Линейная алгебра

Лабораторная работа № 4

Шулуужук А. В.

25 октября 2025

Российский университет дружбы народов, Москва, Россия

Цели и задачи

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

Выполнение лабораторной работы

Поэлементные операции над многомерными массивами

```
[18]: # Для матрицы 4 × 3 рассмотрим поэлементные операции сложения и произведения её элементов:
      # Массив 4x3 со случайными целыми числами (от 1 до 20):
      a = rand(1:20,(4,3))

[18]: 4×3 Matrix{Int64}:
        1  15  14
       11   3   5
       19  16   8
        9   2   2

[20]: # Поэлементная сумма:
      sum(a)

[20]: 105

[22]: # Поэлементная сумма по столбцам:
      sum(a,dims=1)

[22]: 1×3 Matrix{Int64}:
       40  36  29

[24]: # Поэлементная сумма по строкам:
      sum(a,dims=2)

[24]: 4×1 Matrix{Int64}:
       30
       19
       43
       13

[26]: # Поэлементное произведение:
      prod(a)

[26]: 3033676800

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

[28]: 1×3 Matrix{Int64}:
       1881  1440  1120
```

Рис. 1: Поэлементные операции над многомерными массивами

Поэлементные операции над многомерными массивами

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

[30]: 4×1 Matrix{Int64}:
        210
        165
       2432
         36

[32]: # Для работы со средними значениями можно воспользоваться возможностями пакета Statistics:
      # Подключение пакета Statistics:
      import Pkg
      Pkg.add("Statistics")
      using Statistics

          Updating registry at `C:\Users\airan\.julia\registries\General.toml`
         Resolving package versions...
          Updating `C:\Users\airan\.julia\environments\v1.11\Project.toml`
        [10745b16] + Statistics v1.11.1
        No Changes to `C:\Users\airan\.julia\environments\v1.11\Manifest.toml`

[42]: # Вычисление среднего значения массива:
      mean(a)

[42]: 8.75

[44]: # Среднее по столбцам:
      mean(a,dims=1)

[44]: 1×3 Matrix{Float64}:
       10.0  9.0  7.25

[46]: # Среднее по строкам:
      mean(a,dims=2)

[46]: 4×1 Matrix{Float64}:
       10.0
        6.333333333333333
       14.333333333333334
        4.333333333333333
```

Рис. 2: Поэлементные операции над многомерными массивами

Транспонирование, след, ранг, определитель и инверсия матрицы

```
import Pkg
Pkg.add("LinearAlgebra")
using LinearAlgebra
# Массив 4x4 со случайными целыми числами (от 1 до 20):
b = rand(1:20,(4,4))
```

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

```
[53]: 4x4 Matrix{Int64}:
       20  11   8   9
       15   8  15  18
       15  11   2  12
        2  18  14  11
```

```
[54]: # Транспонирование:
      transpose(b)
```

```
[54]: 4x4 transpose(::Matrix{Int64}) with eltype Int64:
       20  15  15   2
       11   8  11  18
        8  15   2  14
        9  18  12  11
```

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

```
[55]: 41
```

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

```
[56]: 4-element Vector{Int64}:
       20
        8
        2
       11
```

Рис. 3: Транспонирование, след, ранг, определитель и инверсия матрицы

```
[61]: # Ранг матрицы:
      rank(b)

[61]: 4

[63]: # Инверсия матрицы (определение обратной матрицы):
      inv(b)

[63]: 4×4 Matrix{Float64}:
        0.0682294  -0.00618964  -0.01407    -0.0303464
        0.0171075  -0.0628707    0.0330114   0.0528699
        0.0547612   0.0349314   -0.110411    0.018483
       -0.110095    0.0595467    0.0890621  -0.0136115

[67]: # Определитель матрицы:
      det(b)

[67]: 34897.0

[69]: # Псевдобратная функция для прямоугольных матриц:
      pinv(a)

[69]: 3×4 Matrix{Float64}:
       -0.0281416    0.0434907   0.0143904   0.030703
       -0.00358715  -0.121284    0.0951274  -0.0521903
        0.0761469    0.131156   -0.101729    0.0459998
```

Рис. 4: Транспонирование, след, ранг, определитель и инверсия матрицы

Вычисление нормы векторов и матриц, повороты, вращения

```
[74]: # Для вычисления нормы используется LinearAlgebra.norm(x).
      # Создание вектора X:
      X = [2, 4, -5]

[74]: 3-element Vector{Int64}:
        2
        4
       -5

[76]: # Вычисление евклидовой нормы:
      norm(X)

[76]: 6.708203932499369

[78]: # Вычисление p-нормы:
      p = 1
      norm(X,p)

[78]: 11.0

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

[80]: 9.486832980505138

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

[82]: 9.486832980505138

[84]: # Угол между двумя векторами:
      acos((transpose(X)*Y)/(norm(X)*norm(Y)))

[84]: 2.4404307889469252
```

Рис. 5: Вычисление нормы векторов и матриц, повороты, вращения

Вычисление нормы векторов и матриц, повороты, вращения

```
•[86]: # Вычисление нормы для двумерной матрицы:
       d = [5 -4 2 ; -1 2 3; -2 1 0]

 [86]: 3×3 Matrix{Int64}:
         5  -4  2
        -1   2  3
        -2   1  0

 [88]: # Вычисление Евклидовой нормы:
       opnorm(d)

 [88]: 7.147682841795258

 [90]: # Вычисление p-нормы:
       p=1
       opnorm(d,p)

 [90]: 8.0

 [92]: # Поворот на 180 градусов:
       rot180(d)

 [92]: 3×3 Matrix{Int64}:
        0   1  -2
        3   2  -1
        2  -4   5

 [94]: # Переворачивание строк:
       reverse(d,dims=1)

 [94]: 3×3 Matrix{Int64}:
        -2   1  0
        -1   2  3
         5  -4  2

 [96]: # Переворачивание столбцов
       reverse(d,dims=2)

 [96]: 3×3 Matrix{Int64}:
        2  -4   5
        3   2  -1
        0   1  -2
```

Рис. 6: Вычисление нормы векторов и матриц, повороты, вращения

Матричное умножение, единичная матрица, скалярное произведение

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

[101]: 2×3 Matrix{Int64}:
        7  6  9
        3  9  2

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

[103]: 3×4 Matrix{Int64}:
        10  6  7   9
         1  9  4  10
         5  7  3   2

[105]: # Произведение матриц A и B:
       A*B

[105]: 2×4 Matrix{Int64}:
        121  159  100  141
         49  113   63  121

[107]: # Единичная матрица 3x3:
       Matrix{Int}(I, 3, 3)

[107]: 3×3 Matrix{Int64}:
        1  0  0
        0  1  0
        0  0  1

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

[109]: -17

[111]: # тоже скалярное произведение:
       X'Y

[111]: -17
```

Рис. 7: Матричное умножение, единичная матрица, скалярное произведение

Факторизация. Специальные матричные структуры

```
[116]: # Решение систем линейный алгебраических уравнений Ax = b:
       # Задаём квадратную матрицу 3x3 со случайными значениями:
       A = rand(3, 3)

[116]: 3×3 Matrix{Float64}:
        0.886681  0.401979  0.916675
        0.262331  0.945059  0.628263
        0.851961  0.99202   0.538704

[118]: # Задаём единичный вектор:
       x = fill(1.0, 3)

[118]: 3-element Vector{Float64}:
        1.0
        1.0
        1.0

[120]: # Задаём вектор b:
       b = A*x

[120]: 3-element Vector{Float64}:
        2.2053346737284576
        1.835653219998258
        2.382684297793384

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

[122]: 3-element Vector{Float64}:
        1.0000000000000007
        1.0
        0.9999999999999998
```

Рис. 8: Решение систем линейный алгебраических уравнений Ax = b

```
[126]: # Julia позволяет вычислять LU-факторизацию и определяет составной тип факторизации для его хранения:
       # LU-факторизация:
       Alu = lu(A)

[126]: LU{Float64, Matrix{Float64}, Vector{Int64}}
       L factor:
       3×3 Matrix{Float64}:
        1.0       0.0       0.0
        0.295858  1.0       0.0
        0.960842  0.733276  1.0
       U factor:
       3×3 Matrix{Float64}:
        0.886681  0.401979   0.916675
        0.0       0.82613    0.357058
        0.0       0.0       -0.603898
```

Рис. 9: LU-факторизация

```
[128]: # Различные части факторизации могут быть извлечены путём доступа к их специальным свойствам:
       # Матрица перестановок:
       Alu.P

[128]: 3×3 Matrix{Float64}:
        1.0  0.0  0.0
        0.0  1.0  0.0
        0.0  0.0  1.0

[130]: # Вектор перестановок:
       Alu.p

[130]: 3-element Vector{Int64}:
        1
        2
        3

[132]: # Матрица L:
       Alu.L

[132]: 3×3 Matrix{Float64}:
        1.0       0.0       0.0
        0.295858  1.0       0.0
        0.960842  0.733276  1.0

[134]: # Матрица U:
       Alu.U

[134]: 3×3 Matrix{Float64}:
        0.886681  0.401979   0.916675
        0.0       0.82613    0.357058
        0.0       0.0       -0.603898
```

Рис. 10: Различные части факторизации

Факторизация. Специальные матричные структуры

```
[136]: # 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

[136]: 3-element Vector{Float64}:
        1.0000000000000007
        1.0
        0.9999999999999998

[138]: # Решение СЛАУ через объект факторизации:
       Alu\b

[138]: 3-element Vector{Float64}:
        1.0000000000000007
        1.0
        0.9999999999999998

[142]: # Аналогично можно найти детерминант матрицы:
       # Детерминант матрицы A:
       det(A)

[142]: -0.44236390523762914

[144]: # Детерминант матрицы A через объект факторизации:
       det(Alu)

[144]: -0.44236390523762914
```

Рис. 11: Решение СЛАУ через объект факторизации

Факторизация. Специальные матричные структуры

```
[146]: # Julia позволяет вычислять QR-факторизацию и определяет составной тип факторизации для его хранения:
       # QR-факторизация:
       Aqr = qr(A)

[146]: LinearAlgebra.QRCompactWY{Float64, Matrix{Float64}, Matrix{Float64}}
       Q factor: 3×3 LinearAlgebra.QRCompactWYQ{Float64, Matrix{Float64}, Matrix{Float64}}
       R factor:
       3×3 Matrix{Float64}:
        -1.25732  -1.15285  -1.14256
         0.0      -0.84247  -0.212981
         0.0       0.0      -0.417617

[148]: # По аналогии с LU-факторизацией различные части QR-факторизации могут быть извлечены путём доступа к их специальным свойствам:
       # Матрица Q:
       Aqr.Q

[148]: 3×3 LinearAlgebra.QRCompactWYQ{Float64, Matrix{Float64}, Matrix{Float64}}

[150]: # Матрица R:
       Aqr.R

[150]: 3×3 Matrix{Float64}:
        -1.25732  -1.15285  -1.14256
         0.0      -0.84247  -0.212981
         0.0       0.0      -0.417617

[152]: # Проверка, что матрица Q - ортогональная:
       Aqr.Q'*Aqr.Q

[152]: 3×3 Matrix{Float64}:
         1.0          0.0          0.0
        -1.11022e-16  1.0          0.0
         0.0          1.11022e-16  1.0

[154]: # Примеры собственной декомпозиции матрицы A:
       # Симметризация матрицы A:
       Asym = A + A'

[154]: 3×3 Matrix{Float64}:
        1.77336  0.66431  1.76864
        0.66431  1.89012  1.62028
        1.76864  1.62028  1.07741
```

Рис. 12: QR-факторизация

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

[156]: Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}
       values:
       3-element Vector{Float64}:
        -0.7198759543256772
         1.174848848234654
         4.285914908945202
       vectors:
       3×3 Matrix{Float64}:
        -0.466672   0.676165   0.570103
        -0.377691  -0.735228   0.56284
         0.799729   0.0473388  0.598492

[158]: # Собственные значения:
       AsymEig.values

[158]: 3-element Vector{Float64}:
        -0.7198759543256772
         1.174848848234654
         4.285914908945202

[160]: #Собственные векторы:
       AsymEig.vectors

[160]: 3×3 Matrix{Float64}:
        -0.466672   0.676165   0.570103
        -0.377691  -0.735228   0.56284
         0.799729   0.0473388  0.598492

[162]: # Проверяем, что получится единичная матрица:
       inv(AsymEig)*Asym

[162]: 3×3 Matrix{Float64}:
         1.0          2.88658e-15  -3.66374e-15
         2.10942e-15  1.0          -2.55351e-15
        -2.88658e-15 -2.44249e-15   1.0
```

Рис. 13: QR-факторизация

```
[164]: # Далее рассмотрим примеры работы с матрицами большой размерности и специальной структуры.
       # Матрица 1000 x 1000:
       n = 1000
       A = randn(n,n)
```

```
[164]: 1000×1000 Matrix{Float64}:
  0.623945   -0.496967   -2.11426    …  -3.00805    -0.219126   0.573915
  1.68989    -0.833416   -0.589519       1.45875    -1.34334   -1.08256
 -2.13341     0.416017   -0.197505      -0.191931    0.916257  -0.529161
  0.28138    -0.543692    0.848357      -0.375581   -0.501058   0.157937
  0.725278   -1.11208    -0.373024       0.258856   -0.264299   1.59997
 -0.7064      0.915139    0.481475   …   0.651622   -1.02356    0.721476
 -2.66783    -1.46398    -1.93301       -0.686501    0.916731  -0.914721
  1.85293     0.0744687  -1.13425       -1.47091    -0.862759   0.145169
 -1.29492    -0.200792    0.515706       1.33222     0.39225    0.685819
  0.0210809   0.86459     0.383984       0.0117303   3.15703   -0.270259
 -1.57396     1.16046    -1.56245    …   0.0594681   0.111526   2.54117
 -1.22432    -1.27418    -0.776848      -0.274711    0.96499   -1.77935
 -1.57427    -1.31015     0.181534       0.431541   -1.08759   -0.58189
  ⋮                                  ⋱
 -2.00994     0.511314    1.32301        0.756117   -1.51579    0.721919
 -0.297101    0.0276618  -0.0342905      0.598987    0.05747    0.223246
  0.135391    0.27086     0.309019   …  -0.394535    0.2114     0.0651544
  0.0959663   0.418049    1.6702        -0.955941    1.02128   -0.204544
  1.64632     0.528453   -0.423169       0.0945252  -1.25647    1.27456
  0.187143   -0.939017   -0.0285036     -1.94846    -0.812934   0.606777
  0.480552   -2.08577     0.37736       -1.35603    -1.60357   -0.794517
 -1.15752     0.685581    0.581649   …  -0.288021   -0.220275  -0.519108
 -0.477518   -1.69567    -0.870685       0.625143    2.74323    0.427758
  0.381588    0.477474   -0.11569        1.08246     1.00305   -0.33671
  0.147353   -1.05072    -0.0593053      0.079177    0.258565  -0.0526398
 -0.149447    0.0739406   0.187968       0.755069   -0.543504  -0.940728
```

Рис. 14: матрицы большой размерности и специальной структуры

```
[166]: 1000×1000 Matrix{Float64}:
  1.24789     1.19293   -4.24767    …  -2.62646   -0.0717734   0.424468
  1.19293    -1.66683   -0.173503       1.93622   -2.39406    -1.00862
 -4.24767    -0.173503  -0.39501       -0.307621   0.856952   -0.341193
  0.661497   -0.754481   0.920563       0.049072  -0.724429    1.08763
  1.08465    -0.794996   0.480442      -0.274565  -0.269998    1.03999
 -0.428831    0.629295  -1.70503    …   0.417652  -1.0594     -0.59288
 -2.95268    -1.4381    -1.16623        0.187729   0.240953   -0.185212
  0.906144    0.394353  -1.43494       -1.89401   -2.85063     1.54254
 -2.64244    -0.576733   0.0450604      0.627701   0.534378    1.39839
 -0.163922    1.77654    1.42639       -0.631411   4.36775    -1.70137
 -1.78534     0.92908   -2.51808    …  -1.63213    0.821041    3.55793
 -0.794167   -0.406059  -1.21009       -1.44541    0.321119    0.681602
 -1.78873    -1.84311   -0.338946      -0.908856   0.118506    0.228114
  ⋮                                 ⋱
 -1.57741     1.52802    0.477084       0.384377  -0.61353    -0.744646
 -1.2265      1.05687    0.209228       0.942976   0.353288   -0.865458
  0.320899    0.921174   0.103883   …   1.43052    0.0708732   0.670939
  0.15611     0.812566  -1.38991       -1.90851    1.40172     0.0109671
  0.977908    1.42513    1.52782       -0.274802  -1.55277     2.20034
  0.670046   -1.39204    0.961488      -2.59513   -3.61322     1.7157
  0.81705    -1.40357    0.23469       -1.81623   -0.565023    0.33051
 -1.05968    -1.25758   -0.294913   …  -0.722654   0.437491   -1.37531
  0.339985   -0.587766  -2.20773        2.7329     2.34075    -0.000816334
 -2.62646     1.93622   -0.307621       2.16492    1.08223     0.418359
 -0.0717734  -2.39406    0.856952       1.08223    0.517131   -0.596144
  0.424468   -1.00862   -0.341193       0.418359  -0.596144   -1.88146
```

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

```
[168]: true
```

```
[170]: # Пример добавления шума в симметричную матрицу (матрица уже не будет симметричной):
       # Добавление шума:
       Asym_noisy = copy(Asym)
       Asym_noisy[1,2] += 5eps()
```

```
[170]: 1.192925531359975
```

Рис. 15: матрицы большой размерности и специальной структуры

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

[172]: false

[174]: # 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)

[174]: 1000×1000 Symmetric{Float64, Matrix{Float64}}:
         1.24789     1.19293   -4.24767    …  -2.62646   -0.0717734   0.424468
         1.19293    -1.66683   -0.173503       1.93622   -2.39406    -1.00862
        -4.24767    -0.173503  -0.39501       -0.307621   0.856952   -0.341193
         0.661497   -0.754481   0.920563       0.049072  -0.724429    1.08763
         1.08465    -0.794996   0.480442      -0.274565  -0.269998    1.03999
        -0.428831    0.629295  -1.70503    …   0.417652  -1.0594     -0.59288
        -2.95268    -1.4381    -1.16623        0.187729   0.240953   -0.185212
         0.906144    0.394353  -1.43494       -1.89401   -2.85063     1.54254
        -2.64244    -0.576733   0.0450604      0.627701   0.534378    1.39839
        -0.163922    1.77654    1.42639       -0.631411   4.36775    -1.70137
        -1.78534     0.92908   -2.51808    …  -1.63213    0.821041    3.55793
        -0.794167   -0.406059  -1.21009       -1.44541    0.321119    0.681602
        -1.78873    -1.84311   -0.338946      -0.908856   0.118506    0.228114
         ⋮                                 ⋱
        -1.57741     1.52802    0.477084       0.384377  -0.61353    -0.744646
        -1.2265      1.05687    0.209228       0.942976   0.353288   -0.865458
         0.320899    0.921174   0.103883   …   1.43052    0.0708732   0.670939
         0.15611     0.812566  -1.38991       -1.90851    1.40172     0.0109671
         0.977908    1.42513    1.52782       -0.274802  -1.55277     2.20034
         0.670046   -1.39204    0.961488      -2.59513   -3.61322     1.7157
         0.81705    -1.40357    0.23469       -1.81623   -0.565023    0.33051
        -1.05968    -1.25758   -0.294913   …  -0.722654   0.437491   -1.37531
         0.339985   -0.587766  -2.20773        2.7329     2.34075    -0.000816334
        -2.62646     1.93622   -0.307621       2.16492    1.08223     0.418359
        -0.0717734  -2.39406    0.856952       1.08223    0.517131   -0.596144
         0.424468   -1.00862   -0.341193       0.418359  -0.596144   -1.88146
```

Рис. 16: матрицы большой размерности и специальной структуры

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

         77.999 ms (21 allocations: 7.99 MiB)

[179]: # Оценка эффективности выполнения операции по нахождению
       # собственных значений зашумлённой матрицы:
       @btime eigvals(Asym_noisy);

         516.202 ms (27 allocations: 7.93 MiB)

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

         65.899 ms (21 allocations: 7.99 MiB)

[186]: # Трёхдиагональная матрица 1000000 x 1000000:
       n = 1000000;
       A = SymTridiagonal(randn(n), randn(n-1))

[186]: 1000000×1000000 SymTridiagonal{Float64, Vector{Float64}}:
        -0.271794  -2.49126    ⋅        …   ⋅         ⋅         ⋅
        -2.49126    1.22846   1.48041        ⋅         ⋅         ⋅
          ⋅         1.48041  -0.368774       ⋅         ⋅         ⋅
          ⋅          ⋅        0.562906       ⋅         ⋅         ⋅
          ⋅          ⋅         ⋅             ⋅         ⋅         ⋅
          ⋅          ⋅         ⋅        …    ⋅         ⋅         ⋅
          ⋅          ⋅         ⋅             ⋅         ⋅         ⋅
          ⋅          ⋅         ⋅             ⋅         ⋅         ⋅
          ⋅          ⋅         ⋅             ⋅         ⋅         ⋅
```

Рис. 17: оценка эффективности выполнения операций над матрицами

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

         469.556 ms (44 allocations: 183.11 MiB)

[188]: 6.642144092346992

[190]: B = Matrix(A)

       OutOfMemoryError()
```

Рис. 18: оценки эффективности выполнения операций над матрицами

Общая линейная алгебра

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

[195]: 3×3 Matrix{Rational{BigInt}}:
        1//2  9//10  7//10
        4//5  3//5   4//5
        1//5  3//10  1//10

[197]: # Единичный вектор:
       x = fill(1, 3)

[197]: 3-element Vector{Int64}:
        1
        1
        1

[199]: # Задаём вектор b:
       b = Arational*x

[199]: 3-element Vector{Rational{BigInt}}:
        21//10
        11//5
         3//5

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

[201]: 3-element Vector{Rational{BigInt}}:
        1
        1
        1

[203]: # LU-разложение:
       lu(Arational)

[203]: LU{Rational{BigInt}, Matrix{Rational{BigInt}}, Vector{Int64}}
       L factor:
       3×3 Matrix{Rational{BigInt}}:
         1     0    0
        5//8   1    0
        1//4  2//7  1
       U factor:
       3×3 Matrix{Rational{BigInt}}:
        4//5   3//5     4//5
         0    21//40    1//5
         0      0     -11//70
```

Рис. 19: Общая линейная алгебра

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

```
[3]: using LinearAlgebra
     v = [1, 2, 3, 4]
     dot_v = dot(v, v)

[3]: 30
```

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

```
[8]: outer_v = v * v'

[8]: 4×4 Matrix{Int64}:
      1  2   3   4
      2  4   6   8
      3  6   9  12
      4  8  12  16
```

Рис. 20: Произведение векторов

```
[29]: # a)
      Aa = [1 1; 1 -1]
      Ba = [2, 3]
      Ca = Aa \ Ba

[29]: 2-element Vector{Float64}:
        2.5
       -0.5

[41]: # b)
      Ab = [1 1; 2 2]
      Bb = [2, 4]
      println(det(Ab))
      println(rank(Ab))
      println(cond(Ab))
      println("Система линейно зависимая, бесконечное множестов решений")

      0.0
      1
      2.0140709820486308e16
      Система линейно зависимая, бесконечное множестов решений

[47]: # c)
      Ac = [1 1; 2 2]
      Bc = [5, 2]
      println(det(Ac))
      println(rank(Ac))
      println(cond(Ac))
      println("Система линейно зависимая, бесконечное множестов решений")

      0.0
      1
      2.0140709820486308e16
      Система линейно зависимая, бесконечное множестов решений
```

Рис. 21: Системы линейных уравнений с 2-мя неизвестными

Самостоятельная работа

```
[59]: # d)
      Ad = [1 1; 2 2; 3 3]
      Bd = [1, 2, 3]
      Cd = Ad \ Bd

[59]: 2-element Vector{Float64}:
       0.4999999999999999
       0.5

[61]: # e)
      Ae = [1 1; 2 1; 1 -1]
      Be = [2, 1, 3]
      Ce = Ae \ Be

[61]: 2-element Vector{Float64}:
        1.5000000000000004
       -0.9999999999999997

[63]: # f)
      Af = [1 1; 2 1; 3 2]
      Bf = [2, 1, 3]
      Cf = Af \ Bf

[63]: 2-element Vector{Float64}:
       -0.9999999999999989
        2.9999999999999982
```

Рис. 22: Системы линейных уравнений с 2-мя неизвестными

```
•[66]: # a)
A2a = [1 1 1; 1 -1 -2]
B2a = [2, 3]
C2a = A2a \ B2
```

```
[66]: 3-element Vector{Float64}:
  2.2142857142857144
  0.35714285714285704
 -0.5714285714285712
```

```
[68]: # b)
A2b = [1 1 1; 2 2 -3; 3 1 1]
B2b = [2, 4, 1]
C2b = A2b \ B2b
```

```
[68]: 3-element Vector{Float64}:
 -0.5
  2.5
  0.0
```

```
[74]: # c)
A2c = [1 1 1; 1 1 2; 2 2 3]
B2c = [1, 0, 1]
#C2c = A2c \ B2c
println(det(A2c))
println(rank(A2c))
println(cond(A2c))
println("Система линейно зависимая, бесконечное множестов решений")
```

```
0.0
2
2.590401531955156e16
Система линейно зависимая, бесконечное множестов решений
```

```
[80]: # d)
A2d = [1 1 1; 1 1 2; 2 2 3]
B2d = [1, 0, 0]
#C2d = A2d \ B2d
println(det(A2d))
println(rank(A2d))
println(cond(A2d))
println("Система линейно зависимая, бесконечное множестов решений")
```

```
0.0
2
2.590401531955156e16
Система линейно зависимая, бесконечное множестов решений
```

Рис. 23: Системы линейных уравнений с 3-мя неизвестными

Самостоятельная работа

1. Приведите приведённые ниже матрицы к диагональному виду

[89]:
```
# a)
Aa = [1 -2; -2 1]
diag(Aa)
```

[89]:
```
2-element Vector{Int64}:
 1
 1
```

[91]:
```
# b)
Ab = [1 -2; -2 3]
diag(Ab)
```

[91]:
```
2-element Vector{Int64}:
 1
 3
```

[95]:
```
# c)
Ac = [1 -2 0; -2 1 2; 0 2 0]
diag(Ac)
```

[95]:
```
3-element Vector{Int64}:
 1
 1
 0
```

Рис. 24: Операции с матрицами

2. Вычислите

```
[98]: # a)
      Aa = [1 -2; -2 1]
      Aa_pow = Aa^10

[98]: 2×2 Matrix{Int64}:
        29525  -29524
       -29524   29525

[108]: # b)
       Ab = [5 -2; -2 5]
       Ab_sqrt = sqrt(Ab)

[108]: 2×2 Matrix{Float64}:
         2.1889   -0.45685
        -0.45685   2.1889

[110]: # c)
       Ac = [1 -2; -2 5]
       Ac_cbrt = Ac^(1/3)

[110]: 2×2 Symmetric{Float64, Matrix{Float64}}:
         0.737843  -0.439807
        -0.439807   1.61746

[112]: # d)
       Ad = [1 2; 2 3]
       Ad_sqrt = sqrt(Ad)

[112]: 2×2 Matrix{ComplexF64}:
        0.568864+0.351578im  0.920442-0.217287im
        0.920442-0.217287im   1.48931+0.134291im
```

Рис. 25: Опреации с матрицами

```
[121]: using BenchmarkTools
       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]

       # собственные значения матрицы
       eigvals_A = eigvals(A)

[121]: 5-element Vector{Float64}:
        -128.49322764802145
         -55.88778455305688
          42.75216727931894
          87.16111477514521
         542.4677301466143

[125]: # диагональная матрица из собственных значений
       D = Diagonal(eigvals_A)

[125]: 5×5 Diagonal{Float64, Vector{Float64}}:
        -128.493     ⋅         ⋅         ⋅         ⋅
           ⋅      -55.8878     ⋅         ⋅         ⋅
           ⋅         ⋅       42.7522     ⋅         ⋅
           ⋅         ⋅         ⋅       87.1611     ⋅
           ⋅         ⋅         ⋅         ⋅       542.468

[129]: # нижнедиагональная матрица
       B = LowerTriangular(A)

[129]: 5×5 LowerTriangular{Int64, Matrix{Int64}}:
        140    ⋅    ⋅    ⋅   ⋅
         97  106    ⋅    ⋅   ⋅
         74   89  152    ⋅   ⋅
        168  131  144   54   ⋅
        131   36   71  142  36
```

Рис. 26: Операции с матрицами

```
[131]: @btime eigvals(A)

  1.470 μs (15 allocations: 2.55 KiB)

[131]: 5-element Vector{Float64}:
        -128.49322764802145
         -55.88778455305688
          42.75216727931894
          87.16111477514521
         542.4677301466143

[133]: @btime LowerTriangular(A)

  159.566 ns (1 allocation: 16 bytes)

[133]: 5×5 LowerTriangular{Int64, Matrix{Int64}}:
        140    ⋅    ⋅    ⋅   ⋅
         97  106    ⋅    ⋅   ⋅
         74   89  152    ⋅   ⋅
        168  131  144   54   ⋅
        131   36   71  142  36
```

Рис. 27: Опреации с матрицами

1. Матрица A называется продуктивной, если решение x системы при любой неотрицательной правой части y имеет только неотрицательные элементы xi . Используя это определение, проверьте, являются ли матрицы продуктивными.

```
A1 = [1 2; 3 4]
A2 = (1/2) * A1
A3 = (1/10) * A1
A4 = [0.1 0.2 0.3; 0 0.1 0.2; 0 0.1 0.3]

function check_productive(A, test_vectors=10)
    n = size(A, 1)
    E = Matrix{Float64}(I, n, n)

    for i in 1:test_vectors
        y = rand(n)
        try
            x = (E - A) \ y
            if any(x .< 0)
                return false
            end
        catch
            return false
        end
    end
    return true
end

println("Проверка продуктивности")
println("a) ", check_productive(A1) ? "Продуктиная" : "Непродуктивная")
println("b) ", check_productive(A2) ? "Продуктиная" : "Непродуктивная")
println("c) ", check_productive(A3) ? "Продуктиная" : "Непродуктивная")
```

```
Проверка продуктивности
a) Непродуктивная
b) Непродуктивная
c) Продуктиная
```

Рис. 28: Операции с матрицами

[141]:
```julia
# проверка продуктивности по критерию (E - A)^(-1)
function check_productive_criterion(A)
    n = size(A, 1)
    E = Matrix{Float64}(I, n, n)
    try
        inv_matrix = inv(E - A)
        return all(inv_matrix .>= 0)
    catch
        return false
    end
end

println("Проверка продуктивности по критерию (E - A)^(-1)")

println("a) ", check_productive_criterion(A1) ? "Продуктина" : "Не продуктивна")
println("b) ", check_productive_criterion(A2) ? "Продуктина" : "Не продуктивна")
println("c) ", check_productive_criterion(A3) ? "Продуктина" : "Не продуктивна")
```

```
Проверка продуктивности по критерию (E - A)^(-1)
a) Не продуктивна
b) Не продуктивна
c) Продуктина
```

[149]:
```julia
# проверка продуктивности по спектральному критерию
function check_productive_spectral(A)
    eig_vals = eigvals(A)
    return all(abs.(eig_vals) .< 1)
end
println("Проверка продуктивности по спектральному критерию")
println("a) ", check_productive_spectral(A1) ? "Продуктина" : "Не продуктивна")
println("b) ", check_productive_spectral(A2) ? "Продуктина" : "Не продуктивна")
println("c) ", check_productive_spectral(A3) ? "Продуктина" : "Не продуктивна")
println("d) ", check_productive_spectral(A3) ? "Продуктина" : "Не продуктивна")
```

```
Проверка продуктивности по спектральному критерию
a) Не продуктивна
b) Не продуктивна
c) Продуктина
d) Продуктина
```

Рис. 29: Опреации с матрицами

Выводы

Выводы

В результате выполнения лабораторной работы были изучены возможностей специализированных паке- тов Julia для выполнения и оценки эффективности операций над объектами линейной алгебры.