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 статистическому данных анализу

Студент: Доре Стевенсон Эдгар
Группа: НКН-бд-01-19

МОСКВА
2023 г.

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

1. Используя Jupyter Lab, повторите примеры
2. Выполните задания для самостоятельной работы.

Выполнение работы

1. Повторение примером

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

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

Out[3]: 4×3 Array{Int64,2}:
          7  20  11
          2  20  18
         18  11  12
          9  12   8

In [4]: # Поэлементная сумма:
        sum(a)

Out[4]: 148

In [5]: # Поэлементная сумма по столбцам:
        sum(a,dims=1)

Out[5]: 1×3 Array{Int64,2}:
         36  63  49

In [6]: # Поэлементная сумма по строкам:
        sum(a,dims=2)

Out[6]: 4×1 Array{Int64,2}:
         38
         40
         41
         29

In [7]: # Поэлементное произведение:
        prod(a)

Out[7]: 2276215603200

In [8]: # Поэлементное произведение по столбцам:
        prod(a,dims=1)

Out[8]: 1×3 Array{Int64,2}:
         2268  52800  19008

In [9]: # Поэлементное произведение по строкам:
        prod(a,dims=2)

Out[9]: 4×1 Array{Int64,2}:
         1540
          720
         2376
          864
```

Для работы со средними значениями можно воспользоваться возможностями пакета Statistics:

In [11]:
```
# Подключение пакета Statistics:
using Statistics
```

In [12]:
```
# Вычисление среднего значения массива:
mean(a)
```

Out[12]:
```
12.333333333333334
```

In [13]:
```
# Среднее по столбцам:
mean(a,dims=1)
```

Out[13]:
```
1×3 Array{Float64,2}:
 9.0  15.75  12.25
```

In [14]:
```
# Среднее по строкам:
mean(a,dims=2)
```

Out[14]:
```
4×1 Array{Float64,2}:
 12.666666666666666
 13.333333333333334
 13.666666666666666
  9.666666666666666
```

4.2.2. Транспонирование, след, ранг, определитель и инверсия матрицы

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

```
9.666666666666666
```

```
In [16]: # Подключение пакета LinearAlgebra:
         using LinearAlgebra

In [17]: # Массив 4x4 со случайными целыми числами (от 1 до 20):
         b = rand(1:20,(4,4))

Out[17]: 4x4 Array{Int64,2}:
           3  15   2   8
          15  20  14   9
          10  17   9  14
           3   8   5  20

In [18]: # Транспонирование:
         transpose(b)

Out[18]: 4x4 Transpose{Int64,Array{Int64,2}}:
           3  15  10   3
          15  20  17   8
           2  14   9   5
           8   9  14  20

In [19]: # След матрицы (сумма диагональных элементов):
         tr(b)

Out[19]: 52

In [20]: # Извлечение диагональных элементов как массив:
         diag(b)

Out[20]: 4-element Array{Int64,1}:
           3
          20
           9
          20

In [21]: # Ранг матрицы:
         rank(b)

Out[21]: 4

In [22]: # Инверсия матрицы (определение обратной матрицы):
         inv(b)

Out[22]: 4x4 Array{Float64,2}:
          -0.334736   -0.496557    1.06083    -0.385233
           0.135807    0.075746   -0.161821    0.0248661
           0.199311    0.556236   -1.0065      0.374522
          -0.0539403  -0.0948738   0.15723     0.00420811

In [23]: # Определитель матрицы:
         det(b)

Out[23]: 2613.999999999997

In [24]: # Псевдобратная функция для прямоугольных матриц:
         pinv(a)

Out[24]: 3x4 Array{Float64,2}:
           0.00591363  -0.0447684   0.0477252   0.0210098
           0.0989015   -0.0418428  -0.0515636   0.0355021
          -0.109164     0.106832    0.0528722  -0.0445798
```

4.2.3. Вычисление нормы векторов и матриц, повороты, вращения

Для вычисления нормы используется LinearAlgebra.norm(x).

Евклидова норма:

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

p-норма:

$$\|\vec{A}\|_2 = \left(\sum_{i=1}^{n} |a_i|^p\right)^{\frac{1}{p}}$$

```
In [25]: # Создание вектора X:
         X = [2, 4, -5]
         # Вычисление евклидовой нормы:
         norm(X)

Out[25]: 6.708203932499369

In [26]: # Вычисление p-нормы:
         p = 1
         norm(X,p)

Out[26]: 11.0
```

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

```
In [27]: # Расстояние между двумя векторами X и Y:
         X = [2, 4, -5];
         Y = [1,-1,3];
         norm(X-Y)

Out[27]: 9.486832980505138

In [28]: # Проверка по базовому определению:
         sqrt(sum((X-Y).^2))

Out[28]: 9.486832980505138
```

Угол между двумя векторами $\vec{X}$ и $\vec{Y}$ определяется как

$$\cos^{-1}\frac{\vec{X}^T\vec{Y}}{\|\vec{X}\|_2\|\vec{Y}\|_2}$$


```
Out[28]: 9.486832980505138
```


```
In [29]: # Угол между двумя векторами:
         acos((transpose(X)*Y)/(norm(X)*norm(Y)))

Out[29]: 2.4404307889469252
```

Вычисление нормы для двумерной матрицы:

In [30]:
```
# Создание матрицы:
d = [5 -4 2 ; -1 2 3; -2 1 0]
# Вычисление Евклидовой нормы:
opnorm(d)
```

Out[30]:
```
7.147682841795258
```

In [31]:
```
# Вычисление p-нормы:
p=1
opnorm(d,p)
```

Out[31]:
```
8.0
```

Операции поворота и перестановки

In [32]:
```
# Поворот на 180 градусов:
rot180(d)
```

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

In [33]:
```
# Переворачивание строк:
reverse(d,dims=1)
```

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

In [34]:
```
# Переворачивание столбцов
reverse(d,dims=2)
```

Out[34]:
```
3×3 Array{Int64,2}:
 2  -4   5
 3   2  -1
 0   1  -2
```

4.2.4. Матричное умножение, единичная матрица, скалярное произведение

```
 0  1  -2
```

In [36]:
```julia
# 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
```

Out[36]:
```
2×4 Array{Int64,2}:
 58  133  152  37
 29   95   96  18
```

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

Out[37]:
```
3×3 Array{Int64,2}:
 1  0  0
 0  1  0
 0  0  1
```

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

Out[38]:
```
-17
```

In [39]:
```julia
# тоже скалярное произведение:
X'Y
```

Out[39]:
```
-17
```

4.2.5. Факторизация.

Решение систем линейный алгебраических уравнений $Ax = b$:

```
In [40]: # 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

Out[40]: 3-element Array{Float64,1}:
          0.9999999999999988
          1.000000000000003
          0.9999999999999977
```

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

```
In [41]: # LU-факторизация:
         Alu = lu(A)

Out[41]: LU{Float64,Array{Float64,2}}
         L factor:
         3x3 Array{Float64,2}:
          1.0       0.0       0.0
          0.489537  1.0       0.0
          0.765247  0.657155  1.0
         U factor:
         3x3 Array{Float64,2}:
          0.97224  0.600521   0.235646
          0.0      0.626827   0.75904
          0.0      0.0       -0.0935752
```

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

```
In [43]: # Матрица перестановок:
         Alu.P

Out[43]: 3×3 Array{Float64,2}:
          1.0  0.0  0.0
          0.0  0.0  1.0
          0.0  1.0  0.0

In [44]: # Вектор перестановок:
         Alu.p

Out[44]: 3-element Array{Int64,1}:
          1
          3
          2

In [45]: # Матрица L:
         Alu.L

Out[45]: 3×3 Array{Float64,2}:
          1.0       0.0       0.0
          0.489537  1.0       0.0
          0.765247  0.657155  1.0

In [46]: # Матрица U:
         Alu.U

Out[46]: 3×3 Array{Float64,2}:
          0.97224  0.600521   0.235646
          0.0      0.626827   0.75904
          0.0      0.0       -0.0935752
```

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

```
In [47]: # Решение СЛАУ через матрицу A:
         A\b
         # Решение СЛАУ через объект факторизации:
         Alu\b

Out[47]: 3-element Array{Float64,1}:
          0.9999999999999988
          1.0000000000000003
          0.9999999999999977
```

Аналогично можно найти детерминант матрицы:

```
In [48]: # Детерминант матрицы A:
         det(A)
         # Детерминант матрицы A через объект факторизации:
         det(Alu)

Out[48]: 0.05702716206432515
```

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

```
In [49]: # QR-факторизация:
         Aqr = qr(A)

Out[49]: LinearAlgebra.QRCompactWY{Float64,Array{Float64,2}}
         Q factor:
         3×3 LinearAlgebra.QRCompactWYQ{Float64,Array{Float64,2}}:
          -0.740183   0.575613   0.347562
          -0.566423  -0.255217  -0.783601
          -0.362347  -0.776875   0.514947
         R factor:
         3×3 Array{Float64,2}:
          -1.31351  -1.27177   -0.82293
           0.0      -0.592096  -0.693101
           0.0       0.0        0.0733256
```

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

```
In [50]: # Матрица Q:
         Aqr.Q

Out[50]: 3×3 LinearAlgebra.QRCompactWYQ{Float64,Array{Float64,2}}:
          -0.740183   0.575613   0.347562
          -0.566423  -0.255217  -0.783601
          -0.362347  -0.776875   0.514947

In [51]: # Матрица R:
         Aqr.R

Out[51]: 3×3 Array{Float64,2}:
          -1.31351  -1.27177   -0.82293
           0.0      -0.592096  -0.693101
           0.0       0.0        0.0733256

In [52]: # Проверка, что матрица Q - ортогональная:
         Aqr.Q'*Aqr.Q

Out[52]: 3×3 Array{Float64,2}:
           1.0          -1.66533e-16  -5.55112e-17
          -1.66533e-16   1.0          -1.11022e-16
          -5.55112e-17  -1.11022e-16   1.0
```

Примеры собственной декомпозиции матрицы *A*:

```
In [53]: # Симметризация матрицы A:
         Asym = A + A'

Out[53]: 3x3 Array{Float64,2}:
          1.94448   1.34452  0.711594
          1.34452   1.74294  1.50636
          0.711594  1.50636  1.74879

In [54]: # Спектральное разложение симметризованной матрицы:
         AsymEig = eigen(Asym)

Out[54]: Eigen{Float64,Float64,Array{Float64,2},Array{Float64,1}}
         values:
         3-element Array{Float64,1}:
          0.07257279791340743
          1.1563661257855435
          4.2072737028563045
         vectors:
         3x3 Array{Float64,2}:
           0.344159   0.762337  -0.548085
          -0.766294  -0.109249  -0.633134
           0.542539  -0.637893  -0.546575

In [55]: # Собственные значения:
         AsymEig.values

Out[55]: 3-element Array{Float64,1}:
          0.07257279791340743
          1.1563661257855435
          4.2072737028563045

In [56]: #Собственные векторы:
         AsymEig.vectors

Out[56]: 3x3 Array{Float64,2}:
           0.344159   0.762337  -0.548085
          -0.766294  -0.109249  -0.633134
           0.542539  -0.637893  -0.546575

In [57]: # Проверяем, что получится единичная матрица:
         inv(AsymEig)*Asym

Out[57]: 3x3 Array{Float64,2}:
          1.0          1.28786e-14  -1.37668e-14
          1.11022e-14  1.0           4.08562e-14
          0.0          2.13163e-14   1.0
```

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


```
          0.0          2.13163e-14   1.0
```


```
In [58]: # Матрица 1000 х 1000:
         n = 1000
         A = randn(n,n)
         # Симметризация матрицы:
         Asym = A + A'
         # Проверка, является ли матрица симметричной:
         issymmetric(Asym)

Out[58]: true
```

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

```
In [59]: # Добавление шума:
         Asym_noisy = copy(Asym)
         Asym_noisy[1,2] += 5eps()
         # Проверка, является ли матрица симметричной:
         issymmetric(Asym_noisy)

Out[59]: false
```

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

```
Out[59]: false
```

```
In [60]: # Явно указываем, что матрица является симметричной:
Asym_explicit = Symmetric(Asym_noisy)
```

```
Out[60]: 1000×1000 Symmetric{Float64,Array{Float64,2}}:
 -0.390747   0.472011   -3.26232    …  -2.01463    -0.355591    3.01247
  0.472011   2.69531    -0.360579       1.75718     2.00499    -0.238584
 -3.26232   -0.360579   -2.48607        1.28053     1.61766     0.801195
  1.21317   -2.22299    -0.690631       0.334739   -0.368579   -0.922732
  0.739758   0.816868    1.09867        0.291704   -1.4145      0.291559
 -2.75284    2.01914     0.564551   …   0.932538    1.55828     0.449386
 -0.429364   0.646839    0.623383       0.343439   -0.961892    0.188053
  1.05565    1.5086     -1.73853        2.19862     1.21499     0.957339
  0.418386  -0.648463    0.336312      -0.47278    -0.604306   -1.25492
  0.891095  -0.747875    1.08442        3.56127     0.27703    -0.451115
 -0.676965   0.916128   -2.87612    …  -0.666718    1.47281    -1.00589
 -0.866665  -0.0337551   0.206033      -0.671757    1.37675    -0.254201
 -0.485876  -1.71532    -2.16416       -0.0374015  -1.22261    -1.27722
  ⋮                                 ⋱
 -0.25432    1.41598    -0.475904       0.898854    1.82393    -1.3355
  2.11712   -0.25766    -0.256758      -1.45633    -2.62939     0.184545
 -0.715591   0.7556     -0.0725563  …  -2.86394     2.15389     0.205752
  0.153613   1.48935    -1.42631       -0.41135     1.96928    -0.752084
  0.268907  -1.26316    -0.70989       -1.31389     3.7536     -3.34728
```

```
In [ ]:
```

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

```
In [61]: using BenchmarkTools
# Оценка эффективности выполнения операции по нахождению
# собственных значений симметризованной матрицы:
@btime eigvals(Asym);

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

```
In [62]: # Оценка эффективности выполнения операции по нахождению
# собственных значений зашумлённой матрицы:
@btime eigvals(Asym_noisy);

  1.370 s (13 allocations: 7.92 MiB)
```

```
In [63]: # Оценка эффективности выполнения операции по нахождению
# собственных значений зашумлённой матрицы,
# для которой явно указано, что она симметричная:
@btime eigvals(Asym_explicit);

  241.651 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:

```
In [64]: # Трёхдиагональная матрица 1000000 х 1000000:
         n = 1000000;
         A = SymTridiagonal(randn(n), randn(n-1))
         # Оценка эффективности выполнения операции по нахождению
         # собственных значений:
         @btime eigmax(A)

           905.160 ms (38 allocations: 183.11 MiB)

Out[64]: 6.286148989922324
```

При попытке задать подобную матрицу обычным способом и посчитать её собственные значения, вы скорее всего получите ошибку переполнения памяти:

```
In [65]: B = Matrix(A)

         OutOfMemoryError()

         Stacktrace:
          [1] Array at .\boot.jl:408 [inlined]
          [2] Array at .\boot.jl:416 [inlined]
          [3] zeros at .\array.jl:525 [inlined]
          [4] zeros at .\array.jl:521 [inlined]
          [5] Array{Float64,2}(::SymTridiagonal{Float64,Array{Float64,1}}) at D:\buildbot\worker\package_win64
         \build\usr\share\julia\stdlib\v1.5\LinearAlgebra\src\tridiag.jl:127
          [6] Array{T,2} where T(::SymTridiagonal{Float64,Array{Float64,1}}) at D:\buildbot\worker\package_win
         64\build\usr\share\julia\stdlib\v1.5\LinearAlgebra\src\tridiag.jl:141
          [7] top-level scope at In[65]:1
          [8] include_string(::Function, ::Module, ::String, ::String) at .\loading.jl:1091
          [9] execute_code(::String, ::String) at C:\Users\Admin\.julia\packages\IJulia\rWZ9e\src\execute_requ
         est.jl:27
          [10] execute_request(::ZMQ.Socket, ::IJulia.Msg) at C:\Users\Admin\.julia\packages\IJulia\rWZ9e\src
         \execute_request.jl:86
          [11] #invokelatest#1 at .\essentials.jl:710 [inlined]
          [12] invokelatest at .\essentials.jl:709 [inlined]
          [13] eventloop(::ZMQ.Socket) at C:\Users\Admin\.julia\packages\IJulia\rWZ9e\src\eventloop.jl:8
          [14] (::IJulia.var"#15#18")() at .\task.jl:356
```

4.2.6. Общая линейная алгебра

In [66]:
```
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
```

Out[66]:
```
3-element Array{Rational{BigInt},1}:
 1//1
 1//1
 1//1
```

In [67]:
```
# LU-разложение:
lu(Arational)
```

Out[67]:
```
LU{Rational{BigInt},Array{Rational{BigInt},2}}
L factor:
3×3 Array{Rational{BigInt},2}:
 1//1   0//1   0//1
 1//6   1//1   0//1
 1//2  15//23  1//1
U factor:
3×3 Array{Rational{BigInt},2}:
 3//5   4//5    7//10
 0//1  23//30   1//12
 0//1   0//1   91//230
```

4.4. Самостоятельное выполнение

4.4.1. Произведение векторов

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

```
In [68]: using LinearAlgebra
         #1
         v=[3, 4 ,5]
         dot_v=dot(v,v)
         dot_v

Out[68]: 50
```

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

```
Out[68]: 50

In [69]: #Умножьте v матрично на себя (внешнее произведение), присвоив результат переменной outer_v.
         outer_v=cross(v,v)
         outer_v

Out[69]: 3-element Array{Int64,1}:
          0
          0
          0
```

4.4.2. Системы линейных уравнений

1. Решить СЛАУ с двумя неизвестными.

Системы решаются способом, указанном при повторении примеров (через матричное деление)

$$\begin{cases} x + y = 2 \\ x - y = 3 \end{cases}$$

```
In [70]: #2 Решить СЛАУ с двумя неизвестными.
         A=[1 1 ; 1 -1]
         b=[2 ; 3]
         A\b

Out[70]: 2-element Array{Float64,1}:
           2.5
          -0.5
```

$$\begin{cases} x + y = 2 \\ 2x + 2y = 4 \end{cases}$$

Данная система имеет линейно-зависимые строки, следовательно, бесконечное множество решений. Предыдущий способ решения результата не дает, обращаемся к библиотеке для решения нелинейых уравнений и получаем частное решение

In [71]:
```
A=[1 1 ; 2 2]
b=[2 ; 4]
lu(A)
A\b
```

```
SingularException(2)

Stacktrace:
 [1] checknonsingular at D:\buildbot\worker\package_win64\build\usr\share\julia\stdlib\v1.5\LinearAlg
ebra\src\factorization.jl:19 [inlined]
 [2] checknonsingular at D:\buildbot\worker\package_win64\build\usr\share\julia\stdlib\v1.5\LinearAlg
ebra\src\factorization.jl:21 [inlined]
```

In [72]:
```
using NLsolve

function f!(F, v)
           x = v[1]
           y = v[2]
           F[1] = x + y - 2
           F[2] = 2*x + 2*y - 4
    end
res=nlsolve(f!, [0.0; 0.0])
res.zero
```

Out[72]:
```
2-element Array{Float64,1}:
 0.9132041931152343
 1.0867958068847656
```

$$\begin{cases} x + y = 2 \\ 2x + 2y = 5 \end{cases}$$

Аналогично предыдущей системе

In [73]:
```
A=[1 1 ; 2 2]
b=[2 ; 5]
A\b
```

```
SingularException(2)

Stacktrace:
 [1] checknonsingular at D:\buildbot\worker\package_win64\build\usr\share\julia\stdlib\v1.5\LinearAlg
ebra\src\factorization.jl:19 [inlined]
```

In [74]:
```
using NLsolve

function f!(F, v)
           x = v[1]
           y = v[2]
           F[1] = x + y - 2
           F[2] = 2*x + 2*y - 5
    end
res=nlsolve(f!, [0.0; 0.0])
res.zero
```

Out[74]:
```
2-element Array{Float64,1}:
 -7.629159860605175
 10.029159860627605
```

$$\begin{cases} x + y = 2 \\ 2x + 2y = 1 \\ x - y = 3 \end{cases}$$

```
In [75]: A=[1 1 ; 2 2 ; 1 -1]
         b=[2 ; 1 ; 3]
         A\b

Out[75]: 2-element Array{Float64,1}:
           1.9000000000000001
          -1.0999999999999996
```

$$\begin{cases} x + y = 2 \\ 2x + y = 1 \\ 3x + 2y = 3 \end{cases}$$

```
In [76]: A=[1 1 ; 2 1 ; 3 2]
         b=[2 ; 1 ; 3]
         A\b

Out[76]: 2-element Array{Float64,1}:
          -0.9999999999999989
           2.9999999999999982
```

2. Решить СЛАУ с тремя неизвестными.

$$\begin{cases} x + y + z = 2 \\ x - y - 2z = 3 \end{cases}$$

```
In [78]: A=[1 1 1 ; 1 -1 -2]
         b=[2 ; 3]
         A\b

Out[78]: 3-element Array{Float64,1}:
           2.2142857142857144
           0.35714285714285704
          -0.5714285714285712
```

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

```
In [79]: A=[1 1 1 ; 2 2 -3 ; 3 1 1]
         b=[2 ; 4 ; 1]
         A\b

Out[79]: 3-element Array{Float64,1}:
          -0.5
           2.5
           0.0
```

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

In [80]:
```
A=[1 1 1 ; 1 1 2 ; 2 2 3]
b=[1 ; 0 ; 1]
A\b
```

```
SingularException(2)

Stacktrace:
 [1] checknonsingular at D:\buildbot\worker\package_win64\build\usr\share\julia\stdlib\v1.5\LinearAlg
ebra\src\factorization.jl:19 [inlined]
 [2] checknonsingular at D:\buildbot\worker\package_win64\build\usr\share\julia\stdlib\v1.5\LinearAlg
```

In [81]:
```
using NLsolve

function f!(F, v)
    x = v[1]
    y = v[2]
    z = v[3]
    F[1] = x + y  + z - 1
    F[2] = x + y + 2*z
    F[3] = 2*x + 2*y + 3*z -1
end
res=nlsolve(f!, [0.0; 0.0; 0.0])
res.zero
```

Out[81]:
```
3-element Array{Float64,1}:
  2.1541665449153977
 -0.154166544912257
 -1.0000000000019942
```

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

In [82]:
```
A=[1 1 1 ; 1 1 2 ; 2 2 3]
b=[1 ; 0 ; 0]
A\b
```

```
SingularException(2)

Stacktrace:
 [1] checknonsingular at D:\buildbot\worker\package_win64\build\usr\share\julia\stdlib\v1.5\LinearAlg
ebra\src\factorization.jl:19 [inlined]
 [2] checknonsingular at D:\buildbot\worker\package_win64\build\usr\share\julia\stdlib\v1.5\LinearAlg
ebra\src\factorization.jl:21 [inlined]
 [3] lu!(::Array{Float64,2}, ::Val{true}; check::Bool) at D:\buildbot\worker\package_win64\build\usr
\share\julia\stdlib\v1.5\LinearAlgebra\src\lu.jl:85
```

In [83]:
```
function f!(F, v)
    x = v[1]
    y = v[2]
    z = v[3]
    F[1] = x + y  + z - 1
    F[2] = x + y + 2*z
    F[3] = 2*x + 2*y + 3*z
end
res=nlsolve(f!, [0.0; 0.0; 0.0])
res.zero
```

Out[83]:
```
3-element Array{Float64,1}:
 -1.3210360630766114
  2.987702729750822
 -0.9999999999975434
```

4.4.3. Операции с матрицами

1. Приведите приведённые ниже матрицы к диагональному виду

Для приведения матриц к диагональному виду используем функцию LU разложения

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

In [85]:
```
A=[ 1 -2 ; -2 1 ]
Alu = lu(A)
```

Out[85]:
```
LU{Float64,Array{Float64,2}}
L factor:
2x2 Array{Float64,2}:
  1.0  0.0
 -0.5  1.0
U factor:
2x2 Array{Float64,2}:
 -2.0   1.0
  0.0  -1.5
```

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

In [87]:
```
A=[ 1 -2 ; -2 3 ]
Alu = lu(A)
```

Out[87]:
```
LU{Float64,Array{Float64,2}}
L factor:
2x2 Array{Float64,2}:
  1.0  0.0
 -0.5  1.0
U factor:
2x2 Array{Float64,2}:
 -2.0   3.0
  0.0  -0.5
```

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

In [33]:
```
A=[ 1 -2 0 ; -2 1 2 ; 0 2 0]
Alu = lu(A)
```

Out[33]:
```
LU{Float64,Array{Float64,2}}
L factor:
3x3 Array{Float64,2}:
  1.0   0.0   0.0
 -0.0   1.0   0.0
 -0.5  -0.75  1.0
U factor:
3x3 Array{Float64,2}:
 -2.0  1.0  2.0
  0.0  2.0  0.0
  0.0  0.0  1.0
```

2. Вычислите

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

```
In [88]: A = [1 -2 ; -2 1]
         A^10
```

```
Out[88]: 2×2 Array{Int64,2}:
           29525  -29524
          -29524   29525
```

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

```
-29524   29525
```

```
In [39]: B= [5 -2; -2 5]
         B^(1/2)
```

```
Out[39]: 2×2 Symmetric{Float64,Array{Float64,2}}:
           2.1889   -0.45685
          -0.45685   2.1889
```

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

```
In [89]: C = [1 -2 ; -2 1]
         C^(1/3)
```

```
Out[89]: 2×2 Symmetric{Complex{Float64},Array{Complex{Float64},2}}:
           0.971125+0.433013im  -0.471125+0.433013im
          -0.471125+0.433013im   0.971125+0.433013im
```

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

```
In [90]: D = [1 2 ; 2 3]
         D^(1/2)
```

```
Out[90]: 2×2 Symmetric{Complex{Float64},Array{Complex{Float64},2}}:
          0.568864+0.351578im  0.920442-0.217287im
          0.920442-0.217287im   1.48931+0.134291im
```

3 Найдите собственные значения матрицы A

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

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

```
In [91]: 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]
         @btime eig_A = Diagonal(eigvals(A))

           5.950 μs (11 allocations: 2.81 KiB)

Out[91]: 5×5 Diagonal{Float64,Array{Float64,1}}:
          -128.493    ⋅         ⋅        ⋅         ⋅
              ⋅     -55.8878    ⋅        ⋅         ⋅
              ⋅        ⋅     42.7522     ⋅         ⋅
              ⋅        ⋅         ⋅    87.1611      ⋅
              ⋅        ⋅         ⋅        ⋅     542.468
```

Выполняем LU разложение и получаем верхне- и нижне-диагональную матрицу

```
    ⋅        ⋅        ⋅        ⋅     542.468

In [92]: @btime Alu=lu(A)

           1.180 μs (3 allocations: 448 bytes)

Out[92]: LU{Float64,Array{Float64,2}}
         L factor:
         5×5 Array{Float64,2}:
          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 Array{Float64,2}:
          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

In [13]: E = [1 0 ; 0 1]
```

4.4.4. 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 x
Используя это определение, проверьте, являются ли матрицы продуктивными.

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

```
In [93]: E=[1 1 ; 1 1]
         A=[1 2 ; 3 4]
         b=[3 ; 3]
         (E-A)\b

Out[93]: 2-element Array{Float64,1}:
           3.0
          -3.0
```

Один из корней отрицательный – матрица не является продуктивной

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

```
-3.0

In [94]: E=[1 1 ; 1 1]
         A=[1 2 ; 3 4]
         b=[3 ; 3]
         (E-(1/2)*A)\b

Out[94]: 2-element Array{Float64,1}:
           6.0
          -6.0
```

Один из корней отрицательный – матрица не является продуктивной

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

```
-6.0

In [96]: E=[1 1 ; 1 1]
         A=[1 2 ; 3 4]
         b=[3 ; 3]
         (E-(1/10)*A)\b

Out[96]: 2-element Array{Float64,1}:
           29.999999999999957
          -29.99999999999995
```

Один из корней отрицательный – матрица не является продуктивной

2 Критерий продуктивности: матрица A является продуктивной тогда и только тогда, когда все элементы матрица $(E-A)^{-1}$ являются неотрицательными числами.

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

```
In [97]: E = [1 0 ; 0 1]
         A =  [ 1 2 ; 3 1]
         inv(E-A)

Out[97]: 2x2 Array{Float64,2}:
          -0.0  -0.333333
          -0.5   0.0
```

3 элемента отрицательны – матрица не является продуктивной

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

```
In [98]: inv(E-(1/2)*A)

Out[98]: 2x2 Array{Float64,2}:
          -0.4  -0.8
          -1.2  -0.4
```

4 элемента отрицательны – матрица не является продуктивной

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

```
 -1.2  -0.4

In [99]: inv(E-(1/10)*A)

Out[99]: 2x2 Array{Float64,2}:
          1.2  0.266667
          0.4  1.2
```

4 элемента положительны– матрица является продуктивной

3 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.

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

```
In [100]: eigvals(A)

Out[100]: 2-element Array{Float64,1}:
           -1.4494897427831779
            3.4494897427831783
```

оба значения по модулю превосходят 1 – матрица не является продуктивной

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

```
In [17]: eigvals(0.5*A)

Out[17]: 2-element Array{Float64,1}:
          -0.7247448713915892
           1.724744871391589
```

одно из значений по модулю превосходит 1 – матрица не является продуктивной

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

```
In [101]: eigvals(0.1*A)

Out[101]: 2-element Array{Float64,1}:
           -0.14494897427831785
            0.34494897427831783
```

оба значения по модулю меньше 1 – матрица является продуктивной

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

```
0.34494897427831783

In [19]: A = [0.1 0.2 0.3 ; 0 0.1 0.2 ; 0 0.1 0.3]
         eigvals(A)

Out[19]: 3-element Array{Float64,1}:
          0.02679491924311228
          0.1
          0.37320508075688774
```

все три значения по модулю меньше 1 – матрица является продуктивной

Выводы

Получал навыки работы с матричными функциями и операциями линейной алгебры