Отчет по лабораторной работе 4

Петрушов Дмитрий, 1032212287

Содержание

Список иллюстраций

Список таблиц

1 Цель работы

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

2 Выполнение лабораторной работы

2.1 Поэлементные операции над многомерными массивами

Для матрицы 4×3 рассмотрим поэлементные операции сложения и произведения её элементов (рис. [2.1]):

```julia
# Массив 4х3 со случайными целыми числами (от 1 до 20):
a = rand(1:20,(4,3))
```

```
4×3 Matrix{Int64}:
 10   9   4
  7  15   6
 18  12  19
 17   3  19
```

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

```
139
```

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

```
1×3 Matrix{Int64}:
 52  39  48
```

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

```
4×1 Matrix{Int64}:
 23
 28
 49
 39
```

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

```
901932796800
```

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

```
1×3 Matrix{Int64}:
 21420  4860  8664
```

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

```
4×1 Matrix{Int64}:
  360
  630
 4104
  969
```

Рис. 2.1: Поэлементные операции сложения и произведения элементов матрицы

Для работы со средними значениями можно воспользоваться возможностями пакета Statistics (рис. [2.2]):

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

```
   Resolving package versions...
  No Changes to `~/.julia/environments/v1.10/Project.toml`
  No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

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

```
11.583333333333334
```

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

```
1×3 Matrix{Float64}:
 13.0  9.75  12.0
```

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

```
4×1 Matrix{Float64}:
  7.666666666666667
  9.333333333333334
 16.333333333333332
 13.0
```

Рис. 2.2: Использование возможностей пакета Statistics для работы со средними значениями

2.2 Транспонирование, след, ранг, определитель и инверсия матрицы

Для выполнения таких операций над матрицами, как транспонирование, диагонализация, определение следа, ранга, определителя матрицы и т.п. можно воспользоваться библиотекой (пакетом) LinearAlgebra (рис. [2.3] - рис. [2.4]):

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

```
   Resolving package versions...
    Updating `~/.julia/environments/v1.10/Project.toml`
  [37e2e46d] + LinearAlgebra
  No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

```
b = rand(1:20,(4,4))
```

```
4×4 Matrix{Int64}:
  8  15   3  16
 19  12   5  20
  6  17  12  11
  9  17   8   4
```

```
transpose(b)
```

```
4×4 transpose(::Matrix{Int64}) with eltype Int64:
  8  19   6   9
 15  12  17  17
  3   5  12   8
 16  20  11   4
```

```
tr(b)
```

```
36
```

```
b
```

```
4×4 Matrix{Int64}:
  8  15   3  16
 19  12   5  20
  6  17  12  11
  9  17   8   4
```

```
diag(b)
```

```
4-element Vector{Int64}:
  8
 12
 12
  4
```

```
rank(b)
```

```
4
```

Рис. 2.3: Использование библиотеки LinearAlgebra для выполнения определённых операций

```
inv(b)
```

```
4×4 Matrix{Float64}:
 -0.0625502   0.066871   -0.0526527   0.0606411
  0.0796322  -0.0530044  -0.045418    0.0713927
 -0.119624    0.0323051   0.133842   -0.0510953
  0.0415494   0.010199    0.0438103  -0.0876708
```

```
det(b)
```

```
19904.0
```

```
pinv(a)
```

```
3×4 Matrix{Float64}:
  0.183203   -0.0856021  -0.0366869   0.0251502
 -0.0211583   0.0723013   0.0178931  -0.0362707
 -0.160409    0.0519753   0.0504535   0.019535
```

Рис. 2.4: Использование библиотеки LinearAlgebra для выполнения определённых операций

2.3 Вычисление нормы векторов и матриц, повороты, вращения

Для вычисления нормы используется LinearAlgebra.norm(x) (рис. [2.5]):

```julia
X = [2, 4, -5]
```

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

```julia
norm(X)
```

```
6.708203932499369
```

```julia
p = 1
norm(X,p)
```

```
11.0
```

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

```
9.486832980505138
```

```julia
sqrt(sum((X-Y).^2))
```

```
9.486832980505138
```

```julia
acos((transpose(X)*Y)/(norm(X)*norm(Y)))
```

```
2.4404307889469252
```

```julia
# 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
```

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

```
7.147682841795258
```

Рис. 2.5: Использование LinearAlgebra.norm(x)

Вычислим нормы для двумерной матрицы (рис. [2.6]):

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

```
8.0
```

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

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

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

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

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

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

Рис. 2.6: Вычисление нормы для двумерной матрицы

2.4 Матричное умножение, единичная матрица, скалярное произведение

Выполним примеры матричного умножения, единичной матрицы и скалярного произведения (рис. [2.7]):

```
# Матрица 2x3 со случайными целыми значениями от 1 до 10:
A = rand(1:10,(2,3))
# Матрица 3x4 со случайными целыми значениями от 1 до 10:
B = rand(1:10,(3,4))
```

```
3×4 Matrix{Int64}:
  4  4  10  7
 10  7   4  9
  5  3   1  4
```

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

```
2×4 Matrix{Int64}:
 51  46  95  76
 50  39  58  60
```

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

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

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

```
-17
```

```
X'Y
```

```
-17
```

Рис. 2.7: Примеры матричного умножения, единичной матрицы и скалярного произведения

2.5 Факторизация. Специальные матричные структуры

Рассмотрим несколько примеров. Для работы со специальными матричными структурами потребуется пакет LinearAlgebra.

Решение систем линейный алгебраических уравнений Ax = b (рис. [2.8]):

```
# Задаём квадратную матрицу 3x3 со случайными значениями:
A = rand(3, 3)
# Задаём единичный вектор:
x = fill(1.0, 3)
# Задаём вектор b:
b = A*x
```

```
3-element Vector{Float64}:
 0.5732494404127212
 2.080767045490446
 1.9703158421826839
```

```
A\b
```

```
3-element Vector{Float64}:
 1.0
 1.0000000000000009
 0.9999999999999992
```

Рис. 2.8: Решение систем линейный алгебраических уравнений Ax = b

Julia позволяет вычислять LU-факторизацию и определяет составной тип факторизации для его хранения (рис. [2.9]):

```
Alu = lu(A)
```

```
LU{Float64, Matrix{Float64}, Vector{Int64}}
L factor:
3×3 Matrix{Float64}:
 1.0         0.0        0.0
 0.0269437   1.0        0.0
 0.917791   -0.123951   1.0
U factor:
3×3 Matrix{Float64}:
 0.848437  0.627937  0.493943
 0.0       0.266125  0.254037
 0.0       0.0       0.336903
```

```
A\b
```

```
3-element Vector{Float64}:
 1.0
 1.0000000000000009
 0.9999999999999992
```

Рис. 2.9: Пример вычисления LU-факторизации и определение составного типа факторизации для его хранения

Исходная система уравнений Ax = b может быть решена или с использованием исходной матрицы, или с использованием объекта факторизации (рис. [2.10]):

```
Alu\b
```

```
3-element Vector{Float64}:
 1.0
 1.0000000000000009
 0.9999999999999992
```

```
det(A)
```

```
0.07606943097699846
```

```
det(Alu)
```

```
0.07606943097699846
```

Рис. 2.10: Пример решения с использованием исходной матрицы и с использованием объекта факторизации

Julia позволяет вычислять QR-факторизацию и определяет составной тип факторизации для его хранения (рис. [2.11]):

```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}:
 -1.15183  -0.835466  -0.882089
  0.0       0.267621   0.228423
  0.0       0.0       -0.246774
```

```julia
# Матрица Q:
Aqr.Q
```

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

```julia
# Матрица R:
Aqr.R
```

```
3×3 Matrix{Float64}:
 -1.15183  -0.835466  -0.882089
  0.0       0.267621   0.228423
  0.0       0.0       -0.246774
```

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

```
3×3 Matrix{Float64}:
  1.0          1.11022e-16  1.11022e-16
  0.0          1.0          0.0
 -1.11022e-16  1.11022e-16  1.0
```

Рис. 2.11: Пример вычисления QR-факторизации и определение составного типа факторизации для его хранения

Примеры собственной декомпозиции матрицы A (рис. [2.12]):

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

```
3×3 Matrix{Float64}:
 0.04572  1.06173  1.11578
 1.06173  1.08666  1.38669
 1.11578  1.38669  0.987885
```

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

```
Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}
values:
3-element Vector{Float64}:
 -0.7191243007910753
 -0.341120347379097
  3.180506297967841
vectors:
3×3 Matrix{Float64}:
  0.886933   0.138077  -0.440777
 -0.202802  -0.740963  -0.640192
 -0.414995   0.657198  -0.629182
```

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

```
3-element Vector{Float64}:
 -0.7191243007910753
 -0.341120347379097
  3.180506297967841
```

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

```
3×3 Matrix{Float64}:
  0.886933   0.138077  -0.440777
 -0.202802  -0.740963  -0.640192
 -0.414995   0.657198  -0.629182
```

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

```
3×3 Matrix{Float64}:
  1.0           1.11022e-16  2.22045e-16
  1.77636e-15   1.0          2.88658e-15
 -2.22045e-16  -2.22045e-16  1.0
```

Рис. 2.12: Примеры собственной декомпозиции матрицы A

Далее рассмотрим примеры работы с матрицами большой размерности и специальной структуры (рис. [2.13]):

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

```
1000×1000 Matrix{Float64}:
  0.337609    0.519499   -0.0323985  …  -0.113286    1.16207    -0.902527
 -0.2188     -1.32053     0.74293        1.15656    -0.693737   -1.50424
  0.73995     0.489306   -0.155427      -0.38398     0.916757    0.197632
  0.661647    0.772271    0.560689      -0.291619   -0.888305   -0.86824
 -1.42908     1.37896    -0.352631      -0.315351    0.572817   -0.950086
 -1.21051    -0.108012   -0.018391   …   0.274048    0.138894   -1.3415
 -1.57722    -0.231669   -1.10252        1.71054    -1.23879     1.13968
  0.78761     0.148086    1.38477        0.141541    0.442647   -0.590033
 -0.316923    0.0189866   0.173869      -0.278284   -0.249112    2.1763
  0.674051   -0.995389   -0.432934      -0.549632   -0.955053    0.336599
  0.149511    0.0812497   0.819825   …  -0.0853111   1.85379    -0.616313
 -0.108787    0.133868    0.0348718     -0.941855    0.20403    -0.564148
  1.17534    -0.814979   -1.1642         2.46189    -0.982499   -0.866496
  ⋮                                  ⋱
 -1.34399    -0.366171    0.0764627      1.54551     0.688253    1.1832
  0.642523    0.242868   -0.517473       1.56176     0.605236   -1.86025
  0.434655    0.766454    0.348385   …   0.40857    -1.20144    -0.869735
  1.92159     0.692595    0.37754        0.686428    0.607705   -0.894495
 -0.176122   -1.91919     0.435952      -0.962394   -0.773402    0.308617
  0.340221    1.18746    -0.85093        0.281541   -0.524794   -1.20695
  0.0629363   0.571605    0.856809       0.527262   -1.02999    -0.397605
  0.404685   -0.802523   -0.690769   …   0.565868   -0.939816   -0.814928
  1.14042     0.651484   -1.58183        0.930157    1.0231      1.65395
 -0.374466    0.30529    -0.127405       0.730432   -0.0799336  -1.63212
 -1.73929     1.72223    -0.151844      -1.09394     0.388045    0.218527
  0.903461   -1.02294     2.4762         0.591404   -0.793751    0.220271
```

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

```
1000×1000 Matrix{Float64}:
  0.675218    0.300699    0.707551  …  -0.487752  -0.57722     0.000934153
  0.300699   -2.64105     1.23224       1.46185    1.02849    -2.52718
  0.707551    1.23224    -0.310854     -0.511385   0.764913    2.67383
 -0.716876    0.425903    0.554105     -0.816669  -2.26906     0.0325796
 -0.222348    1.11693    -0.411437     -0.797233   0.777265   -1.73345
 -0.567717   -1.30094     1.56093   …   1.01361    1.45176    -1.43803
 -2.58906    -1.46641    -1.25445       1.85492   -1.14867     1.21156
 -0.632443   -1.04371     1.21026      -1.01617    0.0761419   0.784158
 -1.37968    -0.656882   -0.857493     -0.362103   0.192844    2.87366
  0.42651    -2.66576     0.348209     -0.224631  -1.78817     0.408209
  0.404985    1.66609     0.695257  …   0.700882   0.804884   -0.0779667
  1.27167     0.142547    0.702403     -0.119727  -0.727482   -1.2615
  2.33387    -1.39398    -0.399441      2.79609   -1.51322     0.379328
```

Рис. 2.13: Примеры работы с матрицами большой размерности и специальной структуры

Пример добавления шума в симметричную матрицу (матрица уже не будет симметричной) (рис. [2.14]):

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

```
0.30069943521674025
```

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

```
false
```

Рис. 2.14: Пример добавления шума в симметричную матрицу

В Julia можно объявить структуру матрица явно, например, используя Diagonal, Triangular, Symmetric, Hermitian, Tridiagonal и SymTridiagonal (рис. [2.15]):

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

```
1000×1000 Symmetric{Float64, Matrix{Float64}}:
  0.675218     0.300699    0.707551  …  -0.487752  -0.57722     0.000934153
  0.300699    -2.64105     1.23224       1.46185    1.02849    -2.52718
  0.707551     1.23224    -0.310854     -0.511385   0.764913    2.67383
 -0.716876     0.425903    0.554105     -0.816669  -2.26906     0.0325796
 -0.222348     1.11693    -0.411437     -0.797233   0.777265   -1.73345
 -0.567717    -1.30094     1.56093   …   1.01361    1.45176    -1.43803
 -2.58906     -1.46641    -1.25445       1.85492   -1.14867     1.21156
 -0.632443    -1.04371     1.21026      -1.01617    0.0761419   0.784158
 -1.37968     -0.656882   -0.857493     -0.362103   0.192844    2.87366
  0.42651     -2.66576     0.348209     -0.224631  -1.78817     0.408209
  0.404985     1.66609     0.695257  …   0.700882   0.804884   -0.0779667
  1.27167      0.142547    0.702403     -0.119727  -0.727482   -1.2615
  2.33387     -1.39398    -0.399441      2.79609   -1.51322     0.379328
  ⋮                                  ⋱
 -0.54303      0.688701    0.177335      2.74886    1.06124     1.79538
  0.143957    -2.90522     0.154359      0.605012   0.945765   -1.88668
 -1.54912      0.0316708   0.683797  …  -1.77238   -1.79562    -1.02596
  1.66346      1.25867    -1.78322       0.666148  -0.148247   -1.55734
 -0.88714     -2.60349     2.31677      -1.93975   -1.95998     1.51839
 -0.794317     0.218783   -0.439028      0.676049  -0.0609014  -3.09433
  0.536423    -0.539068    0.143889      1.04808   -2.40122    -0.880462
  1.36239     -1.86536    -1.13652   …   1.29246   -1.87251     0.252714
  2.41441      1.53582     0.210913      2.12367   -0.0533158   0.184922
 -0.487752     1.46185    -0.511385      1.46086   -1.17387    -1.04072
 -0.57722      1.02849     0.764913     -1.17387    0.77609    -0.575225
  0.000934153 -2.52718     2.67383      -1.04072   -0.575225    0.440542
```

Рис. 2.15: Пример явного объявления структуры матрицы

Далее для оценки эффективности выполнения операций над матрицами большой размерности и специальной структуры воспользуемся пакетом BenchmarkTools (рис. [2.16]):

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

```
   Resolving package versions...
   Installed BenchmarkTools ─ v1.5.0
    Updating `~/.julia/environments/v1.10/Project.toml`
  [6e4b80f9] + BenchmarkTools v1.5.0
    Updating `~/.julia/environments/v1.10/Manifest.toml`
  [6e4b80f9] + BenchmarkTools v1.5.0
  [9abbd945] + Profile
Precompiling project...
  ✓ BenchmarkTools
  1 dependency successfully precompiled in 1 seconds. 432 already precompiled.
```

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

```
  34.330 ms (11 allocations: 7.99 MiB)
```

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

```
  300.674 ms (14 allocations: 7.93 MiB)
```

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

```
  35.085 ms (11 allocations: 7.99 MiB)
```

Рис. 2.16: 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 (рис. [2.17]):

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

```
1000000×1000000 SymTridiagonal{Float64, Vector{Float64}}:
 -0.0581558   0.901308    ⋅         …   ⋅            ⋅           ⋅
  0.901308   -0.12555   -0.327721       ⋅            ⋅           ⋅
   ⋅         -0.327721   0.9198         ⋅            ⋅           ⋅
   ⋅           ⋅         0.93155        ⋅            ⋅           ⋅
   ⋅           ⋅          ⋅             ⋅            ⋅           ⋅
   ⋅           ⋅          ⋅         …   ⋅            ⋅           ⋅
   ⋅           ⋅          ⋅             ⋅            ⋅           ⋅
   ⋅           ⋅          ⋅             ⋅            ⋅           ⋅
   ⋅           ⋅          ⋅             ⋅            ⋅           ⋅
   ⋅           ⋅          ⋅             ⋅            ⋅           ⋅
   ⋅           ⋅          ⋅         …   ⋅            ⋅           ⋅
   ⋅           ⋅          ⋅             ⋅            ⋅           ⋅
   ⋅           ⋅          ⋅             ⋅            ⋅           ⋅
  ⋮                                 ⋱
   ⋅           ⋅          ⋅             ⋅            ⋅           ⋅
   ⋅           ⋅          ⋅             ⋅            ⋅           ⋅
   ⋅           ⋅          ⋅         …   ⋅            ⋅           ⋅
   ⋅           ⋅          ⋅             ⋅            ⋅           ⋅
   ⋅           ⋅          ⋅             ⋅            ⋅           ⋅
   ⋅           ⋅          ⋅             ⋅            ⋅           ⋅
   ⋅           ⋅          ⋅             ⋅            ⋅           ⋅
   ⋅           ⋅          ⋅         …   ⋅            ⋅           ⋅
   ⋅           ⋅          ⋅            1.19599       ⋅           ⋅
   ⋅           ⋅          ⋅           -1.03837      0.00446236   ⋅
   ⋅           ⋅          ⋅            0.00446236   0.49241    -1.47458
   ⋅           ⋅          ⋅             ⋅          -1.47458    -1.0589
```

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

```
  433.600 ms (17 allocations: 183.11 MiB)

5.998410022558604
```

```julia
# 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
```

```
3-element Vector{Rational{BigInt}}:
   2
```

Рис. 2.17: Примеры работы с разряженными матрицами большой размерности

2.6 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) (рис. [2.18]):

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

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

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

```
LU{Rational{BigInt}, Matrix{Rational{BigInt}}, Vector{Int64}}
L factor:
3×3 Matrix{Rational{BigInt}}:
  1       0     0
 8//9     1     0
 4//9  -32//71  1
U factor:
3×3 Matrix{Rational{BigInt}}:
 9//10     1       1//10
  0     -71//90   73//90
  0        0     867//710
```

Рис. 2.18: Решение системы линейных уравнений с рациональными элементами без преобразования в типы элементов с плавающей запятой

2.7 Самостоятельная работа

Выполнение задания “Произведение векторов” (рис. [2.19]):

```
v=[3,2,6]
dot_v=dot(v,v)
```

```
49
```

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

```
3×3 Matrix{Int64}:
  9   6  18
  6   4  12
 18  12  36
```

Рис. 2.19: Решение задания “Произведение векторов”

Выполнение задания “Системы линейных уравнений” (рис. [2.20] - рис. [2.21]):

```
#1
a= [1 1; 1 -1]
b=[2; 3]
a\b
```

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

```
#2
a=[1 1; 2 2]
b=[2;4]
a\b
```

```
SingularException(2)
```

Рис. 2.20: Решение задания “Системы линейных уравнений”

```
#4
a=[1 1; 2 2; 3 3]
b=[1;2;3]
a\b
```

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

```
#5
a=[1 1; 2 1; 1 -1]
b= [2;1;3]
a\b
```

```
2-element Vector{Float64}:
  1.5000000000000004
 -0.9999999999999997
```

```
#6
a=[1 1; 2 1; 3 2]
b= [2;1;3]
a\b
```

```
2-element Vector{Float64}:
 -0.9999999999999994
  2.999999999999999
```

Рис. 2.21: Решение задания “Системы линейных уравнений”

```
#1
a=[1 1 1; 1 -1 -2]
b= [2;3]
a\b
```

```
3-element Vector{Float64}:
  2.214285714285715
  0.35714285714285715
 -0.5714285714285711
```

```
a=[1 1 1; 2 2 -3;3 1 1]
b=[2;4;1]
a\b
```

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

```
#3
a=[1 1 1; 1 1 2; 2 2 3]
b=[1;0;1]
a\b
```

```
SingularException(2)
```

Рис. 2.22: Решение задания “Системы линейных уравнений”

```
#4
a=[1 1 1; 1 1 2; 2 2 3]
b=[1;0;0]
a\b
```

```
SingularException(2)
```

Рис. 2.23: Решение задания “Системы линейных уравнений”

Выполнение задания “Операции с матрицами” (рис. [2.22] - рис. [2.26]):

```
function dm(m)
    Asym= m + m'
    Asyme= eigen(Asym)
    m1= inv(Asyme.vectors) * m * Asyme.vectors
    return m1
end
```

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

```
#1
m=[1 -2; -2 1]
dm(m)
```

```
2×2 Matrix{Float64}:
 -1.0  0.0
  0.0  3.0
```

```
#2
m=[1 -2; -2 3]
dm(m)
```

```
2×2 Matrix{Float64}:
 -0.236068     3.46945e-16
  4.44089e-16  4.23607
```

```
#3
m=[1 -2 0; -2 1 2; 0 2 0]
dm(m)
```

```
3×3 Matrix{Float64}:
 -2.14134       3.55271e-15  -1.9984e-15
  3.38618e-15   0.515138      1.11022e-16
 -7.77156e-16  -5.55112e-16   3.6262
```

Рис. 2.24: Решение задания “Операции с матрицами”

```julia
([1 -2; -2 1])^10
```

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

```julia
([5 -2; -2 5])^0.5
```

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

```julia
([1 -2; -2 1])^(1/3)
```

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

```julia
([1 2; 2 3])^0.5
```

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

Рис. 2.25: Решение задания “Операции с матрицами”

```
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]
b= eigvals(a)
```

```
5-element Vector{Float64}:
 -128.49322764802136
  -55.88778455305686
   42.75216727931889
   87.16111477514512
  542.4677301466143
```

```
@btime b= eigvals(a);
```

```
  1.639 μs (10 allocations: 2.59 KiB)
```

Рис. 2.26: Решение задания “Операции с матрицами”

Выполнение задания “Линейные модели экономики” (рис. [2.27]):

```
function pm(m,a)
    E=[1 0; 0 1]
    Y= E-m
    C= rand(0:100,a)
    X=Y\C
    for i in 1:1:a
        if X[i] < 0
            return "no"
        else
            return "yes"
        end
    end
end
```

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

```
m=[1 2; 3 4]
pm(m, 2)
```

```
"yes"
```

```
m=[1 2; 3 4] *0.5
pm(m, 2)
```

```
"yes"
```

```
m=[1 2; 3 4] * 0.1
pm(m, 2)
```

```
"yes"
```

Рис. 2.27: Решение задания “Линейные модели экономики”

3 Вывод

В ходе выполнения лабораторной работы были изучены возможности специализированных пакетов Julia для выполнения и оценки эффективности операций над объектами линейной алгебры.

Список литературы