Лабораторная работа №4

Линейная алгебра

Дурдалыев Максат

2025-10-25

Российский университет дружбы народов имени Патриса Лумумбы, Москва, Россия

Содержание i

1 Докладчик

- Дурдалыев Максат
- Студент НКНбд-01-22
- Российский университет дружбы народов имени Патриса Лумумбы
- 1132205337@pfur.ru
- https://github.com/mdurdalyyev

2 Цели и задачи

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.

3 Поэлементные операции над многомерными массивами

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

Julia

```
4×3 Matrix{Int64}:
 11  1  20
  8  8   9
  1  4  12
 19  4   5
```

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

Julia

```
102
```

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

Julia

```
1×3 Matrix{Int64}:
 39  17  46
```

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

Julia

```
4×1 Matrix{Int64}:
 32
 25
 17
 28
```


Рисунок 1: Поэлементные операции сложения и произведения элементов матрицы

4 Поэлементные операции над многомерными массивами

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

Julia

```
2311372800
```

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

Julia

```
1×3 Matrix{Int64}:
 1672  128  10800
```

Создать + Code + Markdown

Добавить ячейку кода

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

Julia

```
4×1 Matrix{Int64}:
 220
 576
```

5 Поэлементные операции над многомерными массивами

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

Julia

```
┌ Warning: could not download https://pkg.julialang.org/registries
│   exception = Downloads.RequestError("https://pkg.julialang.org/registries", 6, "Could not res
└ @ Pkg.Registry C:\Users\durda\AppData\Local\Programs\Julia-1.11.7\share\julia\stdlib\v1.11\Pkg
   Updating registry at `C:\Users\durda\.julia\registries\General.toml`
  Resolving package versions...
   Updating `C:\Users\durda\.julia\environments\v1.11\Project.toml`
  [10745b16] + Statistics v1.11.1
No Changes to `C:\Users\durda\.julia\environments\v1.11\Manifest.toml`
```

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

Julia

```
8.5
```

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

Julia

```
1×3 Matrix{Float64}:
 9.75  4.25  11.5
```

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

Julia

```
4×1 Matrix{Float64}:
 10.666666666666666
  8.333333333333334
  5.666666666666667
  9.333333333333334
```

Рисунок 3: Использование возможностей пакета Statistics для работы со средними значениями

6 Транспонирование, след, ранг, определитель и инверсия матрицы

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

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

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

```
4×4 Matrix{Int64}:
  4   9  17  10
 19  17  16   3
 12  17   2   4
 10   5   8   7
```

```julia
# Транспонирование:
transpose(b)
```
Julia

```
4×4 transpose(::Matrix{Int64}) with eltype Int64:
  4  19  12  10
  9  17  17   5
 17  16   2   8
 10   3   4   7
```

Рисунок 4: Использование библиотеки LinearAlgebra для выполнения определённых операций

7 Транспонирование, след, ранг, определитель и инверсия матрицы

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

Julia

```
30
```

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

Julia

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

8 Транспонирование, след, ранг, определитель и инверсия матрицы

```julia
# Ранг матрицы:
rank(b)
```

Julia

```
4
```

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

Julia

```
4×4 Matrix{Float64}:
 -0.0701787   0.0271525   -0.0193084   0.0996519
  0.037967   -0.00278487   0.068647   -0.092272
  0.0372708   0.0510559   -0.0585287  -0.0416802
  0.0305407  -0.0951497    0.0454398   0.11404
```

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

Julia

```
21545.0
```

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

Julia

```
3×4 Matrix{Float64}:
```

9 Вычисление нормы векторов и матриц, повороты, вращения

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

Julia

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

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

Julia

```
6.708203932499369
```

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

Julia

```
11.0
```

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

Julia

```
9.486832980505138
```


Рисунок 7: Использование LinearAlgebra.norm(x)

10 Вычисление нормы векторов и матриц, повороты, вращения

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

Julia

```
9.486832980505138
```

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

Julia

11 Вычисление нормы векторов и матриц, повороты, вращения

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

Julia

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

```julia
# Вычисление Евклидовой нормы:
opnorm(d)
```

Julia

```
7.147682841795258
```

```julia
# Вычисление p-нормы:
p=1
opnorm(d,p)
```

Julia

```
8.0
```

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

Julia

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

Рисунок 9: Вычисление нормы для двумерной матрицы

12 Вычисление нормы векторов и матриц, повороты, вращения

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

Julia

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

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

Julia

```
3×3 Matrix{Int64}:
 2  -4   5
```

13 Матричное умножение, единичная матрица, скалярное произведение

```julia
# Матрица 2x3 со случайными целыми значениями от 1 до 10:
A = rand(1:10,(2,3))
```

```
2×3 Matrix{Int64}:
 5  1  6
 7  9  1
```

```julia
# Матрица 3x4 со случайными целыми значениями от 1 до 10:
B = rand(1:10,(3,4))
```

```
3×4 Matrix{Int64}:
 2  2  5   8
 7  7  6   8
 2  3  1  10
```

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

```
2×4 Matrix{Int64}:
 29  35  37  108
 79  80  90  138
```

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

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

Рисунок 11: Примеры матричного умножения, единичной матрицы и скалярного

14 Матричное умножение, единичная матрица, скалярное произведение

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

Julia

```
-17
```

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

Julia

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

Julia

```
3×3 Matrix{Float64}:
 0.368815    0.879076  0.0452434
 0.00262414  0.892017  0.871027
 0.479004    0.508546  0.0166509
```

```julia
# Задаём единичный вектор:
x = fill(1.0, 3)
# Задаём вектор b:
b = A*x
# Решение исходного уравнения получаем с помощью функции \
# (убеждаемся, что x - единичный вектор):
A\b
```

Julia

```
3-element Vector{Float64}:
 1.0000000000000002
 1.0
 1.0
```

Рисунок 13: Решение систем линейный алгебраических уравнений $Ax = b$

16 Факторизация. Специальные матричные структуры

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

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

```
LU{Float64, Matrix{Float64}, Vector{Int64}}
L factor:
3×3 Matrix{Float64}:
 1.0         0.0       0.0
 0.00547834  1.0       0.0
 0.769963    0.548242  1.0
U factor:
3×3 Matrix{Float64}:
 0.479004  0.508546   0.0166509
 0.0       0.889231   0.870935
 0.0       0.0       -0.445061
```

```julia
# Матрица перестановок:
Alu.P
```

```
3×3 Matrix{Float64}:
 0.0  0.0  1.0
 0.0  1.0  0.0
 1.0  0.0  0.0
```

```julia
# Вектор перестановок:
Alu.p
```

```
3-element Vector{Int64}:
 3
 2
 1
```


Рисунок 14: Пример вычисления LU-факторизации и определение составного типа

```julia
# Матрица L:
Alu.L
```

Julia

```
3×3 Matrix{Float64}:
 1.0         0.0       0.0
 0.00547834  1.0       0.0
 0.769963    0.548242  1.0
```

```julia
# Матрица U:
Alu.U
```

Julia

```
3×3 Matrix{Float64}:
 0.479004  0.508546   0.0166509
 0.0       0.889231   0.870935
 0.0       0.0       -0.445061
```

18 Факторизация. Специальные матричные структуры

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

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

Julia

```
3-element Vector{Float64}:
 1.0000000000000002
 1.0
 1.0
```

```julia
# Решение СЛАУ через объект факторизации:
Alu\b
```

Julia

```
3-element Vector{Float64}:
 1.0000000000000002
 1.0
 1.0
```

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

Julia

```
0.18957141617226508
```

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

Julia

```
0.18957141617226508
```

19 Факторизация. Специальные матричные структуры

Julia позволяет вычислять QR-факторизацию и определяет составной тип факторизации для его хранения:

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

```
LinearAlgebra.QRCompactWY{Float64, Matrix{Float64}, Matrix{Float64}}
Q factor: 3×3 LinearAlgebra.QRCompactWYQ{Float64, Matrix{Float64}, Matrix{Float64}}
R factor:
3×3 Matrix{Float64}:
 -0.604546  -0.943109  -0.0445756
  0.0       -0.968315  -0.808798
  0.0        0.0        0.323837
```

```julia
# Матрица Q:
Aqr.Q
```

```
3×3 LinearAlgebra.QRCompactWYQ{Float64, Matrix{Float64}, Matrix{Float64}}
```

```julia
# Матрица R:
Aqr.R
```

```
3×3 Matrix{Float64}:
 -0.604546  -0.943109  -0.0445756
  0.0       -0.968315  -0.808798
  0.0        0.0        0.323837
```

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

```
3×3 Matrix{Float64}:
 1.0          0.0  2.22045e-16
 5.72459e-17  1.0  0.0
 2.22045e-16  0.0  1.0
```

Рисунок 17: Пример вычисления QR-факторизации и определение составного типа

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

```
3×3 Matrix{Float64}:
 0.73763   0.8817   0.524247
 0.8817    1.78403  1.37957
 0.524247  1.37957  0.0333019
```

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

```
Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}
values:
3-element Vector{Float64}:
 -0.7262364206842493
  0.2752585513733541
  3.005944564298855
vectors:
3×3 Matrix{Float64}:
 -0.0302759   0.910727  -0.411899
 -0.473221   -0.376038  -0.796654
  0.880423   -0.1708    -0.44236
```

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

```
3-element Vector{Float64}:
 -0.7262364206842493
  0.2752585513733541
  3.005944564298855
```

Рисунок 18: Примеры собственной декомпозиции матрицы

21 Факторизация. Специальные матричные структуры

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

Julia

```
3×3 Matrix{Float64}:
 -0.0302759   0.910727  -0.411899
 -0.473221   -0.376038  -0.796654
  0.880423   -0.1708    -0.44236
```

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

Julia

```
3×3 Matrix{Float64}:
  1.0          -1.66533e-15   1.4936e-15
  9.4369e-16    1.0          -4.41661e-15
 -1.16573e-15  -3.10862e-15   1.0
```

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

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

Julia

```
1000×1000 Matrix{Float64}:
  0.0630126  -0.251197     0.0796325  …  -0.442978    1.17073   -2.14755
  0.781128   -1.16548     -0.90138       -0.171808    1.45133   -0.115162
 -0.809731   -0.157451     0.188144       0.304195   -0.206724  -0.54274
 -0.131096    1.63867      0.190479      -1.26302     0.944059   1.32781
 -0.772695    1.9824       0.18503       -0.202502   -0.138186  -0.146277
  0.281455   -0.791349    -0.366522   …  -0.127131    0.542581   1.19349
  0.849647    0.507622     1.1247         0.00268168  0.345812   1.18034
  0.213123   -0.000618445  0.391949      -0.280991    0.41326   -0.836027
 -0.34615    -0.465263     0.269673       1.47107    -0.330542  -0.831115
  0.899631    0.265446     0.681958      -0.567342   -2.63383   -0.829088
  ⋮                                   ⋱
 -1.16336    -0.163909     0.0220217      2.64269     2.60257    0.0110421
  0.785781   -0.403412    -1.9101        -0.44775    -0.912201  -1.23919
 -1.27575    -0.190804     0.943246       0.00440055 -2.48584   -0.3126
```

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

Julia

```
1000×1000 Matrix{Float64}:
  0.126025    0.529932  -0.730099  …   1.14156    1.37564    -2.18103
  0.529932   -2.33096   -1.05883       0.652347   3.08565    -0.777359
 -0.730099   -1.05883    0.376287      1.7045    -0.413621   -1.0388
  0.598695    1.71362   -1.28131      -1.88954    2.86089     3.23633
 -1.65874     1.90243   -1.46164       0.859204  -0.585901    1.01309
 -0.0616244  -0.577444   0.111719  …  -0.11823    0.107333    1.73984
  3.83262     1.03285    0.58751       1.51281    0.477174    0.423242
  0.410881   -1.14214    0.436089     -1.04592   -0.0280401   0.0638841
 -0.518718   -0.734764  -0.720482      0.447524  -1.34249    -0.966866
  1.15758    -0.980827   2.00434      -0.502093  -1.04615    -1.60159
  ⋮                                ⋱
  0.482351    1.01696    1.09086       2.37937    1.40549    -1.26286
 -0.0184439  -0.572385   0.204972     -1.10517   -0.727063   -0.653867
 -0.311058   -2.31479    1.39357       1.50967   -3.21651     2.69152
 -0.494673    2.02499    2.05656      -1.66158    0.491459    2.12003
  0.313642   -1.22436    3.01635   …  -0.395011  -3.01268    -2.07821
 -0.179699   -0.144436  -2.93768       0.350628   0.240106   -0.0677784
  1.14156     0.652347   1.7045       -0.133509  -1.24456    -1.10305
  1.37564     3.08565   -0.413621     -1.24456   -1.79065    -1.65117
 -2.18103    -0.777359  -1.0388       -1.10305   -1.65117    -1.7424
```

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

24 Факторизация. Специальные матричные структуры

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

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

Julia

```
0.529931640365225
```

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

25 Факторизация. Специальные матричные структуры

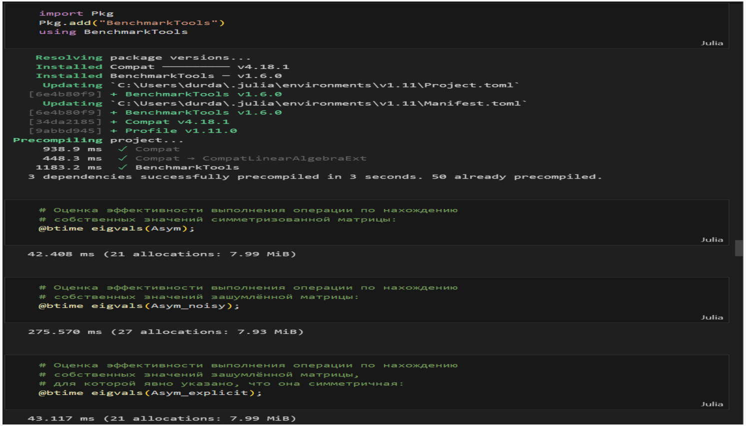
В Julia можно объявить структуру матрица явно, например, используя Diagonal, Triangular, Symmetric, Hermitian, Tridiagonal и SymTridiagonal:

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

Julia

```
1000×1000 Symmetric{Float64, Matrix{Float64}}:
  0.126025    0.529932  -0.730099  …   1.14156    1.37564    -2.18103
  0.529932   -2.33096   -1.05883       0.652347   3.08565    -0.777359
 -0.730099   -1.05883    0.376287      1.7045    -0.413621   -1.0388
  0.598695    1.71362   -1.28131      -1.88954    2.86089     3.23633
 -1.65874     1.90243   -1.46164       0.859204  -0.585901    1.01309
 -0.0616244  -0.577444   0.111719  …  -0.11823    0.107333    1.73984
  3.83262     1.03285    0.58751       1.51281    0.477174    0.423242
  0.410881   -1.14214    0.436089     -1.04592   -0.0280401   0.0638841
 -0.518718   -0.734764  -0.720482      0.447524  -1.34249    -0.966866
  1.15758    -0.980827   2.00434      -0.502093  -1.04615    -1.60159
  ⋮                                ⋱
  0.482351    1.01696    1.09086       2.37937    1.40549    -1.26286
 -0.0184439  -0.572385   0.204972     -1.10517   -0.727063   -0.653867
```

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



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

```
   Resolving package versions...
   Installed Compat ─────────── v4.18.1
   Installed BenchmarkTools ─ v1.6.0
    Updating `C:\Users\durda\.julia\environments\v1.11\Project.toml`
  [6e4b80f9] + BenchmarkTools v1.6.0
    Updating `C:\Users\durda\.julia\environments\v1.11\Manifest.toml`
  [6e4b80f9] + BenchmarkTools v1.6.0
  [34da2185] + Compat v4.18.1
  [9abbd945] + Profile v1.11.0
Precompiling project...
    938.9 ms  ✓ Compat
    448.3 ms  ✓ Compat → CompatLinearAlgebraExt
   1183.2 ms  ✓ BenchmarkTools
  3 dependencies successfully precompiled in 3 seconds. 50 already precompiled.
```

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

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

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

```
  275.570 ms (27 allocations: 7.93 MiB)
```

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

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

Рисунок 24: Использование пакета BenchmarkTools

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:

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

```
1000000×1000000 SymTridiagonal{Float64, Vector{Float64}}:
  1.30813    -0.0463764    ⋅        …   ⋅    ⋅    ⋅
 -0.0463764  -0.310195    0.412643       ⋅    ⋅    ⋅
   ⋅          0.412643    0.51616        ⋅    ⋅    ⋅
   ⋅           ⋅         -1.2908         ⋅    ⋅    ⋅
   ⋅           ⋅           ⋅             ⋅    ⋅    ⋅
   ⋅           ⋅           ⋅        …    ⋅    ⋅    ⋅
   ⋅           ⋅           ⋅             ⋅    ⋅    ⋅
   ⋅           ⋅           ⋅             ⋅    ⋅    ⋅
   ⋅           ⋅           ⋅             ⋅    ⋅    ⋅
   ⋅           ⋅           ⋅             ⋅    ⋅    ⋅
  ⋮                                 ⋱
   ⋅           ⋅           ⋅             ⋅    ⋅    ⋅
   ⋅           ⋅           ⋅             ⋅    ⋅    ⋅
   ⋅           ⋅           ⋅             ⋅    ⋅    ⋅
```

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):

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

```
3×3 Matrix{Rational{BigInt}}:
 1//5    9//10   3//5
 7//10   3//10   3//10
 2//5     1      3//10
```

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

```
3-element Vector{Rational{BigInt}}:
 17//10
 13//10
 17//10
```

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

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

29 Самостоятельная работа

Задания для самостоятельного выполнения

Произведение векторов

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

```julia
# Задаем вектор v
v = [7, 12, 17]

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

Julia

```
482
```

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

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

Julia

```
3×3 Matrix{Int64}:
  49   84  119
  84  144  204
 119  204  289
```

30 Самостоятельная работа

Системы линейных уравнений

1. Решить СЛАУ с двумя неизвестными:

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

Julia

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

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

Julia

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

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

31 Самостоятельная работа

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

Julia

```
Нет решения
```

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

Julia

```
[0.49999999999999, 0.5]
```

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

Julia

```
[1.5000000000000004, -0.9999999999999997]
```

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

32 Самостоятельная работа

2. Решить СЛАУ с тремя неизвестными:

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

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

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

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

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

```
Нет решения
```

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

Операции с матрицами

1. Приведите приведённые ниже матрицы к диагональному виду:

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

Julia

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

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

Julia

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

```julia
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

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

Рисунок 31: Решение задания «Операции с матрицами»

34 Самостоятельная работа

2. Вычислите:

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

```
2×2 Matrix{Int64}:
  29525  -29524
 -29524   29525
```

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

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

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

```
2×2 Symmetric{ComplexF64, Matrix{ComplexF64}}:
  0.971125+0.433013im  -0.471125+0.433013im
 -0.471125+0.433013im   0.971125+0.433013im
```

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

```
2×2 Matrix{ComplexF64}:
 0.568864+0.351578im  0.920442-0.217287im
 0.920442-0.217287im   1.48931+0.134291im
```

Рисунок 32: Решение задания «Операции с матрицами»

35 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.

```julia
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
A_eigen = eigen(A)
```

Julia

```
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
```

36 Самостоятельная работа

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

Julia

```
5×5 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

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

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

Julia

```
  59.045 ns (2 allocations: 272 bytes)
  107.945 ns (1 allocation: 16 bytes)

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

Рисунок 34: Решение задания «Операции с матрицами»

37 Самостоятельная работа

Линейные модели экономики

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

Julia

```
2×2 Matrix{Bool}:
 1  0
 0  1
```

```julia
inv(E-A1) # не продуктивная
```

Julia

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

Рисунок 35: Решение задания «Линейные модели экономики»

38 Самостоятельная работа

```julia
inv(E-A2) # не продуктивная
```

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

```julia
inv(E-A3) # продуктивная
```

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

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

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

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

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

39 Самостоятельная работа

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

Julia

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

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

Julia

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

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

Julia

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

40 Выводы

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