Лабораторная работа №1

по численным методам

Выполнила: Хренникова Ангелина

Группа: М8О-308Б-19

Вариант: 20

Задача: Реализовать алгоритм LU - разложения матриц (с выбором главного элемента) в виде программы. Используя разработанное программное обеспечение, решить систему линейных алгебраических уравнений (СЛАУ). Для матрицы СЛАУ вычислить определитель и обратную матрицу.

```
from sys import stdin
from copy import deepcopy

import numpy as np
import math
import copy
import cmath

class Matrix:

    #Инициализация матрицы
    def __init__(self, matrix):
        self.matrix = deepcopy(matrix)
        self.size = self._Size()

    #Печать
    def __str__(self):
        return '\n'.join(['\t'.join(['%f\t' % i for i in row]) for
                          row in self.matrix])

    #Доступ к элементу
    def __getitem__(self, index):
        return self.matrix[index]

    #Размер матрицы
    def _Size(self):
        rows = len(self.matrix)
        cols = 0
        for row in self.matrix:
            if (type(row) == int) | (type(row) == float):
                break
            if len(row) > cols:
```

```
                cols = len(row)
        return (rows, cols)

    #Алгоритм Дулиттла для получения ЛУ разложения
    def Get_LU_Doolittle(self):
        if self.size[0] != self.size[1]:
            raise Exception("Матрица должна быть квадратной")

        n = self.size[0]

        mat = np.zeros((n, n), float)
        U = Matrix(mat)
        L = Matrix(mat)

        for i in range(n):
            for k in range(i, n):
                sum = 0;
                for j in range(i):
                    sum += (L.matrix[i][j] * U.matrix[j][k]);

                U.matrix[i][k] = self.matrix[i][k] - sum;

            for k in range(i, n):
                if (i == k):
                    L.matrix[i][i] = 1;
                else:
                    sum = 0;
                    for j in range(i):
                        sum += (L.matrix[k][j] * U.matrix[j][i]);

                    L.matrix[k][i] = (self.matrix[k][i] - sum) /
U.matrix[i][i];

        return L, U

    #ЛУП разложение с выбором максимального элемента
    def Get_LUP(self):
        if self.size[0] != self.size[1]:
            raise Exception("Матрица должна быть квадратной")

        n = self.size[0]
        P = [i for i in range(n)]
        mat = np.zeros((n, n), float)
        LU = Matrix(mat)

        for i in range(n):
            for j in range(n):
                LU.matrix[i][j] = self.matrix[i][j]

        for k in range(n):
```

```
            main_elem = 0
            for i in range(k, n):
                if (abs(self[i][k]) > main_elem):
                    main_elem = abs(self[i][k])
                    row = i
            if (main_elem == 0):
                raise Exception('Столбец нулевой')

            P[k], P[row] = P[row], P[k]

            for i in range(n):
                LU.matrix[k][i], LU.matrix[row][i] = LU.matrix[row]
[i], LU.matrix[k][i]
            for i in range(k + 1, n):
                LU.matrix[i][k] /= LU.matrix[k][k]
                for j in range(k + 1, n):
                    LU.matrix[i][j] -= LU.matrix[i][k] * LU.matrix[k]
[j]

        return LU, P, main_elem

    #ЛУ разложение
    def Get_LU(self):
        if self.size[0] != self.size[1]:
            raise Exception("Матрица должна быть квадратной")

        n = self.size[0]

        mat = np.zeros((n, n), float)
        U = Matrix(mat)
        L = Matrix(mat)
        LU, P, p = self.Get_LUP()

        for i in range(n):
            L.matrix[i][i] = 1
            for j in range(n):
                if (j < i):
                    L.matrix[i][j] = LU.matrix[i][j]
                else:
                    U.matrix[i][j] = LU.matrix[i][j]

        return L, U

    #Определитель
    def Det(self):
        if self.size[0] != self.size[1]:
            raise Exception("Матрица должна быть квадратной")
        n = self.size[0]
        L, U = self.Get_LU()
        LU, P, p = self.Get_LUP()
```

```
        det = pow(-1, p)
        for k in range(n):
            det *= U.matrix[k][k]
        return det

    def Multiply(n, m):
        if n.size[1] != m.size[0]:
            raise Exception("Несоответствие размерностей")
        res = []
        rows = []
        for i in range(n.size[0]):
            for j in range(m.size[1]):
                val = 0
                for k in range(n.size[1]):
                    val += n.matrix[i][k] * m.matrix[k][j]

                rows.append(val)
            res.append(rows)
            rows = []
        return Matrix(res)

    def Sum(self, m):
        if self.size != m.size:
            raise Exception("Несоответствие размерностей: {0}
{1}".format(self.size, m.size))
        res = []
        rows = []
        for i, row in enumerate(self.matrix):
            for j, col in enumerate(row):
                rows.append(self.matrix[i][j] + m.matrix[i][j])
            res.append(rows)
            rows = []
        return Matrix(res)

    def MultiNum(self, n):
        res = []
        rows = []
        for i, row in enumerate(self.matrix):
            for j, col in enumerate(row):
                rows.append(n * self.matrix[i][j])
            res.append(rows)
            rows = []
        return Matrix(res)

    def Transpose(self):
        res = self
        if self.size[0] == self.size[1]:
            for i in range(self.size[0]):
                for j in range(i + 1, self.size[0]):
                    a = res.matrix[i][j]
```

```
                    res.matrix[i][j] = res.matrix[j][i]
                    res.matrix[j][i] = a
                    a = 0
            return res
        else:
            res = []
            for i in range(self.size[1]):
                rows = []
                for j in range(self.size[0]):
                    rows.append(self.matrix[j][i])
                res.append(rows)
            return Matrix(res)

    #Обратная матрица
    def Invert(self):
        if self.size[0] != self.size[1]:
            raise Exception("Матрица должна быть квадратной")

        n = self.size[0]
        mat = np.zeros((n, n), float)
        res = Matrix(mat)
        for k in range(n):
            x = get_solution(self, e(k, self.size[0]))
            for i in range(n):
                res.matrix[i][k] = x[i]

        return res

#Вектор с одной единицей и остальными нулями
def e(i, n):
    e = []
    for j in range(n):
        if j == i:
            e.append(1)
        else:
            e.append(0)
    return e

#Решение
def get_solution(A, b):
    L, U = A.Get_LU()
    LU, P, p = A.Get_LUP()

    n = A.size[0]
    x = [0] * n
    y = [0] * n

    #Ly = b
    for i in range(n):
        sum = 0
```

```
        for j in range(i):
            sum += L.matrix[i][j] * y[j]

        y[i] = b[P[i]] - sum

    #Ux = y
    for i in range(n - 1, -1, -1):
        sum = 0
        for j in range(i + 1, n):
            sum += U.matrix[i][j] * x[j]

        x[i] = (y[i] - sum) / U.matrix[i][i]

    return x

A = Matrix([[7, 8, 4, -6], [-1, 6, -2, -6], [2, 9, 6, -4], [5, 9, 1,
1]])
B = [-126, -42, -115, -67]
print("Result: {}".format(get_solution(A, B)))
```

```
Result: [-3.999999999999998, -5.0, -7.0, 5.000000000000002]
```

```
A.Det()
```

```
2865.9999999999995
```

```
L, U = A.Get_LU()
print("Result:\n{}".format(L.Multiply(U)))
```

```
Result:
7.000000   8.000000   4.000000   -6.000000
2.000000   9.000000   6.000000   -4.000000
-1.000000  6.000000   -2.000000  -6.000000
5.000000   9.000000   1.000000   1.000000
```

```
print("Result:\n{}".format(A.Invert()))
```

```
Result:
0.146546   -0.059316  -0.122819  0.032100
-0.070482  0.052338   0.049546   0.089323
-0.005234  -0.105024  0.147244   -0.072575
-0.093161  -0.069435  0.020935   0.108165
```

Задача: Реализовать метод прогонки в виде программы, задавая в качестве входных данных ненулевые элементы матрицы системы и вектор правых частей. Используя разработанное программное обеспечение, решить СЛАУ с трехдиагональной матрицей.

```
#Метод прогонки
def Progonka(A, b):
    n = A.size[0]

    x = [0 for k in range(0, n)]
    v = [0 for k in range(0, n)]
```

```
    u = [0 for k in range(0, n)]

    v[0] = A.matrix[0][1] / (-1 * A.matrix[0][0])
    u[0] = (-1 * b.matrix[0]) / (-1 * A.matrix[0][0])

    for i in range(1, n - 1):
        v[i] = A.matrix[i][i+1] / ( -1 * A.matrix[i][i] - A.matrix[i]
[i-1] * v[i-1])
        u[i] = (A.matrix[i][i-1] * u[i-1] - b.matrix[i]) / (-1 *
A.matrix[i][i] - A.matrix[i][i-1] * v[i-1])

    v[n-1] = 0
    u[n-1] = (A.matrix[n-1][n-2] * u[n-2] - b.matrix[n-1]) / (-1 *
A.matrix[n-1][n-1] - A.matrix[n-1][n-2] * v[n-2])

    x[n-1] = u[n-1]
    for i in range(n-1, 0, -1):
        x[i-1] = v[i-1] * x[i] + u[i-1]

    print("v:\n{}".format(v))
    print("u:\n{}".format(u))

    return x

A = Matrix([[-6, 6, 0, 0, 0], [2, 10, -7, 0, 0], [0, -8, 18, 9, 0],
[0, 0, 6, -17, -6], [0, 0, 0, 9, 14]])
b = Matrix([30, -31, 108, -114, 124])
Progonka(A, b)

v:
[1.0, 0.5833333333333334, -0.675, -0.2850356294536817, 0]
u:
[-5.0, -1.75, 7.050000000000001, 7.425178147268409, 5.0]

[-5.0, 0.0, 3.0, 6.0, 5.0]
```

Задача: Реализовать метод простых итераций и метод Зейделя в виде программ, задавая в качестве входных данных матрицу системы, вектор правых частей и точность вычислений. Используя разработанное программное обеспечение, решить СЛАУ. Проанализировать количество итераций, необходимое для достижения заданной точности.

```
#Метод простых итераций
def Iter(A, b, eps):
    t = True
    n = A.size[0]
    x = [0 for k in range(n)]
    x_ = [0 for k in range(n)]
    num_it = 0
    while (t):
        for i in range(n):
```

```
            s = 0
            for j in range(n):
                if (i != j):
                    s += A.matrix[i][j] * x[j]
            x_[i] = (b[i] - s) / A.matrix[i][i]
        num_it += 1

        if (x_[0] == None):
             t = False
        res = [0 for k in range(n)]

        for i in range(n):
            res[i] = pow(x[i] - x_[i],2)

        if (math.sqrt(sum(res)) > eps):
             t = True
        else: t = False

        x = copy.copy(x_)
    return x_, num_it

A = Matrix([[10, -1, -2, 5], [4, 28, 7, 9], [6, 5, -23, 4], [1, 4, 5,
-15]])
B = [-99, 0, 67, 58]

x, it = Iter(A, B, 0.0001)
x,it
```

```
([-8.000013559736155,
  3.999973220901675,
  -4.999995764128967,
  -4.999989701911018],
 16)
```

```
#Метод Зейделя
def Zeydel(A, b, eps):
    t = True
    num_it = 0
    n = A.size[0]
    x = [0 for i in range(n)]
    x_ = [0 for i in range(n)]
    while (t):
        for i in range(n):
            s = 0
            for j in range(n):
                if (j < i):
                    s += A.matrix[i][j] * x_[j]

                elif i != j:
                    s += A.matrix[i][j] * x[j]
```

```
            x_[i] = (b[i] - s) / A[i][i]

        num_it += 1

        if (x_[0] == None):
             t = False
        res = [0 for k in range(n)]

        for i in range(n):
            res[i] = pow(x[i] - x_[i],2)

        if (math.sqrt(sum(res)) > eps):
             t = True
        else: t = False

        x = copy.copy(x_)
    return x_, num_it

x, it = Zeydel(A, B, 0.0001)
x,it
```

```
([-7.999999424480523,
  3.999999158869865,
  -5.00000016872013,
  -5.0000002421734475],
 7)
```

Задача: Реализовать метод вращений в виде программы, задавая в качестве входных данных матрицу и точность вычислений. Используя разработанное программное обеспечение, найти собственные значения и собственные векторы симметрических матриц. Проанализировать зависимость погрешности вычислений от числа итераций.

```
A = Matrix([[-7, -9, 1], [-9, 7, 2], [1, 2, 9]])

def t(A):
    a = 0
    for l in range(A.size[0]):
        for m in range(l + 1, A.size[0]):
            a += A.matrix[l][m] * A.matrix[l][m]
    a = math.sqrt(a)
    return a

def max_ij(A):
    if (A.size[0] != A.size[1]):
        raise Exception("Матрица должна быть квадратной")

    m = 0
    i = 0
    j = 0
    f = 0
    n = A.size[0]
    for k in range(n):
```

```
        for g in range(n):
            if ((abs(A.matrix[k][g]) > m or abs(A.matrix[k][g]) == 0)
and k < g):
                i = k
                j = g
                m = abs(A.matrix[k][g])
                f = A.matrix[k][g]

    if ((i == 0) and (j == 0)):
        raise Exception("Матрица вырожденная")
    return i, j, f

def Multiply(n, m):
    if n.size[1] != m.size[0]:
        raise Exception("Несоответствие размерностей")
    res = []
    rows = []
    for i in range(n.size[0]):
        for j in range(m.size[1]):
            val = 0
            for k in range(n.size[1]):
                val += n.matrix[i][k] * m.matrix[k][j]
            rows.append(val)
        res.append(rows)
        rows = []
    return Matrix(res)

def Rotation(M, eps):
    Ak = M
    num_it = 0
    e = t(M)
    n = M.size[0]
    while e > eps:
        mat = np.zeros((n, n), float)
        R = Matrix(mat)
        num_it += 1
        i, j, m = max_ij(Ak)
        if (Ak.matrix[i][i] - Ak.matrix[j][j] != 0):
            phi = math.atan((2 * Ak.matrix[i][j]) / (Ak.matrix[i][i] -
Ak.matrix[j][j])) / 2
        else:
            phi = math.pi / 4
        for r in range(Ak.size[0]):
            for c in range(Ak.size[0]):
                if r == c:
                    R.matrix[r][c] = 1
        c1 = math.cos(phi)
        s1 = math.sin(phi)
        R.matrix[i][i] = c1
        R.matrix[i][j] = -s1
```

```
            R.matrix[j][i] = s1
            R.matrix[j][j] = c1

            F = Multiply(Ak, R)
            T = copy.deepcopy(R)
            Ak = Multiply((R.Transpose()), F)
            e = t(Ak)
        res = []
        for l in range(Ak.size[0]):
            res.append(Ak.matrix[l][l])

        print("Iter:", num_it)
        print("Res:", res)

Rotation(A, 0.01)
```

```
Iter: 5
Res: [-11.555975526027277, 12.036590138386002, 8.519385387641274]
```

Задача: Реализовать алгоритм QR – разложения матриц в виде программы. На его основе разработать программу, реализующую QR – алгоритм решения полной проблемы собственных значений произвольных матриц, задавая в качестве входных данных матрицу и точность вычислений. С использованием разработанного программного обеспечения найти собственные значения матрицы.

```
A = Matrix([[6, 5, -6], [4, -6, 9], [-6, 6, 1]])

def Householder(v):
    n = len(v)
    v1 = []
    for i in range(n):
        rows = []
        for j in range(n):
            rows.append(v[i] * v[j])
        v1.append(rows)
    v2 = 0
    mat = np.eye(n)
    E= Matrix(mat)
    for i in range(n):
        v2 += v[i] * v[i]
    return E.Sum(Matrix(v1).MultiNum(-2 / v2))

def H(v):
    v1 = []
    for i in range(len(v)):
        rows = []
        for j in range(len(v)):
            rows.append(v[i] * v[j])
        v1.append(rows)
    v2 = 0
    for i in range(len(v)):
```

```
        v2 += v[i] * v[i]
    return Matrix.E(len(v)).Sum(Matrix(v1).MultiNum(-2 / v2))

def sign(x):
    return -1 if x < 0 else 1 if x > 0 else 0


def get_QR(A):
    n = A.size[0]
    mat = np.eye(n)
    Q = Matrix(mat)
    Ak = A

    for i in range(n):
        v = []
        for j in range(n):
            if j < i:
                v.append(0)
            elif i == j:
                a = 0
                for k in range(n):
                    a += Ak.matrix[k][i] * Ak.matrix[k][i]
                v.append(Ak.matrix[j][i] + sign(Ak.matrix[j][i]) *
math.sqrt(a))
            else:
                v.append(A.matrix[j][i])
        Ak = Householder(v).Multiply(Ak)
        Q = Q.Multiply(Householder(v))
    return Q, Ak

Q, R = get_QR(A)
print("Q:\n {}".format(Q))
print("R:\n {}".format(R))
print("A = QR = \n {}".format(Q.Multiply(R)))
```

```
Q:
 -0.639602 0.739176   -0.211015
-0.426401  -0.569562  -0.702696
0.639602   0.359469   -0.679479
R:
 -9.380832 3.198011   0.639602
0.000000   9.270064   -9.201645
-0.000000  -0.915773  -5.737650
A = QR =
 6.000000  5.000000   -6.000000
4.000000   -6.000000  9.000000
-6.000000  6.000000   1.000000
```

```
def eps_2(A, k):
    e = 0
    for l in range(k + 1, A.size[0]):
```

```
            e += A.matrix[l][k] * A.matrix[l][k]
        e = math.sqrt(e)
        return e

    def eps_1(A, k):
        e = 0
        for l in range(k + 2, A.size[0]):
            e += A.matrix[l][k] * A.matrix[l][k]
        e = math.sqrt(e)
        return e

    def eps_l(l):
        return abs(l)

    def solve_lambda(A, k):
        b = A.matrix[k][k] + A.matrix[k + 1][k + 1]
        c = A.matrix[k][k] * A.matrix[k + 1][k + 1] - A.matrix[k][k + 1] *
    A.matrix[k + 1][k]
        d = b * b - 4 * c

        return complex((complex(-b) + cmath.sqrt(d)) / complex(2)),
    complex((complex(-b) - cmath.sqrt(d)) / complex(2))

    def QR_values(A, eps=0.01):
        if A.size[0] != A.size[1]:
            raise Exception("Матрица должна быть квадратной")
        it = 0
        Q, R = get_QR(A)
        res = []
        cmplx = False
        for k in range(A.size[0]):
            if cmplx == True:
                cmplx = False
                continue
            it += 1
            cmplx = False
            Ak = R.Multiply(Q)
            e_1 = eps_1(Ak, k)
            e_2 = eps_2(Ak, k)
            count = 0
            while e_1 > eps:
                it += 1
                Q, R = get_QR(Ak)
                Ak = R.Multiply(Q)
                e_1 = eps_1(Ak, k)

                l_1, l_2 = solve_lambda(Ak, k)

            while e_2 > eps:
                count += 1
```

```
            it += 1

            Q, R = get_QR(Ak)
            Ak = R.Multiply(Q)
            e_2 = eps_2(Ak, k)

            lk_1, lk_2 = solve_lambda(Ak, k)

            e_l_1 = eps_l(l_1 - lk_1)
            e_l_2 = eps_l(l_2 - lk_2)

            l_1 = lk_1
            l_2 = lk_2
            if count > 100:
                cmplx = True
                while (e_l_1 > eps) & (e_l_2 > eps):
                    it += 1
                    lk_1, lk_2 = solve_lambda(Ak, k)

                    e_l_1 = eps_l(l_1 - lk_1)
                    e_l_2 = eps_l(l_2 - lk_2)

                    l_1 = lk_1
                    l_2 = lk_2
                break

        if cmplx == True:
            res.append(l_1)
            res.append(l_2)
        else:
            res.append(Ak.matrix[k][k])

    print("Result: {}".format(res))
    print("Iterations: {}".format(it))

QR_values(A)
```

```
Result: [-13.27548089985948, 9.86073051001206, 4.414750389847406]
Iterations: 32
```