РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ

Факультет физико-математических и естественных наук

Кафедра прикладной информатики и теории вероятностей

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ № 4

дисциплина: Компьютерный практикум

по математическому моделированию

Студент: Абрамян Артём

Группа: НПИбд-01-20

МОСКВА

2023 г.

Постановка задачи

Изучение возможностей специализированных пакетов Julia для выполнения и оценки эффективности операций над объектами линейной алгебры.

Выполнение работы

1. Используя Jupyter Lab, повторите примеры из раздела 4.2.

1.1 Поэлементные операции над многомерными массивами

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

```
4×1 Matrix{Int64}:
 1309
   40
   20
 2023
```

```
begin
    # Массив 4x3 со случайными целыми числами (от 1 до 20):
    a = rand(1:20,(4,3))
    # Поэлементная сумма:
    sum(a)
    # Поэлементная сумма по столбцам:
    sum(a,dims=1)
    # Поэлементная сумма по строкам:
    sum(a,dims=2)
    # Поэлементное произведение:
    prod(a)
    # Поэлементное произведение по столбцам:
    prod(a,dims=1)
    # Поэлементное произведение по строкам:
    prod(a,dims=2)
end
```

Для работы со средними значениями можно воспользоваться возможностями пакета Statistics:

```
4×1 Matrix{Float64}:
 11.666666666666666
  7.666666666666667
  3.3333333333333335
 13.666666666666666
```

```
begin
    # Вычисление среднего значения массива:
    mean(a)
    # Среднее по столбцам:
    mean(a,dims=1)
    # Среднее по строкам:
    mean(a,dims=2)
end
```

1.2 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:

```
3×4 Matrix{Float64}:
  0.167567    0.00341813  -0.943875     0.109641
 -0.0131115   0.0516634    0.00861057   0.00450098
 -0.0280496  -0.0437052    0.381605    -0.0202218
```

```
begin
    # Массив 4х4 со случайными целыми числами (от 1 до 20):
    b = rand(1:20,(4,4))
    # Транспонирование:
    transpose(b)
    # След матрицы (сумма диагональных элементов):
    tr(b)
    # Извлечение диагональных элементов как массив:
    diag(b)
    # Ранг матрицы:
    rank(b)
    # Инверсия матрицы (определение обратной матрицы):
    inv(b)
    # Определитель матрицы:
    det(b)
    # Псевдобратная функция для прямоугольных матриц:
    pinv(a)

end
```

1.3 Вычисление нормы векторов и матриц, повороты, вращения

Для вычисления нормы используется LinearAlgebra.norm(x). Евклидова норма:

$$\|\vec{X}\|_2 = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2};$$

p-норма:

$$\|\vec{A}\|_p = \left(\sum_{i=1}^{n} |a_i|^p \right)^{1/p}.$$

```
11.0
begin
    # Создание вектора X:
    X = [2, 4, -5]
    # Вычисление евклидовой нормы:
    println(norm(X))
    # Вычисление p-нормы:
    p = 1
    norm(X,p)
end
```

```
6.708203932499369
```

Евклидово расстояние между двумя векторами $\vec{X}$ и $\vec{Y}$ определяется как $||\vec{X} - \vec{Y}||_2$.

```
9.486832980505138
begin
    # Расстояние между двумя векторами X и Y:
    X = [2, 4, -5];
    Y = [1,-1,3];
    norm(X-Y)
end
```

```
9.486832980505138
# Проверка по базовому определению:
sqrt(sum((X-Y).^2))
```

Угол между двумя векторами $\vec{X}$ и $\vec{Y}$ определяется как $\cos^{-1} \dfrac{\vec{X}^T\vec{Y}}{||\vec{X}||_2||\vec{Y}||_2}$.

```
2.4404307889469252
```

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

Вычисление нормы для двумерной матрицы:

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

```
begin
    # Создание матрицы:
    d = [5 -4 2 ; -1 2 3; -2 1 0]
    # Вычисление Евклидовой нормы:
    opnorm(d)
    # Вычисление p-нормы:
    p=1
    opnorm(d,p)
    # Поворот на 180 градусов:
    rot180(d)
    # Переворачивание строк:
    reverse(d,dims=1)
    # Переворачивание столбцов
    reverse(d,dims=2)
end
```

1.4 Матричное умножение, единичная матрица, скалярное произведение

```
2×4 Matrix{Int64}:
  77   93  37   63
 170  195  88  135
```

```
begin
    # Матрица 2х3 со случайными целыми значениями от 1 до 10:
    A = rand(1:10,(2,3))
    # Матрица 3х4 со случайными целыми значениями от 1 до 10:
    B = rand(1:10,(3,4))
    # Произведение матриц A и B:
    A*B

end
```

```
-17
```

```
begin
    # Единичная матрица 3x3:
    Matrix{Int}(I, 3, 3)
    # Скалярное произведение векторов X и Y:
    X = [2, 4, -5]
    Y = [1,-1,3]
    dot(X,Y)
    # тоже скалярное произведение:
    X'Y
end
```

1.5 Факторизация. Специальные матричные структуры

В математике факторизация (или разложение) объекта — его декомпозиция (например, числа, полинома или матрицы) в произведение других объектов или факторов, которые, будучи перемноженными, дают исходный объект.
Матрица может быть факторизована на произведение матриц специального вида
для приложений, в которых эта форма удобна. К специальным видам матриц относят ортогональные, унитарные и треугольные матрицы.
LU-разложение — представление матрицы A в виде произведения двух матриц L и U, L — нижняя треугольная матрица, а U — верхняя треугольная матрица. LU-разложение
существует только в том случае, когда матрица A обратима, а все её ведущие (угловые) главные миноры невырождены.
Обращение матрицы A эквивалентно решению линейной системы $AX = I$, где X
— неизвестная матрица, I — единичная матрица. Решение X этой системы является обратной матрицей $A{-}1$
.
LUP-разложение — представление матрицы A в виде произведения $P\ A = LU$, где матрица L является нижнетреугольной с единицами на главной диагонали, U — верхнетреугольная общего вида матрица,P — матрица перестановок, получаемая из единичной
матрицы путём перестановки строк или столбцов.
QR-разложение матрицы — представление матрицы в виде произведения унитарной (или ортогональной) матрицы Q и верхнетреугольной матрицы R. QR-разложение применяется для нахождения собственных векторов и собственных значений матрицы.
Q является ортогональной матрицей, если $QTQ = I$, где I — единичная матрица.
Спектральное разложение матрицы A — представление её в виде произведения
$A = V\ \Lambda V\ {-}1$, где V — матрица, столбцы которой являются собственными векторами матрицы A, Λ — диагональная матрица с соответствующими собственными значениями
на главной диагонали, V
-1 — матрица, обратная матрице V.
Рассмотрим несколько примеров. Для работы со специальными матричными

структурами потребуется пакет LinearAlgebra.
Решение систем линейный алгебраических уравнений $Ax = b$:

```
▸[1.0, 1.0, 1.0]
```

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

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

```
Alu = LU{Float64, Matrix{Float64}, Vector{Int64}}
      L factor:
      3×3 Matrix{Float64}:
       1.0        0.0       0.0
       0.817651   1.0       0.0
       0.0438418  0.107676  1.0
      U factor:
      3×3 Matrix{Float64}:
       0.403483   0.968555   0.93956
       0.0       -0.29352   -0.419362
       0.0        0.0        0.0645152
```

```
# LU-факторизация:
Alu = lu(A)

```

Различные части факторизации могут быть извлечены путём доступа к их специальным свойствам:

```
3×3 Matrix{Float64}:
 0.403483   0.968555   0.93956
 0.0       -0.29352   -0.419362
 0.0        0.0        0.0645152
```

```
begin
    # Матрица перестановок:
    Alu.P
    # Вектор перестановок:
    Alu.p
    # Матрица L:
    Alu.L
    # Матрица U:
    Alu.U

end
```

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

```
▸[1.0, 1.0, 1.0]
```

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

end
```

```
-0.007640548512007329
```

```julia
begin
    # Аналогично можно найти детерминант матрицы:
    # Детерминант матрицы A:
    det(A)
    # Детерминант матрицы A через объект факторизации:
    det(Alu)
end
```

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

```
Aqr = LinearAlgebra.QRCompactWY{Float64, Matrix{Float64}, Matrix{Float64}}
      Q factor:
      3×3 LinearAlgebra.QRCompactWYQ{Float64, Matrix{Float64}, Matrix{Float64}}:
       -0.632627  -0.767036  -0.106954
       -0.033921  -0.110526   0.993294
       -0.773713   0.632013   0.0439029
      R factor:
      3×3 Matrix{Float64}:
       -0.521489  -1.06507   -0.949709
        0.0        0.228633   0.319526
        0.0        0.0        0.0640826
```

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

По аналогии с LU-факторизацией различные части QR-факторизации могут быть извлечены путём доступа к их специальным свойствам:

```
3×3 Matrix{Float64}:
  1.0           0.0          -1.52656e-16
  1.11022e-16   1.0          -4.44089e-16
 -1.11022e-16  -6.66134e-16   1.0
```

```julia
begin
    # Матрица Q:
    Aqr.Q
    # Матрица R:
    Aqr.R
    # Проверка, что матрица Q – ортогональная:
    Aqr.Q'*Aqr.Q

end
```

Примеры собственной декомпозиции матрицы A:

```
3×3 Matrix{Float64}:
  1.0          -4.44089e-16  -2.22045e-15
 -3.33067e-16   1.0           1.11022e-15
  1.22125e-15   1.11022e-15   1.0
```

```
begin
    # Симметризация матрицы A:
    Asym = A + A'
    # Спектральное разложение симметризованной матрицы:
    AsymEig = eigen(Asym)
    # Собственные значения:
    AsymEig.values
    #Собственные векторы:
    AsymEig.vectors
    # Проверяем, что получится единичная матрица:
    inv(AsymEig)*Asym
end
```

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

```
true
```

```
begin
    # Матрица 1000 х 1000:
    n = 1000
    A = randn(n,n)
    # Симметризация матрицы:
    Asym = A + A'
    # Проверка, является ли матрица симметричной:
    issymmetric(Asym)
end
```

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

```
false
```

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

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

```
Asym_explicit =
1000×1000 Symmetric{Float64, Matrix{Float64}}:
  2.32617    4.53906   -0.515212  -2.70843   …   1.26312    0.297646    1.51699
  4.53906   -1.87123   -0.551281   2.83809      -0.160188  -0.678394    1.74008
 -0.515212  -0.551281  -1.41131   -0.59902       0.358006   1.28896    -0.138374
 -2.70843    2.83809   -0.59902   -0.524195     -1.35017   -1.74089     0.649915
  0.731101  -0.941536   0.697613   0.687452      3.01432   -1.13248     0.754698
 -1.93464    0.526741  -1.91906    1.24152   …   1.41181   -0.377776   -0.711587
 -2.89159   -1.03592    0.837335  -1.55737      -0.486875   0.200195    1.23287
  ⋮                                          ⋱
 -1.90703    1.42224    0.336196   1.25613       1.5972    -1.14181    -0.793861
 -2.32831   -0.258837  -0.994849  -0.414734  …   1.94432   -0.219663   -1.03796
 -0.605486   1.33617    1.75273   -0.56749       1.64505   -1.84485     0.973908
  1.26312   -0.160188   0.358006  -1.35017       5.14015   -0.768924    0.460974
  0.297646  -0.678394   1.28896   -1.74089      -0.768924  -0.424995    0.0573533
  1.51699    1.74008   -0.138374   0.649915      0.460974   0.0573533  -2.09935
```

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

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

```
▸[-89.9099, -88.5021, -87.8672, -87.0282, -86.4944, -85.6546, -85.461, -85.0766, -84.4263,
```

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

```
88.650 ms (11 allocations: 7.99 MiB)
588.080 ms (13 allocations: 7.92 MiB)
88.757 ms (11 allocations: 7.99 MiB)
```

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:

```
1000000×1000000 SymTridiagonal{Float64, Vector{Float64}}:
  1.34099  -1.26144    ⋅          ⋅         …  ⋅          ⋅           ⋅
 -1.26144  -0.644347  -0.0260257  ⋅            ⋅          ⋅           ⋅
   ⋅       -0.0260257  1.11743    0.187301     ⋅          ⋅           ⋅
   ⋅         ⋅         0.187301  -0.449754     ⋅          ⋅           ⋅
   ⋅         ⋅          ⋅         0.258799     ⋅          ⋅           ⋅
   ⋅         ⋅          ⋅          ⋅        …  ⋅          ⋅           ⋅
   ⋅         ⋅          ⋅          ⋅           ⋅          ⋅           ⋅
  ⋮                                         ⋱
   ⋅         ⋅          ⋅          ⋅           ⋅          ⋅           ⋅
   ⋅         ⋅          ⋅          ⋅        …  ⋅          ⋅           ⋅
   ⋅         ⋅          ⋅          ⋅          0.11687     ⋅           ⋅
   ⋅         ⋅          ⋅          ⋅         -0.703375   1.18332      ⋅
   ⋅         ⋅          ⋅          ⋅          1.18332    1.18122    -0.0132932
   ⋅         ⋅          ⋅          ⋅           ⋅        -0.0132932  -0.367884
```

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

end
```

```
6.8317366989236845
```

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

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

При попытке задать подобную матрицу обычным способом и посчитать её собственные
значения, вы скорее всего получите ошибку переполнения памяти:
B = Matrix(A)
OutOfMemoryError()

1.6 Общая линейная алгебра

Обычный способ добавить поддержку числовой линейной алгебры - это обернуть подпрограммы BLAS и LAPACK. Собственно, для матриц с элементами Float32,Float64, Complex {Float32} или Complex {Float64} разработчики Julia использовали такое же решение. Однако Julia также поддерживает общую линейную алгебру, что позволяет, например, работать с матрицами и векторами рациональных чисел. Для задания рационального числа используется двойная косая черта: 1//2 В следующем 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):

```
LU{Rational{BigInt}, Matrix{Rational{BigInt}}, Vector{Int64}}
L factor:
3×3 Matrix{Rational{BigInt}}:
 1//1  0//1   0//1
 1//8  1//1   0//1
 1//4  2//69  1//1
U factor:
3×3 Matrix{Rational{BigInt}}:
 4//5   3//10   1//2
 0//1  69//80   3//80
 0//1   0//1   66//115
```

```
begin
    # Матрица с рациональными элементами:
    Arational = Matrix{Rational{BigInt}}(rand(1:10, 3, 3))/10
    # Единичный вектор:
    x = fill(1, 3)
    # Задаём вектор b:
    b = Arational*x
    # Решение исходного уравнения получаем с помощью функции \
    # (убеждаемся, что x - единичный вектор):
    Arational\b
    # LU-разложение:
    lu(Arational)
end
```

2. Задания для самостоятельной работы

2.1 Произведение векторов

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

```
55
begin
    v = [1,2,3,4,5]
    dot_v = dot(v,v)
end
```

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

```
outer_v = 5×5 Matrix{Int64}:
 1   2   3   4   5
 2   4   6   8  10
 3   6   9  12  15
 4   8  12  16  20
 5  10  15  20  25
outer_v = v * v'
```

2.2 Системы линейных уравнений

Решить СЛАУ с двумя неизвестными.

a) $\begin{cases} x + y = 2, \\ x - y = 3. \end{cases}$

b) $\begin{cases} x + y = 2, \\ 2x + 2y = 4. \end{cases}$

c) $\begin{cases} x + y = 2, \\ 2x + 2y = 5. \end{cases}$

d) $\begin{cases} x + y = 1, \\ 2x + 2y = 2, \\ 3x + 3y = 3. \end{cases}$

e) $\begin{cases} x + y = 2, \\ 2x + y = 1, \\ x - y = 3. \end{cases}$

f) $\begin{cases} x + y = 2, \\ 2x + y = 1, \\ 3x + 2y = 3. \end{cases}$

```
▸[2.5, -0.5]
```

```
begin
    K = [1 1; 1 -1]
    answers = [2; 3]
    K\answers
end
```

```
LinearAlgebra.SingularException(2)

 1. checknonsingular @ factorization.jl:19 [inlined]
 2. checknonsingular @ factorization.jl:21 [inlined]
 3. #lu!#170 @ lu.jl:82 [inlined]
 4. #lu#177 @ lu.jl:279 [inlined]
 5. lu @ lu.jl:278 [inlined]
 6. \(::Matrix{Int64}, ::Vector{Int64}) @ generic.jl:1110
 7. top-level scope @ Local: 4 [inlined]
```

```
begin
    K2 = [1 1; 2 2]
    answers2 = [2; 4]
    K2\answers
end
# решений бесконечно
```

```
LinearAlgebra.SingularException(2)

 1. checknonsingular @ factorization.jl:19 [inlined]
 2. checknonsingular @ factorization.jl:21 [inlined]
 3. #lu!#170 @ lu.jl:82 [inlined]
 4. #lu#177 @ lu.jl:279 [inlined]
 5. lu @ lu.jl:278 [inlined]
 6. \(::Matrix{Int64}, ::Vector{Int64}) @ generic.jl:1110
 7. top-level scope @ Local: 4 [inlined]
```

```
begin
    K3 = [1 1; 2 2]
    answers3 = [2; 5]
    K3\answers3
end
# пустое множество решений
```

```
[0.5, 0.5]
```

```
begin
    K4 = [1 1; 2 2; 3 3]
    answers4 = [1;2;3]
    K4\answers4
end
```

```
[1.5, -1.0]
```

```
begin
    K5 = [1 1; 2 1; 1 -1]
    answers5 = [2;1;3]
    K5\answers5
end
# полученные значения неверны
```

99.2 μs

99.2 μs

```
[-1.0, 3.0]
```

```
begin
    K6 = [1 1; 2 1; 3 2]
    answers6 = [2;1;3]
    K6\answers6
end
```

Решить СЛАУ с тремя неизвестными

a) $\begin{cases} x+y+z=2, \\ x-y-2z=3. \end{cases}$

b) $\begin{cases} x+y+z=2, \\ 2x+2y-3z=4, \\ 3x+y+z=1. \end{cases}$

c) $\begin{cases} x+y+z=1, \\ x+y+2z=0, \\ 2x+2y+3z=1. \end{cases}$

d) $\begin{cases} x+y+z=1, \\ x+y+2z=0, \\ 2x+2y+3z=0. \end{cases}$

```
[2.21429, 0.357143, -0.571429]
```

```
begin
    K7 = [1 1 1; 1 -1 -2]
    answers7 = [2;3]
    K7\answers7
end
# решения неверны, так как необходимо минимум 3 уравнения для решения СУ с 3 неизвестными
```

123 μs

123 μs

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

```
begin
    K8 = [1 1 1; 2 2 -3;3 1 1]
    answers8 = [2;4;1]
    K8\answers8
end
# решение верное
```

44.1 μs

44.1 μs

```
LinearAlgebra.SingularException(2)

1. checknonsingular @ factorization.jl:19 [inlined]
2. checknonsingular @ factorization.jl:21 [inlined]
3. #lu!#170 @ lu.jl:82 [inlined]
4. #lu#177 @ lu.jl:279 [inlined]
5. lu @ lu.jl:278 [inlined]
6. \(::Matrix{Int64}, ::Vector{Int64}) @ generic.jl:1110
7. top-level scope @ Local: 4 [inlined]
```

```
begin
    K9 = [1 1 1;1 1 2; 2 2 3]
    answers9 = [1;0;1]
    K9\answers9
end
# пустое множество решений
```

```
LinearAlgebra.SingularException(2)

  1. checknonsingular @ factorization.jl:19 [inlined]
  2. checknonsingular @ factorization.jl:21 [inlined]
  3. #lu!#170 @ lu.jl:82 [inlined]
  4. #lu#177 @ lu.jl:279 [inlined]
  5. lu @ lu.jl:278 [inlined]
  6. \(::Matrix{Int64}, ::Vector{Int64}) @ generic.jl:1110
  7. top-level scope @ Local: 4 [inlined]
```

```
begin
    K10 = [1 1 1;1 1 2; 2 2 3]
    answers10 = [1;0;0]
    K10\answers10
end
# пустое множество решений
```

2.3 Операции с матрицами

Приведите приведённые ниже матрицы к диагональному виду

a) $\begin{pmatrix} 1 & -2 \\ -2 & 1 \end{pmatrix}$

b) $\begin{pmatrix} 1 & -2 \\ -2 & 3 \end{pmatrix}$

c) $\begin{pmatrix} 1 & -2 & 0 \\ -2 & 1 & 2 \\ 0 & 2 & 0 \end{pmatrix}$

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

```
begin
    matrix1 = [1 -2; -2 1]
    Asym = matrix1 + matrix1'
    AsymEig = eigen(Asym)
    inv(AsymEig.vectors) * matrix1 * AsymEig.vectors
end
```

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

```
begin
        matrix = [1 -2; -2 3]
        Asym = matrix + matrix'
        AsymEig = eigen(Asym)
        inv(AsymEig.vectors) * matrix * AsymEig.vectors
    end
```

```
3×3 Matrix{Float64}:
 -2.14134       3.55271e-15  -1.9984e-15
  3.38618e-15   0.515138      1.11022e-16
 -6.66134e-16  -4.44089e-16   3.6262
```

```
begin
        matrix = [1 -2 0; -2 1 2; 0 2 0]
        Asym = matrix + matrix'
        AsymEig = eigen(Asym)
        inv(AsymEig.vectors) * matrix * AsymEig.vectors
    end
```

Вычислите

a) $\begin{pmatrix} 1 & -2 \\ -2 & 1 \end{pmatrix}^{10}$

b) $\sqrt{\begin{pmatrix} 5 & -2 \\ -2 & 5 \end{pmatrix}}$

c) $\sqrt[3]{\begin{pmatrix} 1 & -2 \\ -2 & 1 \end{pmatrix}}$

d) $\sqrt{\begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix}}$

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

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

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

```
sqrt([5 -2; -2 5])
```

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

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

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

```
sqrt([1 2; 2 3])
```

2.4 Найдите собственные значения матрицы A, если

$$A = \begin{pmatrix} 140 & 97 & 74 & 168 & 131 \\ 97 & 106 & 89 & 131 & 36 \\ 74 & 89 & 152 & 144 & 71 \\ 168 & 131 & 144 & 54 & 142 \\ 131 & 36 & 71 & 142 & 36 \end{pmatrix}.$$

```
[-128.493, -55.8878, 42.7522, 87.1611, 542.468]
```

```
begin
    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]
    eigenvalues = eigvals(A)
end
```

Создайте диагональную матрицу из собственных значений матрицы A. Создайте нижнедиагональную матрицу из матрица A. Оцените эффективность выполняемых операций

```
[-128.493, -55.8878, 42.7522, 87.1611, 542.468]
```

```
@btime eigvals(A)
```

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

2.6 s

```
matrix_new = 5×5 Matrix{Float64}:
              0.0  0.0  0.0  0.0  0.0
              0.0  0.0  0.0  0.0  0.0
              0.0  0.0  0.0  0.0  0.0
              0.0  0.0  0.0  0.0  0.0
              0.0  0.0  0.0  0.0  0.0
```

```
matrix_new = zeros(5,5)
```

```
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
```

```
begin

        @btime for i in 1:1:5
            matrix_new[i, i] = eigenvalues[i]
        end


    matrix_new
end
```

```
229.695 ns (5 allocations: 80 bytes)
```

```
LU{Float64, Matrix{Float64}, Vector{Int64}}
L factor:
5×5 Matrix{Float64}:
 1.0        0.0        0.0        0.0        0.0
 0.779762   1.0        0.0        0.0        0.0
 0.440476  -0.47314    1.0        0.0        0.0
 0.833333   0.183929  -0.556312   1.0        0.0
 0.577381  -0.459012  -0.189658   0.897068   1.0
U factor:
5×5 Matrix{Float64}:
 168.0  131.0      144.0      54.0      142.0
   0.0  -66.1488   -41.2857   99.8929   -74.7262
   0.0    0.0       69.0375  167.478    -26.9035
   0.0    0.0        0.0     197.797     11.4442
   0.0    0.0        0.0       0.0      -95.657
```

```
begin
    # Создайте нижнедиагональную матрицу из матрица A.

    Alu = lu(A)
    @btime lu(A)

end
```

```
362.679 ns (3 allocations: 384 bytes)
```

2.5 Линейные модели экономики

Линейная модель экономики может быть записана как СЛАУ
$x - Ax = y$,
где элементы матрицы A и столбца y — неотрицательные числа. По своему смыслу в экономике элементы матрицы A и столбцов x, y не могут быть отрицательными числами.
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.

a) $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$

b) $\frac{1}{2}\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$

c) $\frac{1}{10}\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$

```
is_productive_matrix (generic function with 1 method)
function is_productive_matrix(matrix, size)
        ans=""
         #единичная матрица
        E = [1 0; 0 1]
        # зададим любые неотрицательные числа
        Y = rand(0:1000, size)
        # По формуле вычислим x - A*x = y
        S = E - matrix
        # найдем значения x
        X = S\Y
        # теперь проверим есть ли среди x отрциательное число
        for i in 1:1:size
            if X[i] < 0
                ans = "Матрица непродуктивная"
                break
            else
                ans = "Матрица продуктивная"
            end
        end
        return ans
    end
```

```
begin
    matrix1 = [1 2; 3 4]
    matrix2 = ([1 2; 3 4]) * (1/2)
    matrix3 = ([1 2; 3 4]) * (1/10)
    println(is_productive_matrix(matrix1, 2))
    println(is_productive_matrix(matrix2, 2))
    println(is_productive_matrix(matrix3, 2))
end
```

```
Матрица непродуктивная
Матрица непродуктивная
Матрица продуктивная
```

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

a) $\begin{pmatrix} 1 & 2 \\ 3 & 1 \end{pmatrix}$

b) $\frac{1}{2}\begin{pmatrix} 1 & 2 \\ 3 & 1 \end{pmatrix}$

c) $\frac{1}{10}\begin{pmatrix} 1 & 2 \\ 3 & 1 \end{pmatrix}$

```
is_productive_matrix_2 (generic function with 1 method)
```

```julia
function is_productive_matrix_2(matrix, size)
        # единичная матрица
        ans = ""
        E = [1 0; 0 1]
        matrix_new = E - matrix
        inv_matrix_new = inv(matrix_new)
        for i in 1:1:size
            for j in 1:1:size
                if inv_matrix_new[i, j] < 0
                    ans = "Матрица непродуктивная"
                    break
                else
                    ans = "Матрица продуктивная"
                end
            end
        end
        return ans
    end
```

```julia
begin
    matrix1 = [1 2; 3 1]
    matrix2 = ([1 2; 3 1]) * (1/2)
    matrix3 = ([1 2; 3 1]) * (1/10)

    println(is_productive_matrix_2(matrix1,2))
    println(is_productive_matrix_2(matrix2,2))
    println(is_productive_matrix_2(matrix3,2))
end
```

```
Матрица непродуктивная
Матрица непродуктивная
Матрица продуктивная
```

Спектральный критерий продуктивности: матрица A является продуктивной тогда и только тогда, когда все её собственные значения по модулю меньше 1. Используя этот критерий, проверьте, являются ли матрицы продуктивными.

a) $\begin{pmatrix} 1 & 2 \\ 3 & 1 \end{pmatrix}$

b) $\frac{1}{2}\begin{pmatrix} 1 & 2 \\ 3 & 1 \end{pmatrix}$

c) $\frac{1}{10}\begin{pmatrix} 1 & 2 \\ 3 & 1 \end{pmatrix}$

d) $\begin{pmatrix} 0.1 & 0.2 & 0.3 \\ 0 & 0.1 & 0.2 \\ 0 & 0.1 & 0.3 \end{pmatrix}$

```
is_productive_matrix_3 (generic function with 1 method)
function is_productive_matrix_3(matrix, size)
        ans=""
        # найдем собственные значения переданной матрицы
        eigenvalues = eigvals(matrix)
        for i in 1:1:size
            if abs(eigenvalues[i]) > 1
                ans = "Матрица непродуктивная"
                break
            else
                ans = "Матрица продуктивная"
            end
        end
        return ans
    end
```

```
begin
    matrix1 = [1 2; 3 1]
    matrix2 = ([1 2; 3 1])*(1/2)
    matrix3 = ([1 2; 3 1])*(1/10)
    matrix4 = [0.1 0.2 0.3; 0 0.1 0.2; 0 0.1 0.3]
    println(is_productive_matrix_3(matrix1,2))
    println(is_productive_matrix_3(matrix2,2))
    println(is_productive_matrix_3(matrix3,2))
    println(is_productive_matrix_3(matrix4,2))

end
```

```
Матрица непродуктивная
Матрица непродуктивная
Матрица продуктивная
Матрица продуктивная
```

Выводы

В ходе выполнения лабораторной работы успешно удалось изучить возможности специализированных пакетов Julia для выполнения и оценки эффективности операций над объектами линейной алгебры.