>



Цель работы

Основной целью работы является изучение возможностей специализированных пакетов Julia для выполнения и оценки эффективности операций над объектами линейной алгебры.

Задание

1. Используя Jupyter Lab, повторите примеры.
2. Выполните задания для самостоятельной работы.

Поэлементные операции над многомерными массивами

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

Julia

```
4x3 Matrix{Int64}:  
14 13 17  
18 18 7  
8 19 13  
15 16 3
```

```
# Поэлементная сумма:  
sum(a)
```

Julia

153

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

Julia

```
1x3 Matrix{Int64}:  
47 66 40
```

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

Julia

```
4x1 Matrix{Int64}:  
44  
35  
40  
34
```

Рис. 1: Поэлементные операции сложения и произведения элементов матрицы

Поэлементные операции над многомерными массивами

```
# Поэлементное произведение:  
prod(a)
```

5546388556800

Julia

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

1x3 Matrix{Int64}:
16800 71136 4641

Julia

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

4x1 Matrix{Int64}:
3094
1260
1976
720

Julia

Рис. 2: Поэлементные операции сложения и произведения элементов матрицы

Поэлементные операции над многомерными массивами

The screenshot shows a Julia development environment with several code snippets and their results:

- Code:** # Подключение пакета Statistics:
import Pkg
Pkg.add("Statistics")
using Statistics
- Output:** Updating registry at `C:\Users\angelina\.julia\registries\General.toml`
Resolving package versions...
Updating `C:\Users\angelina\.julia\environments\v1.11\Project.toml`
[10745b16] + Statistics v1.11.1
No Changes to `C:\Users\angelina\.julia\environments\v1.11\Manifest.toml`
- Code:** # Вычисление среднего значения массива:
mean(a)
- Output:** 12.75
- Code:** # Среднее по столбцам:
mean(a,dims=1)
- Output:** 1x3 Matrix{Float64}:
11.75 16.5 10.0
- Code:** # Среднее по строкам:
mean(a,dims=2)
- Output:** 4x1 Matrix{Float64}:
14.66666666666666
11.66666666666666
13.33333333333334
11.33333333333334

Рис. 3: Использование возможностей пакета Statistics для работы со средними значениями

Транспонирование, след, ранг, определитель и инверсия матрицы

```
import Pkg
Pkg.add("LinearAlgebra")
using LinearAlgebra
Julia

Resolving package versions...
Updating `C:\Users\angelina\.julia\environments\v1.11\Project.toml`
[37e2e46d] + LinearAlgebra v1.11.0
No Changes to `C:\Users\angelina\.julia\environments\v1.11\Manifest.toml`


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

4x4 Matrix{Int64}:
 1  4  18   6
 9  2  18   7
 7  6  18   9
 5  20  20  17


# Транспонирование:
transpose(b)
Julia

4x4 transpose(::Matrix{Int64}) with eltype Int64:
 1   9   7   5
 4   2   6  20
 18  18  18  20
 6   7   9  17
```

Рис. 4: Использование библиотеки LinearAlgebra для выполнения определённых операций

Транспонирование, след, ранг, определитель и инверсия матрицы

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

38

Julia

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

```
4-element Vector{Int64}:  
1  
2  
18  
17
```

Julia

Рис. 5: Использование библиотеки LinearAlgebra для выполнения определённых операций

Транспонирование, след, ранг, определитель и инверсия матрицы

```
# Ранг матрицы:  
rank(b)
```

4

Julia

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

Julia

```
4x4 Matrix{Float64}:  
0.0612245 1.19388 -1.66327 0.367347  
0.387755 2.31122 -3.61735 0.826531  
0.204082 0.729592 -1.12755 0.22449  
-0.714286 -3.92857 6.07143 -1.28571
```

```
# Определитель матрицы:  
det(b)
```

-392.0000000000284

Julia

```
# ПсевдоДоматрица для прямоугольных матриц:  
pinv(a)
```

Julia

```
3x4 Matrix{Float64}:  
0.0544327 -0.0248448 -0.0744492 0.0721322  
-0.0586165 0.0446518 0.0534294 -0.00355481  
0.0576668 -0.0199853 0.0246542 -0.0536477
```

9/41

Рис. 6: Использование библиотеки LinearAlgebra для выполнения определённых операций

Вычисление нормы векторов и матриц, повороты, вращения

```
# Создание вектора X:  
X = [2, 4, -5]  
  
Julia  
  
3-element Vector{Int64}:  
2  
4  
-5  
  
# Вычисление евклидовой нормы:  
norm(X)  
  
Julia  
6.708203932499369  
  
# Вычисление p-нормы:  
p = 1  
norm(X,p)  
  
Julia  
11.0  
  
# Расстояние между двумя векторами X и Y:  
X = [2, 4, -5];  
Y = [1, -1, 3];  
norm(X-Y)  
  
Julia  
9.486832980505138
```

Рис. 7: Использование LinearAlgebra.norm(x)

Вычисление нормы векторов и матриц, повороты, вращения

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

9.486832980505138

Julia

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

2.4404307889469252

Julia

Рис. 8: Использование LinearAlgebra.norm(x)

Вычисление нормы векторов и матриц, повороты, вращения

```
# Создание матрицы:  
d = [5 -4 2; -1 2 3; -2 1 0] Julia  
  
3x3 Matrix{Int64}:  
5 -4 2  
-1 2 3  
-2 1 0  
  
# Вычисление Евклидовой нормы:  
norm(d) Julia  
  
7.147682841795258  
  
# Вычисление p-нормы:  
p=1  
norm(d,p) Julia  
  
8.0  
  
# Поворот на 180 градусов:  
rot180(d) Julia  
  
3x3 Matrix{Int64}:  
0 1 -2  
3 2 -1  
2 -4 5
```

Рис. 9: Вычисление нормы для двумерной матрицы

Вычисление нормы векторов и матриц, повороты, вращения

```
# Переворачивание строк:  
reverse(d,dims=1)
```

Julia

```
3x3 Matrix{Int64}:  
-2 1 0  
-1 2 3  
5 -4 2
```

```
# Переворачивание столбцов  
reverse(d,dims=2)
```

Julia

```
3x3 Matrix{Int64}:  
2 -4 5  
3 2 -1  
0 1 -2
```

Рис. 10: Вычисление нормы для двумерной матрицы

Матричное умножение, единичная матрица, скалярное произведение

```
# Матрица 2x3 со случайными целыми значениями от 1 до 10:  
A = rand(1:10,(2,3))  
  
Julia  
  
2x3 Matrix{Int64}:  
1 1 1  
1 1 7  
  
  
# Матрица 3x4 со случайными целыми значениями от 1 до 10:  
B = rand(1:10,(3,4))  
  
Julia  
  
3x4 Matrix{Int64}:  
2 10 7 8  
8 5 7 1  
7 7 5 10  
  
  
# Произведение матриц A и B:  
A*B  
  
Julia  
  
2x4 Matrix{Int64}:  
17 22 19 19  
59 64 49 79  
  
  
# Единичная матрица 3x3:  
Matrix{Int}(I, 3, 3)  
  
Julia  
  
3x3 Matrix{Int64}:  
1 0 0  
0 1 0  
0 0 1
```

Рис. 11: Примеры матричного умножения, единичной матрицы и скалярного произведения

Матричное умножение, единичная матрица, скалярное произведение

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

Julia

-17

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

Julia

-17

Рис. 12: Примеры матричного умножения, единичной матрицы и скалярного произведения

Факторизация. Специальные матричные структуры

```
# Задаём квадратную матрицу 3х3 со случайными значениями:  
A = rand(3, 3)                                Julia  
  
3x3 Matrix{Float64}:  
0.737313  0.14065   0.915475  
0.174333  0.91222   0.17704  
0.49423   0.719616  0.705277  
  
# Задаём единичный вектор:  
x = fill(1.0, 3)                                Julia  
  
3-element Vector{Float64}:  
1.0  
1.0  
1.0  
  
# Задаём вектор b:  
b = A*x                                Julia  
  
3-element Vector{Float64}:  
1.7934376331222044  
1.2635934328931144  
1.9191236166471417  
  
# Решение исходного уравнения получаем с помощью функции \  
# (убеждаемся, что x - единичный вектор):  
A\b                                Julia  
  
3-element Vector{Float64}:  
0.9999999999999999  
1.0  
1.0000000000000009
```

Рис. 13: Решение систем линейных алгебраических уравнений $Ax = b$

Факторизация. Специальные матричные структуры

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:  
3x3 Matrix{Float64}:  
1.0 0.0 0.0  
0.236444 1.0 0.0  
0.678313 0.711447 1.0  
U factor:  
3x3 Matrix{Float64}:  
0.737313 0.14865 0.915475  
0.0 0.878964 -0.8394186  
0.0 0.0 0.119667  
  
# Матрица перестановок:  
Alu.P  
  
3x3 Matrix{Float64}:  
1.0 0.0 0.0  
0.0 1.0 0.0  
0.0 0.0 1.0  
  
# Вектор перестановок:  
Alu.p  
  
3-element Vector{Int64}:  
1  
2  
3
```

Рис. 14: Пример вычисления LU-факторизации и определение составного типа факторизации для его хранения

Факторизация. Специальные матричные структуры

```
# Матрица L:  
Alu.L
```

Julia

```
3x3 Matrix{Float64}:  
1.0      0.0      0.0  
0.236444 1.0      0.0  
0.670313 0.711447 1.0
```

```
# Матрица U:  
Alu.U
```

Julia

```
3x3 Matrix{Float64}:  
0.737313 0.14065   0.915475  
0.0       0.878964 -0.0394186  
0.0       0.0       0.119667
```

Рис. 15: Пример вычисления LU-факторизации и определение составного типа факторизации для его хранения

Факторизация. Специальные матричные структуры

Исходная система уравнений $Ax = b$ может быть решена или с использованием исходной матрицы, или с использованием объекта факторизации:

```
# Решение СЛАУ через матрицу A:  
A\b  
  
3-element Vector{Float64}:  
0.99999999999999  
1.0  
1.0000000000000009  
  
# Решение СЛАУ через объект факторизации:  
Alu\b  
  
3-element Vector{Float64}:  
0.99999999999999  
1.0  
1.0000000000000009  
  
# Детерминант матрицы A:  
det(A)  
  
0.07755281028277528  
  
# Детерминант матрицы A через объект факторизации:  
det(Alu)  
  
0.07755281028277528
```

Рис. 16: Пример решения с использованием исходной матрицы и с использованием объекта

Факторизация. Специальные матричные структуры

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: 3x3 LinearAlgebra.QRCompactWYQ{Float64, Matrix{Float64}, Matrix{Float64}}  
R factor:  
3x3 Matrix{Float64}:  
-0.984591 -0.683612 -1.16564  
0.0 -0.949974 -0.000997145  
0.0 0.0 0.0902472  
  
# Матрица Q:  
Aqr.Q  
  
3x3 LinearAlgebra.QRCompactWYQ{Float64, Matrix{Float64}, Matrix{Float64}}  
  
# Матрица R:  
Aqr.R  
  
3x3 Matrix{Float64}:  
-0.984591 -0.683612 -1.16564  
0.0 -0.949974 -0.000997145  
0.0 0.0 0.0902472  
  
# Проверка, что матрица Q - ортогональная:  
Aqr.Q'*Aqr.Q  
  
3x3 Matrix{Float64}:  
1.0 -5.55112e-17 1.11022e-16  
-5.55112e-17 1.0 0.0  
1.11022e-16 0.0 1.0
```

Рис. 17: Пример вычисления QR-факторизации и определение составного типа факторизации для его хранения

Факторизация. Специальные матричные структуры

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

Julia

```
3x3 Matrix{Float64}:  
1.47463  0.314984  1.4097  
0.314984  1.82444  0.896656  
1.4097    0.896656  1.41055
```

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

Julia

```
Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}  
values:  
3-element Vector{Float64}:  
-0.07482006126437302  
1.4375774282462976  
3.3468623755633247  
vectors:  
3x3 Matrix{Float64}:  
-0.624328  0.531084  -0.572856  
-0.246421  -0.829792  -0.580722  
0.741277  0.171451  -0.648933
```

```
# Собственные значения:  
AsymEig.values
```

Julia

```
3-element Vector{Float64}:  
-0.07482006126437302  
1.4375774282462976  
3.3468623755633247
```

Рис. 18: Примеры собственной декомпозиции матрицы

Факторизация. Специальные матричные структуры

```
#Собственные векторы:  
AsymEig.vectors
```

Julia

```
3x3 Matrix{Float64}:  
-0.624328   0.531084   -0.572856  
-0.246421   -0.829792   -0.500722  
0.741277    0.171451   -0.648933
```

```
# Проверяем, что получится единичная матрица:  
inv(AsymEig)*Asym
```

Julia

```
3x3 Matrix{Float64}:  
1.0          6.21725e-15  -3.19744e-14  
7.99361e-15  1.0          -1.24345e-14  
-1.77636e-14 -2.66454e-15  1.0
```

Рис. 19: Примеры собственной декомпозиции матрицы

Факторизация. Специальные матричные структуры

Далее рассмотрим примеры работы с матрицами большой размерности и специальной структуры:

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

0.425318	1.22891	-0.849322	...	0.038386	-1.36644	1.74687
0.532148	-0.0683445	1.14957		0.0143723	-0.77164	0.148296
-0.996531	0.404782	-1.32305		-0.011747	-1.32671	-2.45213
-0.289675	1.06816	-1.07935		-0.916511	0.636577	-2.46619
0.449	0.509505	0.961422		0.33811	1.31405	0.685894
1.89516	0.137476	0.384805	...	0.248673	0.790506	0.427286
-1.64042	-0.602952	1.04468		0.56485	-0.228502	-0.657765
-0.167493	-0.476201	-0.119808		1.17856	-1.18736	0.139803
-1.5806	-1.04554	-1.42424		1.30654	-0.317519	-1.66068
-0.653765	2.06532	-1.25557		-1.29709	0.0009651	0.186939
:						
-0.0924705	1.02043	-0.65733		0.305228	1.48423	0.498798
-0.356041	-1.40266	-0.379382		-0.985984	-1.52783	-1.12308
-0.69457	-0.380283	-1.58256		-0.601262	-0.572094	1.14134
0.142481	1.04664	-0.340014		1.04413	-0.900609	1.69761
1.32621	-0.942717	0.406584	...	-1.252	-2.08874	-1.40599
-0.785725	-0.306074	0.11628		-0.823618	0.39381	1.11556
-0.949094	-0.610857	2.12673		-1.93954	-0.812335	1.28528
1.22951	-0.0202556	0.0874981		-1.00452	1.37859	-0.186782
0.531804	-1.53585	-1.08785		0.181619	0.711781	-1.08036

Julia

Рис. 20: Примеры работы с матрицами большой размерности и специальной структуры

Факторизация. Специальные матричные структуры

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

Julia

```
1000x1000 Matrix{Float64}:  
 0.850636  1.75306 -1.84585 ... -0.910708 -0.136933  2.27867  
 1.75306 -0.136689  1.55435   -0.596485 -0.791896 -1.38756  
 -1.84585  1.55435 -2.6461   2.11498 -1.23921 -3.53998  
 -0.848472 -0.42742 -2.61159   -0.753267  0.705361 -1.36016  
 1.50371  1.7017  0.360662   -0.0936283 -0.299328  0.723578  
 2.30978  0.501434 -1.84291 ... 1.27479 -0.31106 -0.877769  
 -1.42366 -0.261772 -0.266711  1.60639  0.302629 -0.568203  
 -1.23758  0.383195 -0.538402  0.358255 -0.848562 -0.381213  
 -2.17664  0.492679 -0.889353 -0.198953  0.422633  0.968967  
 -0.841459  1.70555 -2.67158   -0.610607 -0.54081  0.33299  
 :           :       :  
 0.0330194  0.766703  0.216643  0.662512  3.16748  0.0512782  
 -0.114326 -1.40181 -0.286361 -0.68189 -1.82442 -3.15223  
 -7.10677e-5 0.753181 -0.666509 -1.75376  0.356444  0.596603  
 1.19361  0.6615  0.283538  1.00072 -0.86253  1.50714  
 1.80279 -0.168534  3.24526 ... -0.221696 -2.93415 -1.71958  
 -0.318273 -0.289092  0.667093 -2.4851 -0.302665  0.707986  
 -0.910708 -0.596485  2.11498 -3.87908 -1.81686  1.3869  
 -0.136933 -0.791896 -1.23921 -1.81686  2.75717  0.524999  
 2.27867 -1.38756 -3.53998  1.3869  0.524999 -2.16073
```

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

Julia

true

Рис. 21: Примеры работы с матрицами большой размерности и специальной структуры

Факторизация. Специальные матричные структуры

Пример добавления шума в симметричную матрицу (матрица уже не будет симметричной):

```
# Добавление шума:  
Asym_noisy = copy(Asym)  
Asym_noisy[1,2] += 5eps()  
# Проверка, является ли матрица симметричной:  
issymmetric(Asym_noisy)
```

false

Julia

Рис. 22: Пример добавления шума в симметричную матрицу

Факторизация. Специальные матричные структуры

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

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

Julia

```
1000×1000 Symmetric{Float64, Matrix{Float64}}:  
 0.850636   1.75306  -1.84585  ... -0.910708  -0.136933  2.27867  
 1.75306  -0.136689  1.55435  -0.596485  -0.791896  -1.38756  
 -1.84585   1.55435  -2.6461   2.11498  -1.23921  -3.53998  
 -0.848472  -0.42742  -2.61159  -0.753267  0.705361  -1.36016  
 1.50371   1.7617  0.366662  -0.0936283  -0.299328  0.723578  
 2.30978   0.501434  -1.84291  ... 1.27479  -0.31106  -0.877769  
 -1.42366  -0.261772  -0.266711  1.60639  0.302629  -0.568203  
 -1.23758   0.383195  -0.538402  0.358255  -0.848562  -0.381213  
 -2.17664   0.492679  -0.889353  -0.198953  0.422633  0.968967  
 -0.841459  1.70555  -2.67158  -0.610607  -0.54081  0.33299  
 :          ..  
 0.0330194   0.766703  0.216643  0.662512  3.16748  0.0512782  
 -0.114326  -1.40181  -0.286361  -0.68189  -1.82442  -3.15223  
 -7.10677e-5  0.753181  -0.666509  -1.75376  0.356444  0.596603  
 1.19361   0.6615  0.283538  1.00072  -0.86253  1.50714  
 1.80279  -0.168534  3.24526  ... -0.221696  -2.93415  -1.71958  
 -0.310273  -0.289092  0.667093  -2.4851  -0.302665  0.707986  
 -0.910708  -0.596485  2.11498  -3.87908  -1.81686  1.3869  
 -0.136933  -0.791896  -1.23921  -1.81686  2.75717  0.524999  
 2.27867  -1.38756  -3.53998  1.3869  0.524999  -2.16073
```

Рис. 23: Пример явного объявления структуры матрицы

Факторизация. Специальные матричные структуры

Далее для оценки эффективности выполнения операций над матрицами большой размерности и специальной структуры воспользуемся пакетом BenchmarkTools:

```
PKG.add("BenchmarkTools")
using BenchmarkTools

Resolving package versions...
Installed BenchmarkTools - v1.6.0
Updating `C:\Users\angelina\.julia\environments\v1.11\Project.toml'
[6e4b00f9] + BenchmarkTools v1.6.0
  Updating `C:\Users\angelina\.julia\environments\v1.11\Manifest.toml'
[6e4b00f9] + BenchmarkTools v1.6.0
[9abbd945] + Profile v1.11.0
Precompiling project...
5449.0 ms ✓ BenchmarkTools
1 dependency successfully precompiled in 9 seconds. 452 already precompiled.
```

```
# Оценка эффективности выполнения операции по нахождению
# собственных значений симметрической матрицы:
@time eigvals(Asym);

149.899 ms (21 allocations: 7.99 MiB)
```

```
# Оценка эффективности выполнения операции по нахождению
# собственных значений замкнутой матрицы:
@time eigvals(Asym_noisy);

1.343 s (27 allocations: 7.93 MiB)
```

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

157.929 ms (21 allocations: 7.99 MiB)
```

Рис. 24: Использование пакета BenchmarkTools

Факторизация. Специальные матричные структуры

Далее рассмотрим примеры работы с разреженными матрицами большой размерности.

Использование типов Tridiagonal и SymTridiagonal для хранения трёхдиагональных матриц позволяет работать с потенциально очень большими трёхдиагональными матрицами:

```
# Трёхдиагональная матрица 1000000 x 1000000:  
n = 1000000;  
A = SymTridiagonal(randn(n), randn(n-1))  
# Оценка эффективности выполнения операции по нахождению  
# собственных значений:  
@btime eigmax(A)
```

Julia

```
783.488 ms (44 allocations: 183.11 MiB)
```

```
6.5601813712800485
```

Рис. 25: Примеры работы с разреженными матрицами большой размерности

Общая линейная алгебра

В 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
# Единичный вектор:
x = fill(1, 3)
# Задаём вектор b:
b = Arational*x
# Решение исходного уравнения получаем с помощью функции \
# (убеждаемся, что x - единичный вектор):
Arational\b
# LU-разложение:
lu(Arational)
```

Julia

```
LU{Rational{BigInt}, Matrix{Rational{BigInt}}, Vector{Int64}}
L factor:
3x3 Matrix{Rational{BigInt}}:
 1      0      0
 3//8    1      0
 7//8   -1//5  1
U factor:
3x3 Matrix{Rational{BigInt}}:
 4//5   4//5    3//5
 0     1//2   7//40
 0      0    11//100
```

Рис. 26: Решение системы линейных уравнений с рациональными элементами без преобразования в

Самостоятельная работа

Самостоятельное выполнение

Произведение векторов

1) Задайте вектор v . Умножьте вектор v скалярно сам на себя и сохраните результат в `dot_v`:

```
# Задаем вектор v
v = [1, 3, 5]

# Скалярное произведение
dot_v = dot(v, v)
```

Julia

35

2) Умножьте v матрично на себя (внешнее произведение), присвоив результат переменной `outer_v`:

```
# Матричное (внешнее) произведение
outer_v = v * v'
```

Julia

3x3 Matrix{Int64}:

1	3	5
3	9	15
5	15	25

Рис. 27: Решение задания “Произведение векторов”

Самостоятельная работа

Системы линейных уравнений

1) Решить СЛАУ с двумя неизвестными:

```
function res(A, b)
    if (det(A) == 0)
        println("Нет решения")
    else
        println(A\b)
    end
end
```

Julia

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

```
A1 = [1 1; 1 -1]
b1 = [2; 3]
res(A1, b1)
```

Julia

```
[2.5, -0.5]
```

```
A2 = [1 1; 2 2]
b2 = [2; 4]
res(A2, b2)
```

Julia

```
Нет решения
```

Рис. 28: Решение задания “Системы линейных уравнений”

Самостоятельная работа

```
A3 = [1 1; 2 2]
b3 = [2; 5]
res(A3, b3)
```

Julia

Нет решения

```
A4 = [1 1; 2 2; 3 3]
b4 = [1; 2; 3]
println(A4\b4)
```

Julia

[0.4999999999999999, 0.5]

```
A5 = [1 1; 2 1; 1 -1]
b5 = [2; 1; 3]
println(A5\b5)
```

Julia

[1.500000000000004, -0.9999999999999997]

```
A6 = [1 1; 2 1; 3 2]
b6 = [2; 1; 3]
println(A6\b6)
```

Julia

[-0.999999999999989, 2.999999999999982]

Рис. 29: Решение задания “Системы линейных уравнений”

Самостоятельная работа

2) Решить СЛАУ с тремя неизвестными:

```
A1 = [1 1 1; 1 -1 -2]
b1 = [2; 3]
println(A1\b1)
```

Julia

```
[2.2142857142857144, 0.35714285714285704, -0.5714285714285712]
```

```
A2 = [1 1 1; 2 2 -3; 3 1 1]
b2 = [2; 4; 1]
res(A2, b2)
```

Julia

```
[-0.5, 2.5, 0.0]
```

```
A3 = [1 1 1; 1 1 2; 2 2 3]
b3 = [1; 0; 1]
res(A3, b3)
```

Julia

Нет решения

```
A4 = [1 1 1; 1 1 2; 2 2 3]
b4 = [1; 0; 0]
res(A4, b4)
```

Julia

Нет решения

Самостоятельная работа

Операции с матрицами

1) Приведите приведённые ниже матрицы к диагональному виду:

```
A = [1 -2; -2 1]
eigen_A = eigen(A) # Собственные значения и векторы
diag_matrix = Diagonal(eigen_A.values) # Диагональная матрица
```

Julia

```
2x2 Diagonal{Float64, Vector{Float64}}:
-1.0
   3.0
```

```
B = [1 -2; -2 3]
eigen_B = eigen(B) # Собственные значения и векторы
diag_matrix = Diagonal(eigen_B.values) # Диагональная матрица
```

Julia

```
2x2 Diagonal{Float64, Vector{Float64}}:
-0.236068
   4.23607
```

```
C = [1 -2 0; -2 1 2; 0 2 0]
eigen_C = eigen(C) # Собственные значения и векторы
diag_matrix = Diagonal(eigen_C.values) # Диагональная матрица
```

Julia

```
3x3 Diagonal{Float64, Vector{Float64}}:
-2.14134
   0.515138
   3.6262
```

Рис. 31: Решение задания “Операции с матрицами”

Самостоятельная работа

2) Вычислите:

```
A = [1 -2; -2 1]
display(A^10)
```

2×2 Matrix{Int64}:

29525	-29524
-29524	29525

```
A = [5 -2;
      -2 5]
display(sqrt(A))
```

2×2 Matrix{Float64}:

2.1889	-0.45685
-0.45685	2.1889

```
A = [1 -2;
      -2 1]
display(A^(1/3))
```

2×2 Symmetric{ComplexF64, Matrix{ComplexF64}}:

0.971125+0.433013im	-0.471125+0.433013im
-0.471125+0.433013im	0.971125+0.433013im

```
A = [1 2; 2 3]
display(sqrt(A))
```

2×2 Matrix{ComplexF64}:

0.568864+0.351578im	0.920442-0.217287im
0.920442-0.217287im	1.48931+0.134291im

Рис. 32: Решение задания “Операции с матрицами”

Самостоятельная работа

3) Найдите собственные значения матрицы A . Создайте диагональную матрицу из собственных значений матрицы A . Создайте нижнедиагональную матрицу из матрицы A . Оцените эффективность выполняемых операций:

The screenshot shows a Julia code editor interface with a toolbar at the top. The code in the editor is as follows:

```
# Исходная матрица A
A = [
    140  97  74 168 131;
    97 106  89 131  36;
    74  89 152 144  71;
   168 131 144  54 142;
    131  36  71 142  36
]

# 1. Нахождение собственных значений и векторов
Aeigen = eigen(A)
```

Below the code, the Julia interpreter output is displayed:

```
Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}
values:
5-element Vector{Float64}:
-128.49322764802145
-55.887784553057
42.752167279318854
87.16111477514488
542.467730146614
vectors:
5×5 Matrix{Float64}:
 -0.147575  0.647178  0.010882  0.548903  -0.507907
 -0.256795  -0.173068  0.834628  -0.239864  -0.387253
 -0.185537  0.239762  -0.422161  -0.731925  -0.440631
  0.819704  -0.247506  -0.0273194  0.0366447  -0.514526
 -0.453805  -0.657619  -0.352577  0.322668  -0.364928
```

Рис. 33: Решение задания “Операции с матрицами”

Самостоятельная работа

```
# 2. Создание диагональной матрицы из собственных значений  
# Прямое создание переменной и вывод без использования @btime  
diagm(Aeigen.values)
```

Julia

```
5x5 Matrix{Float64}:  
-128.493   0.0    0.0    0.0    0.0  
 0.0    -55.8878  0.0    0.0    0.0  
 0.0     0.0    42.7522  0.0    0.0  
 0.0     0.0     0.0    87.1611  0.0  
 0.0     0.0     0.0     0.0    542.468
```

```
# 3. Создание нижнедиагональной матрицы из A  
LowerTriangular(A)
```

Julia

```
5x5 LowerTriangular{Int64, Matrix{Int64}}:  
140   .   .   .  
97  106   .   .  
74   89  152   .  
168  131  144   54  
131   36   71  142   36
```

```
# 4. Оценка эффективности  
@btime diagm(Aeigen.values)  
@btime LowerTriangular(A)
```

Julia

```
151.562 ns (2 allocations: 272 bytes)  
274.351 ns (1 allocation: 16 bytes)  
  
5x5 LowerTriangular{Int64, Matrix{Int64}}:  
140   .   .   .  
97  106   .   .  
74   89  152   .  
168  131  144   54  
131   36   71  142   36
```

Рис. 34: Решение задания “Операции с матрицами”

Самостоятельная работа

Линейные модели экономики

- 1) Матрица A называется продуктивной, если решение x системы при любой неотрицательной правой части y имеет только неотрицательные элементы x_i . Используя это определение, проверьте, являются ли матрицы продуктивными;
- 2) Критерий продуктивности: матрица A является продуктивной тогда и только тогда, когда все элементы матрицы $(E - A)^{-1}$ являются неотрицательными числами. Используя этот критерий, проверьте, являются ли матрицы продуктивными;
- 3) Спектральный критерий продуктивности: матрица A является продуктивной тогда и только тогда, когда все её собственные значения по модулю меньше 1. Используя этот критерий, проверьте, являются ли матрицы продуктивными;

```
A1 = [1 2; 3 4]
A2 = (1/2) * A1
A3 = (1/10) * A1
E = Matrix(I, 2, 2)
```

Julia

```
2x2 Matrix{Bool}:
1 0
0 1
```

```
inv(E-A1) #не продуктивная
```

Julia

```
2x2 Matrix{Float64}:
0.5 -0.333333
-0.5 0.0
```

Рис. 35: Решение задания “Линейные модели экономики”

Самостоятельная работа

```
inv(E-A2) #не продуктивная
```

Julia

```
2×2 Matrix{Float64}:
 0.5   -0.5
 -0.75  -0.25
```

```
inv(E-A3) #продуктивная
```

Julia

```
2×2 Matrix{Float64}:
 1.25   0.416667
 0.625  1.875
```

```
A4 = [0.1 0.2 0.3; 0 0.1 0.2; 0 0.1 0.3]
```

Julia

```
3×3 Matrix{Float64}:
 0.1  0.2  0.3
 0.0  0.1  0.2
 0.0  0.1  0.3
```

```
abs.(eigen(A1).values).<1 #не продуктивная
```

Julia

```
2-element BitVector:
 1
 0
```

Рис. 36: Решение задания “Линейные модели экономики”

Самостоятельная работа

```
abs.(eigen(A2).values).<1 #не продуктивная
```

Julia

```
2-element BitVector:  
1  
0
```

```
abs.(eigen(A3).values).<1 #продуктивная
```

Julia

```
2-element BitVector:  
1  
1
```

```
abs.(eigen(A4).values).<1 #продуктивная
```

Julia

```
3-element BitVector:  
1  
1  
1
```

Рис. 37: Решение задания “Линейные модели экономики”

Выводы

В результате выполнения данной лабораторной работы мы изучили возможности специализированных пакетов Julia для выполнения и оценки эффективности операций над объектами линейной алгебры.