# Лабораторная работа №4

Компьютерный практикум по статистическому анализу данных

Коннова Татьяна

2025

Российский университет дружбы народов имени Патриса Лумумбы, Москва, Россия

# Докладчик

- Коннова Татьяна Алексеевна
- Студентка группы НПИбд-01-22
- Студ. билет 1132221814
- Российский университет дружбы народов имени Патриса Лумумбы

# Цель лабораторной работы

- Изучить возможности специализированных пакетов Julia для выполнения и оценки эффективности операций над объектами линейной алгебры.

# Выполнение лабораторной работы

Для матрицы 4 × 3 рассмотрим поэлементные операции сложения и произведения её элементов:

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

[1]:
```
4×3 Matrix{Int64}:
 17   6  18
 18  16  14
  5  19  18
  6   1   5
```

[2]:
```julia
# Поэлементная сумма:
sum(a)
```

[2]:
```
143
```

[3]:
```julia
# Поэлементная сумма по столбцам:
sum(a,dims=1)
```

[3]:
```
1×3 Matrix{Int64}:
 46  42  55
```

[4]:
```julia
# Поэлементная сумма по строкам:
sum(a,dims=2)
```

[4]:
```
4×1 Matrix{Int64}:
 41
 48
 42
 12
```

[6]:
```julia
# Поэлементное произведение:
prod(a)
```

[6]:
```
379761177600
```

[7]:
```julia
# Поэлементное произведение по столбцам:
prod(a,dims=1)
```

[7]:
```
1×3 Matrix{Int64}:
 9180  1824  22680
```

[8]:
```julia
# Поэлементное произведение по строкам:
prod(a,dims=2)
```

[8]:
```
4×1 Matrix{Int64}:
 1836
 4032
 1710
   30
```

**Рис. 1:** Поэлементные операции сложения и произведения элементов матрицы

Для работы со средними значениями можно воспользоваться возможностями пакета Statistics:

[10]:
```julia
# Подключение пакета Statistics:
using Statistics

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

[10]:
```
11.916666666666666
```

[11]:
```julia
# Среднее по столбцам:
mean(a,dims=1)
```

[11]:
```
1×3 Matrix{Float64}:
 11.5  10.5  13.75
```

[12]:
```julia
# Среднее по строкам:
mean(a,dims=2)
```

[12]:
```
4×1 Matrix{Float64}:
 13.666666666666666
 16.0
 14.0
  4.0
```

**Рис. 2:** Использование возможностей пакета Statistics для работы со средними значениями

# Транспонирование, след, ранг, определитель и инверсия матрицы

## 2. Транспонирование, след, ранг, определитель и инверсия матрицы

```
[14]: # Подключение пакета LinearAlgebra:
      using LinearAlgebra

      # Массив 4х4 со случайными целыми числами (от 1 до 20):
      b = rand(1:20,(4,4))
```

```
[14]: 4×4 Matrix{Int64}:
        2  20  16   4
        4  11  10   3
       10   3  14   6
       19   8   5  14
```

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

```
[15]: 4×4 transpose(::Matrix{Int64}) with eltype Int64:
        2   4  10  19
       20  11   3   8
       16  10  14   5
        4   3   6  14
```

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

```
[16]: 41
```

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

```
[17]: 4-element Vector{Int64}:
        2
       11
       14
       14
```

Рис. 3: Использование библиотеки LinearAlgebra для выполнения определённых операций

# Транспонирование, след, ранг, определитель и инверсия матрицы

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

[18]: 4

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

[19]: 4×4 Matrix{Float64}:
       -0.342422    0.650064   -0.068699   -0.0120223
       -0.052383    0.203521   -0.0888793   0.00944611
        0.0517389  -0.0944611   0.0918849  -0.0339201
        0.47617    -0.964792    0.111207    0.0944611

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

[20]: -4657.999999999995

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

[21]: 3×4 Matrix{Float64}:
        0.00319438   0.0652316  -0.0564106   0.00893
       -0.0643184    0.0614736   0.0222671  -0.0207411
        0.0684769   -0.0828166   0.0473269   0.0149927
```

**Рис. 4:** Использование библиотеки LinearAlgebra для выполнения определённых операций

### 3. Вычисление нормы векторов и матриц, повороты, вращения

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

[22]: 3-element Vector{Int64}:
        2
        4
       -5

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

[23]: 6.708203932499369

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

[24]: 11.0

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

[25]: 9.486832980505138

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

[26]: 9.486832980505138

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

[27]: 2.4404307889469252
```

**Рис. 5:** Использование LinearAlgebra.norm(x)

# Вычисление нормы векторов и матриц, повороты, вращения

**Вычисление нормы для двумерной матрицы:**

[28]:
```
# Создание матрицы:
d = [5 -4 2 ; -1 2 3; -2 1 0]
```

[28]:
```
3×3 Matrix{Int64}:
  5  -4  2
 -1   2  3
 -2   1  0
```

[29]:
```
# Вычисление Евклидовой нормы:
opnorm(d)
```

[29]:
```
7.147682841795258
```

[30]:
```
# Вычисление p-нормы:
p=1
opnorm(d,p)
```

[30]:
```
8.0
```

[31]:
```
# Поворот на 180 градусов:
rot180(d)
```

[31]:
```
3×3 Matrix{Int64}:
 0   1  -2
 3   2  -1
 2  -4   5
```

[32]:
```
# Переворачивание строк:
reverse(d,dims=1)
```

[32]:
```
3×3 Matrix{Int64}:
 -2   1  0
 -1   2  3
  5  -4  2
```

[33]:
```
# Переворачивание столбцов
reverse(d,dims=2)
```

[33]:
```
3×3 Matrix{Int64}:
 2  -4   5
 3   2  -1
 0   1  -2
```

**Рис. 6:** Вычисление нормы для двумерной матрицы

### 4. Матричное умножение, единичная матрица, скалярное произведение

```
[34]: # Матрица 2х3 со случайными целыми значениями от 1 до 10:
      A = rand(1:10,(2,3))

[34]: 2x3 Matrix{Int64}:
       2  1   1
       8  7  10

[35]: # Матрица 3х4 со случайными целыми значениями от 1 до 10:
      B = rand(1:10,(3,4))

[35]: 3x4 Matrix{Int64}:
       3  4  10  8
       2  6   3  9
       1  1   2  2

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

[36]: 2x4 Matrix{Int64}:
        9  15   25   27
       48  84  121  147

[37]: # Единичная матрица 3х3:
      Matrix{Int}(I, 3, 3)

[37]: 3x3 Matrix{Int64}:
       1  0  0
       0  1  0
       0  0  1

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

[38]: -17

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

[39]: -17
```

**Рис. 7:** Примеры матричного умножения, единичной матрицы и скалярного произведения

## 5. Факторизация. Специальные матричные структуры

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

[40]: 3x3 Matrix{Float64}:
       0.882431  0.943103   0.605134
       0.640795  0.0451344  0.635614
       0.690356  0.246666   0.592887

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

[41]: 3-element Vector{Float64}:
       1.0
       1.0
       1.0

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

[42]: 3-element Vector{Float64}:
       2.430669297689036
       1.3215433484125134
       1.5299083315262747

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

[43]: 3-element Vector{Float64}:
       1.0000000000000069
       0.9999999999999984
       0.9999999999999929
```

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

# Факторизация. Специальные матричные структуры

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

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

[44]: LU{Float64, Matrix{Float64}, Vector{Int64}}
      L factor:
      3×3 Matrix{Float64}:
       1.0       0.0       0.0
       0.726169  1.0       0.0
       0.782334  0.767769  1.0
      U factor:
      3×3 Matrix{Float64}:
       0.882431   0.943103   0.605134
       0.0       -0.639719   0.196184
       0.0        0.0       -0.0311543

[45]: # Матрица перестановок:
      Alu.P

[45]: 3×3 Matrix{Float64}:
       1.0  0.0  0.0
       0.0  1.0  0.0
       0.0  0.0  1.0

[46]: # Вектор перестановок:
      Alu.p

[46]: 3-element Vector{Int64}:
       1
       2
       3

[47]: # Матрица L:
      Alu.L

[47]: 3×3 Matrix{Float64}:
       1.0       0.0       0.0
       0.726169  1.0       0.0
       0.782334  0.767769  1.0

[48]: # Матрица U:
      Alu.U

[48]: 3×3 Matrix{Float64}:
       0.882431   0.943103   0.605134
       0.0       -0.639719   0.196184
       0.0        0.0       -0.0311543
```

**Рис. 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:

```
[49]: # Решение СЛАУ через матрицу A:
      A\b

[49]: 3-element Vector{Float64}:
       1.0000000000000069
       0.9999999999999984
       0.9999999999999929

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

[50]: 3-element Vector{Float64}:
       1.0000000000000069
       0.9999999999999984
       0.9999999999999929

[51]: # Детерминант матрицы A:
      det(A)

[51]: 0.017586842847094095

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

[52]: 0.017586842847094095
```

**Рис. 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:

[53]:
```
# QR-факторизация:
Aqr = qr(A)
```

[53]:
```
LinearAlgebra.QRCompactWY{Float64, Matrix{Float64}, Matrix{Float64}}
Q factor: 3×3 LinearAlgebra.QRCompactWYQ{Float64, Matrix{Float64}, Matrix{Float64}}
R factor:
3×3 Matrix{Float64}:
 -1.2907  -0.79913   -1.04641
  0.0      0.560104  -0.161714
  0.0      0.0       -0.0243274
```

[54]:
```
# Матрица Q:
Aqr.Q
```

[54]:
```
3×3 LinearAlgebra.QRCompactWYQ{Float64, Matrix{Float64}, Matrix{Float64}}
```

[55]:
```
# Матрица R:
Aqr.R
```

[55]:
```
3×3 Matrix{Float64}:
 -1.2907  -0.79913   -1.04641
  0.0      0.560104  -0.161714
  0.0      0.0       -0.0243274
```

[56]:
```
# Проверка, что матрица Q - ортогональная:
Aqr.Q'*Aqr.Q
```

[56]:
```
3×3 Matrix{Float64}:
 1.0          0.0          -5.55112e-17
 1.66533e-16  1.0           0.0
 0.0          2.22045e-16   1.0
```

**Рис. 11:** Пример вычисления QR-факторизации и определение составного типа факторизации для его хранения

Примеры собственной декомпозиции матрицы $A$:

[57]:
```julia
# Симметризация матрицы A:
Asym = A + A'
```

[57]:
```
3×3 Matrix{Float64}:
 1.76486  1.5839     1.29549
 1.5839   0.0902688  0.88228
 1.29549  0.88228    1.18577
```

[58]:
```julia
# Спектральное разложение симметризованной матрицы:
AsymEig = eigen(Asym)
```

[58]:
```
Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}
values:
3-element Vector{Float64}:
 -0.8695583384409882
  0.21485179692981338
  3.6956119617767946
vectors:
3×3 Matrix{Float64}:
  0.491179   -0.488132  -0.721436
 -0.868753   -0.214306  -0.446476
  0.0633308   0.84605   -0.529329
```

[59]:
```julia
# Собственные значения:
AsymEig.values
```

[59]:
```
3-element Vector{Float64}:
 -0.8695583384409882
  0.21485179692981338
  3.6956119617767946
```

[60]:
```julia
#Собственные векторы:
AsymEig.vectors
```

[60]:
```
3×3 Matrix{Float64}:
  0.491179   -0.488132  -0.721436
 -0.868753   -0.214306  -0.446476
  0.0633308   0.84605   -0.529329
```

[61]:
```julia
# Проверяем, что получится единичная матрица:
inv(AsymEig)*Asym
```

[61]:
```
3×3 Matrix{Float64}:
  1.0          -1.9984e-15   -3.10862e-15
 -9.99201e-16   1.0          -1.55431e-15
  4.44089e-15   3.10862e-15   1.0
```

**Рис. 12:** Примеры собственной декомпозиции матрицы A

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

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

```
[62]: 1000×1000 Matrix{Float64}:
  0.189086    -1.53028     0.486299   …   0.758935   -1.11152    -1.37552
 -0.61427      0.649888   -0.172458       1.68446     0.8061     -1.04439
  0.845243     1.60924    -0.337591      -0.323535   -0.338181   -0.0884662
  0.252898     0.162762    1.86293        0.50689    -1.07474     1.01221
 -0.4357      -0.433722    0.0688926      0.32438     1.3684      0.882943
 -1.27788      3.0514     -1.37049    …  -1.91543     0.137764    0.659376
  0.617412     0.858475    1.93548        0.731247    0.985046    1.90142
  0.809633     0.0857344  -1.23836       -0.119184    0.276476   -0.605794
 -2.35817     -1.53724     0.324926       0.0551818  -1.11042    -0.51897
  0.169373    -1.37595     0.121443      -0.897302    0.111827    1.53789
 -0.268557    -1.9483      0.703303   …   0.844949   -0.105929   -0.973703
 -3.14989     -0.4618      1.08879       -1.73328    -0.0458121   0.732366
  1.05188      1.43149     1.16767       -0.430053   -1.23154    -1.30803
  ⋮                                   ⋱
  1.85675     -0.692866    1.04126       -0.296344   -1.00001     0.214219
 -1.97448      1.2969      0.822812       1.17699    -1.12317     0.702147
 -1.12883     -0.170724   -1.36066    …   0.855854    0.410456    1.17483
 -0.0146883    0.366683    0.253385       0.805336    1.68599    -0.0144126
 -0.481541    -1.37168    -1.85985        1.36922    -0.769374    0.431689
 -0.00427183  -1.00815     0.547827       0.548881   -0.158676   -1.459
  0.00476204  -1.34654    -0.583644      -1.21564    -1.00342    -1.09884
 -0.958769     1.04103     0.743432   …  -1.23923     0.52895     0.036475
 -0.378971     0.674633   -0.542285      -1.10648    -0.242543    0.36251
 -0.234844     0.382717   -0.0809377     -0.909557   -0.434977    0.258415
  0.20623     -0.773274    0.971406       2.22529     0.714398   -0.825674
  0.145106     1.97052     1.15124       -0.568779   -1.11365     1.33837
```

```
[63]: # Симметризация матрицы:
      Asym = A + A'
```

```
[63]: 1000×1000 Matrix{Float64}:
  0.378171   -2.14455     1.33154     …   0.524091   -0.905286   -1.23041
 -2.14455     1.29978     1.43678         2.06718     0.0328261   0.92613
  1.33154     1.43678    -0.675183       -0.404472    0.633225    1.06277
  1.03899    -0.316458    1.37217        -0.931503   -1.68343    -0.222548
 -0.797724    0.288688   -0.00214946      0.609433    2.01135     0.372868
 -2.44661     1.72085     0.105528    …  -2.37402     0.247145    1.3086
 -0.608267   -0.0960106   2.9525         -0.70481     0.245804    2.73805
  2.1318     -1.00553    -2.30285         0.978318   -0.347693   -1.17926
 -1.81651    -2.6546     -1.11806         0.585098   -0.469279    0.184027
  3.31587    -0.652932   -0.179783       -0.167379   -0.513926    0.147241
  0.651795   -2.31253     2.20225     …  -0.661855   -2.516       0.863578
 -3.80064    -0.106805    0.760797       -1.48799     0.210751    1.2835
  1.1911      1.73589     1.11657         0.779074   -0.269166   -1.24056
  ⋮                                   ⋱
 -0.432956   -0.99338     1.22839         1.01865    -1.89691    -0.172508
 -2.70779    -0.13743     1.24084         0.637531   -1.12259     1.66219
 -2.89039     0.447319   -0.982212    …  -0.537538    0.140869    1.45391
 -0.406158    1.4372     -0.212816        2.51235     1.62976     0.579377
 -0.820229   -0.0172196  -1.17387         2.54116    -1.03185     0.969346
  0.364596    1.69348     0.268008        0.661939   -1.9942     -2.66392
  1.19641    -0.911753    0.176437       -0.533748   -2.92098    -0.707762
  0.217823   -1.22359     0.666128    …   0.0614431  -0.618577   -1.63402
  0.0385428  -0.150121   -0.37511        -0.975737    1.05767     0.0268845
  0.524091    2.06718    -0.404472       -1.81911     1.79032    -0.310364
 -0.905286    0.0328261   0.633225        1.79032     1.4288     -1.93932
 -1.23041     0.92613     1.06277        -0.310364   -1.93932     2.67675
```

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

```
[64]: true
```

Рис. 13: Примеры работы с матрицами большой размерности и специальной структуры

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

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

[66]: -2.1445519133285655

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

[67]: false
```

**Рис. 14:** Пример добавления шума в симметричную матрицу

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

[68]:
```
# Явно указываем, что матрица является симметричной:
Asym_explicit = Symmetric(Asym_noisy)
```

[68]:
```
1000×1000 Symmetric{Float64, Matrix{Float64}}:
  0.378171   -2.14455     1.33154     …   0.524091   -0.905286   -1.23041
 -2.14455     1.29978     1.43678         2.06718     0.0328261   0.92613
  1.33154     1.43678    -0.675183       -0.404472    0.633225    1.06277
  1.03899    -0.316458    1.37217        -0.931503   -1.68343    -0.222548
 -0.797724    0.288688   -0.00214946      0.609433    2.01135     0.372868
 -2.44661     1.72085     0.105528    …  -2.37402     0.247145    1.3086
 -0.608267   -0.0960106   2.9525         -0.70481     0.245804    2.73805
  2.1318     -1.00553    -2.30285         0.978318   -0.347693   -1.17926
 -1.81651    -2.6546     -1.11806         0.585098   -0.469279    0.184027
  3.31587    -0.652932   -0.179783       -0.167379   -0.513926    0.147241
  0.651795   -2.31253     2.20225     …  -0.661855   -2.516       0.863578
 -3.80064    -0.106805    0.760797       -1.48799     0.210751    1.2835
  1.1911      1.73589     1.11657         0.779074   -0.269166   -1.24056
  ⋮                                   ⋱
 -0.432956   -0.99338     1.22839         1.01865    -1.89691    -0.172508
 -2.70779    -0.13743     1.24084         0.637531   -1.12259     1.66219
 -2.89039     0.447319   -0.982212    …  -0.537538    0.140869    1.45391
 -0.406158    1.4372     -0.212816        2.51235     1.62976     0.579377
 -0.820229   -0.0172196  -1.17387         2.54116    -1.03185     0.969346
  0.364596    1.69348     0.268008        0.661939   -1.9942     -2.66392
  1.19641    -0.911753    0.176437       -0.533748   -2.92098    -0.707762
  0.217823   -1.22359     0.666128    …   0.0614431  -0.618577   -1.63402
  0.0385428  -0.150121   -0.37511        -0.975737    1.05767     0.0268845
  0.524091    2.06718    -0.404472       -1.81911     1.79032    -0.310364
 -0.905286    0.0328261   0.633225        1.79032     1.4288     -1.93932
 -1.23041     0.92613     1.06277        -0.310364   -1.93932     2.67675
```

**Рис. 15:** Пример явного объявления структуры матрицы

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

[92]:
```
using BenchmarkTools
# Оценка эффективности выполнения операции по нахождению
# собственных значений симметризованной матрицы:
@btime eigvals(Asym);
```

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

[90]:
```
# Оценка эффективности выполнения операции по нахождению
# собственных значений зашумлённой матрицы:
@btime eigvals(Asym_noisy);
```

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

[89]:
```
# Оценка эффективности выполнения операции по нахождению
# собственных значений зашумлённой матрицы,
# для которой явно указано, что она симметричная:
@btime eigvals(Asym_explicit);
```

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

**Рис. 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:

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

```
[88]: 1000000×1000000 SymTridiagonal{Float64, Vector{Float64}}:
 -0.332691   0.419       ⋅         …   ⋅           ⋅           ⋅
  0.419      0.672462  -0.891012       ⋅           ⋅           ⋅
   ⋅        -0.891012   1.89828        ⋅           ⋅           ⋅
   ⋅          ⋅         1.1953         ⋅           ⋅           ⋅
   ⋅          ⋅          ⋅             ⋅           ⋅           ⋅
   ⋅          ⋅          ⋅         …   ⋅           ⋅           ⋅
   ⋅          ⋅          ⋅             ⋅           ⋅           ⋅
   ⋅          ⋅          ⋅             ⋅           ⋅           ⋅
   ⋅          ⋅          ⋅             ⋅           ⋅           ⋅
   ⋅          ⋅          ⋅             ⋅           ⋅           ⋅
   ⋅          ⋅          ⋅         …   ⋅           ⋅           ⋅
   ⋅          ⋅          ⋅             ⋅           ⋅           ⋅
   ⋅          ⋅          ⋅             ⋅           ⋅           ⋅
  ⋮                                ⋱
   ⋅          ⋅          ⋅             ⋅           ⋅           ⋅
   ⋅          ⋅          ⋅             ⋅           ⋅           ⋅
   ⋅          ⋅          ⋅         …   ⋅           ⋅           ⋅
   ⋅          ⋅          ⋅             ⋅           ⋅           ⋅
   ⋅          ⋅          ⋅             ⋅           ⋅           ⋅
   ⋅          ⋅          ⋅             ⋅           ⋅           ⋅
   ⋅          ⋅          ⋅             ⋅           ⋅           ⋅
   ⋅          ⋅          ⋅         …   ⋅           ⋅           ⋅
   ⋅          ⋅          ⋅            0.892529     ⋅           ⋅
   ⋅          ⋅          ⋅           -1.192      -0.0353966    ⋅
   ⋅          ⋅          ⋅           -0.0353966  -0.971044    0.847344
   ⋅          ⋅          ⋅             ⋅          0.847344    0.729344
```

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

```
  484.820 ms (17 allocations: 183.11 MiB)
[87]: 7.228144545523525
```

**Рис. 17:** Примеры работы с разряженными матрицами большой размерности

# Общая линейная алгебра

### 6. Общая линейная алгебра

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

[81]: 3×3 Matrix{Rational{BigInt}}:
       3//5   7//10  7//10
       7//10  9//10   1
       2//5   7//10  3//10

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

[83]: 3-element Vector{Int64}:
       1
       1
       1

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

[84]: 3-element Vector{Rational{BigInt}}:
         2
       13//5
        7//5

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

[85]: 3-element Vector{Rational{BigInt}}:
       1
       1
       1

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

[86]: LU{Rational{BigInt}, Matrix{Rational{BigInt}}, Vector{Int64}}
      L factor:
      3×3 Matrix{Rational{BigInt}}:
        1      0     0
       4//7    1     0
       6//7  -5//13  1
      U factor:
      3×3 Matrix{Rational{BigInt}}:
       7//10   9//10     1
        0     13//70  -19//70
        0       0     -17//65
```

**Рис. 18:** Решение системы линейных уравнений с рациональными элементами без преобразования в типы элементов с плавающей запятой

## Произведение векторов

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

[93]:
```julia
using LinearAlgebra

# Задаем вектор v
v = [1, 2, 3]

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

[93]:
```
14
```

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

[94]:
```julia
# Матричное (внешнее) произведение
outer_v = v * v'
```

[94]:
```
3x3 Matrix{Int64}:
 1  2  3
 2  4  6
 3  6  9
```

**Рис. 19:** Решение задания “Произведение векторов”

# Самостоятельная работа

Системы линейных уравнений

1) Решить СЛАУ с двумя неизвестными:

```
[95]: using LinearAlgebra

# Система a
A1 = [1 1; 1 -1]
b1 = [2; 3]
x1 = A1 \ b1
println("Система a: x = $x1")

# Система b
A2 = [1 1; 2 2]
b2 = [2; 4]
try
    x2 = A2 \ b2
    println("Система b: x = $x2")
catch e
    println("Система b: Нет решения (линейно зависимая система)")
end

# Система c
A3 = [1 1; 2 2]
b3 = [2; 5]
try
    x3 = A3 \ b3
    println("Система c: x = $x3")
catch e
    println("Система c: Нет решения (несовместная система)")
end

# Система d
A4 = [1 1; 2 2; 3 3]
b4 = [1; 2; 3]
try
    x4 = A4 \ b4
    println("Система d: x = $x4")
catch e
    println("Система d: Нет решения (линейно зависимая система)")
end

# Система e
A5 = [1 1; 2 1; 1 -1]
b5 = [2; 1; 3]
try
    x5 = A5 \ b5
    println("Система e: x = $x5")
catch e
    println("Система e: Нет решения (несовместная система)")
end

# Система f
A6 = [1 1; 2 1; 3 2]
b6 = [2; 1; 3]
try
    x6 = A6 \ b6
    println("Система f: x = $x6")
catch e
    println("Система f: Нет решения (сингулярная матрица или неопределённое решение)")
end
```

```
Система a: x = [2.5, -0.5]
Система b: Нет решения (линейно зависимая система)
Система c: Нет решения (несовместная система)
Система d: x = [0.4999999999999999, 0.5]
Система e: x = [1.5000000000000004, -0.9999999999999997]
Система f: x = [-0.9999999999999989, 2.9999999999999982]
```

**Рис. 20:** Решение задания “Системы линейных уравнений”

# Самостоятельная работа

2) Решить СЛАУ с тремя неизвестными:

[96]:
```
using LinearAlgebra

# Система a
A1 = [1 1 1; 1 -1 -2]
b1 = [2; 3]
try
    x1 = A1 \ b1
    println("Система a: x = $x1")
catch e
    println("Система a: Нет решения (недостаточно уравнений для определения решения)")
end

# Система b
A2 = [1 1 1; 2 2 -3; 3 1 1]
b2 = [2; 4; 1]
try
    x2 = A2 \ b2
    println("Система b: x = $x2")
catch e
    println("Система b: Нет решения")
end

# Система c
A3 = [1 1 1; 1 1 2; 2 2 3]
b3 = [1; 0; 1]
try
    x3 = A3 \ b3
    println("Система c: x = $x3")
catch e
    println("Система c: Нет решения (сингулярная матрица или неопределённое решение)")
end

# Система d
A4 = [1 1 1; 1 1 2; 2 2 3]
b4 = [1; 0; 0]
try
    x4 = A4 \ b4
    println("Система d: x = $x4")
catch e
    println("Система d: Нет решения (сингулярная матрица или неопределённое решение)")
end
```

```
Система a: x = [2.2142857142857144, 0.35714285714285704, -0.5714285714285712]
Система b: x = [-0.5, 2.5, 0.0]
Система c: Нет решения (сингулярная матрица или неопределённое решение)
Система d: Нет решения (сингулярная матрица или неопределённое решение)
```

**Рис. 21:** Решение задания “Системы линейных уравнений”

### Операции с матрицами

1) Приведите приведённые ниже матрицы к диагональному виду:

[100]:
```
# a)

A = [1 -2; -2 1]

eigen_A = eigen(A) # Собственные значения и векторы
diag_matrix = Diagonal(eigen_A.values) # Диагональная матрица
```

[100]:
```
2×2 Diagonal{Float64, Vector{Float64}}:
 -1.0   ⋅
   ⋅   3.0
```

[98]:
```
# b)

B = [1 -2; -2 3]

eigen_B = eigen(B) # Собственные значения и векторы
diag_matrix = Diagonal(eigen_B.values) # Диагональная матрица
```

[98]:
```
2×2 Diagonal{Float64, Vector{Float64}}:
 -0.236068   ⋅
   ⋅        4.23607
```

[99]:
```
# c)

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
```

[99]:
```
3×3 Diagonal{Float64, Vector{Float64}}:
 -2.14134   ⋅         ⋅
   ⋅       0.515138   ⋅
   ⋅        ⋅        3.6262
```

**Рис. 22:** Решение задания “Операции с матрицами”

2) Вычислите:

[109]:
```
# Исходная матрица (a)
A = [1 -2;
     -2 1]

# Собственные значения и векторы
eigen_decomp = eigen(A)
P = eigen_decomp.vectors  # Матрица собственных векторов
D = Diagonal(eigen_decomp.values)  # Диагональная матрица собственных значений

# Возводим диагональную матрицу в 10-ю степень
D_10 = D.^10

# Вычисляем A^10
A_10 = P * D_10 * inv(P)

println("Матрица A^10:")
println(A_10)
```

```
Матрица A^10:
[29525.0 -29524.0; -29524.0 29525.0]
```

**Рис. 23:** Решение задания “Операции с матрицами”

```
[108]: # Исходная матрица (b)
A = [5 -2;
     -2 5]

# Собственные значения и векторы
eigen_decomp = eigen(A)
eigenvalues = eigen_decomp.values
eigenvectors = eigen_decomp.vectors

# Проверяем, что собственные значения неотрицательные
if all(eigenvalues .>= 0)
    # Диагональная матрица с квадратными корнями собственных значений
    sqrt_D = Diagonal(sqrt.(eigenvalues))

    # Квадратный корень матрицы
    sqrt_A = eigenvectors * sqrt_D * inv(eigenvectors)

    println("Исходная матрица A:")
    println(A)

    println("\nКвадратный корень матрицы sqrt(A):")
    println(sqrt_A)

    # Проверка, что sqrt(A)^2 = A
    println("\nПроверка: sqrt(A)^2:")
    println(sqrt_A * sqrt_A)
else
    println("Матрица A имеет отрицательные собственные значения, квадратный корень не определён.")
end
```

```
Исходная матрица A:
[5 -2; -2 5]

Квадратный корень матрицы sqrt(A):
[2.188901059316734 -0.45685025174785676; -0.45685025174785676 2.188901059316734]

Проверка: sqrt(A)^2:
[5.000000000000002 -2.000000000000001; -2.000000000000001 5.000000000000002]
```

**Рис. 24:** Решение задания “Операции с матрицами”

```
[107]: # Исходная матрица (c)
A = [1 -2;
     -2 1]

# Собственные значения и векторы
eigen_decomp = eigen(A)
eigenvalues = eigen_decomp.values
eigenvectors = eigen_decomp.vectors

# Преобразуем собственные значения в комплексные для вычисления кубического корня
complex_eigenvalues = Complex.(eigenvalues)
cube_root_D = Diagonal(complex_eigenvalues .^ (1/3))

# Кубический корень матрицы
cube_root_A = eigenvectors * cube_root_D * inv(eigenvectors)

println("Исходная матрица A:")
println(A)

println("\nКубический корень матрицы 3√(A):")
println(cube_root_A)

# Проверка: (3√(A))^3 = A
println("\nПроверка: (3√(A))^3:")
println(cube_root_A * cube_root_A * cube_root_A)
```

```
Исходная матрица A:
[1 -2; -2 1]

Кубический корень матрицы 3√(A):
ComplexF64[0.971124785153704 + 0.4330127018922193im -0.47112478515370404 + 0.4330127018922193im; -0.47112478515370404 + 0.4330127018922193im 0.971124785153704 + 0.4330127018922193im]

Проверка: (3√(A))^3:
ComplexF64[0.9999999999999991 + 0.0im -1.9999999999999991 + 5.551115123125783e-17im; -1.9999999999999991 + 5.551115123125783e-17im 0.9999999999999991 + 0.0im]
```

**Рис. 25:** 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:

[110]:
```julia
# 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
@btime eigen_decomp = eigen(A)
eigenvalues = eigen_decomp.values
eigenvectors = eigen_decomp.vectors

println("Собственные значения матрицы A:")
println(eigenvalues)

# 2. Создание диагональной матрицы из собственных значений
# Прямое создание переменной и вывод без использования @btime
diag_matrix = Diagonal(eigenvalues)

println("\nДиагональная матрица из собственных значений:")
println(Matrix(diag_matrix))  # Преобразуем в стандартный массив для вывода

# 3. Создание нижнедиагональной матрицы из A
lower_triangular = LowerTriangular(A)

println("\nНижнедиагональная матрица из A:")
println(Matrix(lower_triangular))

# 4. Оценка эффективности
println("\nЭффективность выполнения операций:")
@btime eigen(A)
@btime Diagonal(eigenvalues)
@btime LowerTriangular(A)
```

```
  5.433 μs (11 allocations: 3.00 KiB)
Собственные значения матрицы A:
[-1.0, 3.0]

Диагональная матрица из собственных значений:
[-1.0 0.0; 0.0 3.0]

Нижнедиагональная матрица из A:
[140 0 0 0 0; 97 106 0 0 0; 74 89 152 0 0; 168 131 144 54 0; 131 36 71 142 36]

Эффективность выполнения операций:
  5.383 μs (11 allocations: 3.00 KiB)
  193.239 ns (1 allocation: 16 bytes)
  177.997 ns (1 allocation: 16 bytes)
```

[110]:
```
5×5 LowerTriangular{Int64, Matrix{Int64}}:
 140    ⋅    ⋅    ⋅   ⋅
  97  106    ⋅    ⋅   ⋅
  74   89  152    ⋅   ⋅
 168  131  144   54   ⋅
 131   36   71  142  36
```

**Рис. 26:** Решение задания “Операции с матрицами”

Линейные модели экономики

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

2) Критерий продуктивности: матрица $A$ является продуктивной тогда и только тогда, когда все элементы матрица$(E - A)$^(−1) являются неотрицательными числами. Используя этот критерий, проверьте, являютсяли матрицы продуктивными;

3) Спектральный критерий продуктивности: матрица $A$ является продуктивной тогдаи только тогда, когда все её собственные значения по модулю меньше 1. Используяэтот критерий, проверьте, являются ли матрицы продуктивными;

[112]:

```julia
# Матрицы
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]

# Функция проверки через (E - A)^(-1)
function check_productivity_via_inverse(A)
    E = I(size(A, 1))
    B = E - A
    try
        B_inv = inv(B)
        all(B_inv .>= 0)
    catch
        false
    end
end

# Функция проверки спектрального критерия
function check_productivity_via_spectrum(A)
    eigenvalues = eigvals(A)
    all(abs.(eigenvalues) .< 1)
end

# Проверка продуктивности для всех матриц
matrices = [A1, A2, A3, A4]
for (i, A) in enumerate(matrices)
    println("Matrix A$i:")
    println(A)
    println("Via (E - A)^(-1): ", check_productivity_via_inverse(A))
    println("Via spectrum: ", check_productivity_via_spectrum(A))
    println("-"^30)
end
```

```
Matrix A1:
[1.0 2.0; 3.0 4.0]
Via (E - A)^(-1): false
Via spectrum: false
------------------------------
Matrix A2:
[0.5 1.0; 1.5 2.0]
Via (E - A)^(-1): false
Via spectrum: false
------------------------------
Matrix A3:
[0.1 0.2; 0.30000000000000004 0.4]
Via (E - A)^(-1): true
Via spectrum: true
------------------------------
Matrix A4:
[0.1 0.2 0.3; 0.0 0.1 0.2; 0.0 0.1 0.3]
Via (E - A)^(-1): true
Via spectrum: true
------------------------------
```

**Рис. 27:** Решение задания “Линейные модели экономики”

# Вывод

# Вывод

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

# Список литературы. Библиография

# Список литературы. Библиография

[1] Julia Documentation: https://docs.julialang.org/en/v1/