**Московский Авиационный Институт**

(Национальный Исследовательский Университет)

Институт №8 «Компьютерные науки и прикладная математика»

**Лабораторная работа №1**

**«Численные методы»**

**Вариант 9**

**Выполнила студентка группы**: М8О-305Б-21

Бондарева Елена Евгеньевна

# Преподаватель: Филиппов Глеб Сергеевич

Оценка: !

Дата: !

Москва, 2024

**Задание 1.1**

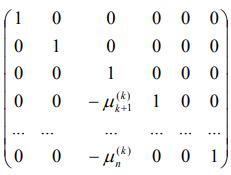
Реализовать алгоритм LU - разложения матриц (с выбором главного элемента) в виде программы. Используя разработанное программное обеспечение, решить систему линейных алгебраических уравнений (СЛАУ). Для матрицы СЛАУ вычислить определитель и обратную матрицу.



**Теоретические сведения**

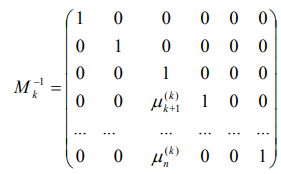
LU – разложение представляет собой разложение матрицы A в произведение нижней и верхней треугольных матриц, т.е. , где L - нижняя треугольная матрица (матрица, у которой все элементы, находящиеся выше главной диагонали равны нулю, ), U - верхняя треугольная матрица (матрица, у которой все элементы, находящиеся ниже главной диагонали равны нулю, ).

LU – разложение может быть построено с использованием метода Гаусса. Рассмотрим k-ый шаг метода Гаусса, на котором осуществляется обнуление поддиагональных элементов k-го столбца матрицы :

В терминах матричных операций такая операция эквивалентна умножению , где элементы матрицы определяются следующим образом:

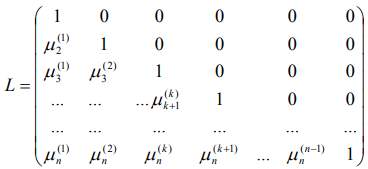
Т.е. матрица имеет вид:

При этом выражение для обратной операции запишется в виде:



В результате прямого хода метода Гаусса получим ,

где – верхняя треугольная матрица, а – нижняя треугольная матрица, имеющая вид:



В дальнейшем *LU-*разложение может быть использовано при решении системы линейных алгебраических уравнений вида Подставляя *LU*-разложение в СЛАУ, получим , т.е. процесс СЛАУ сводится к двум простым этапам.

На первом этапе решается СЛАУ , поскольку матрица системы – нижняя треугольная, решение можно записать в явном виде:

На втором этапе решается СЛАУ с верхней треугольной матрицей. Здесь, как и на предыдущем этапе, решение представляется в явном виде:

Зная 𝐿𝑈-разложение легко также получить определитель матрицы 𝐴 и обратную ей матрицу. Определитель находится как произведение элементов на главных диагоналях матриц 𝐿 и 𝑈:

Обратную матрицу можно найти из отношения . Это уравнение также можно решить методом 𝐿𝑈-разложения.

**Код программы:**

import copy

def matrixmult(A, B):

    C = [[0.0 for col in range(len(B[0]))] for row in range(len(A))]

    for i in range(len(A)):

        for j in range(len(B[0])):

            for k in range(len(B)):

                C[i][j] += A[i][k]\*B[k][j]

    return C

def pivot\_matrix(M):

    m = len(M)

    MCopy = copy.deepcopy(M)

    id\_mat = [[float(i==j) for i in range(m)] for j in range(m)]

    row\_exchanges = 0

    for i in range(m):

        maxElem = abs(MCopy[i][i])

        maxRow = i

        for k in range(i+1, m):

            if(abs(MCopy[k][i]) > maxElem):

                maxElem = abs(MCopy[k][i])

                maxRow = k

        if i != maxRow:

            id\_mat[i], id\_mat[maxRow] = id\_mat[maxRow], id\_mat[i]

            MCopy[i], MCopy[maxRow] = MCopy[maxRow], MCopy[i]

            row\_exchanges += 1

    return id\_mat, row\_exchanges

def lup\_decomposition(A):

    n = len(A)

    L = [[0.0] \* n for i in range(n)]

    U = [[0.0] \* n for i in range(n)]

    P, rowExc = pivot\_matrix(A)

    PA = matrixmult(P, A)

    for j in range(n):

        L[j][j] = 1.0

        for i in range(j+1):

            s = sum(L[i][k] \* U[k][j] for k in range(i))

            U[i][j] = PA[i][j] - s

        for i in range(j, n):

            s = sum(L[i][k] \* U[k][j] for k in range(j))

            L[i][j] = (PA[i][j] - s) / U[j][j]

    return (P, L, U, rowExc)

def lup\_solve(P,L,U,B):

    n = len(P)

    Bt = matrixmult(P, [[i] for i in B])

    Y = [0.0 for i in range(n)]

    for i in range(n):

        Y[i] = Bt[i][0]/L[i][i]

        for k in range(i):

            Y[i] -= Y[k]\*L[i][k]

    X = [0.0 for i in range(n)]

    for i in range(n-1,-1,-1):

        s = sum(X[k]\*U[i][k] for k in range(i+1,n))

        X[i] = (Y[i] - s)/U[i][i]

    return X

def lup\_invert(P,L,U):

    n = len(P)

    IA = [[float(i==j) for i in range(n)] for j in range(n)]

    for i in range(n) :

        b = [IA[i][k] for k in range(n)]

        IA[i] = lup\_solve(P,L,U,b)

    for i in range(n):

        for j in range(i):

            IA[i][j],IA[j][i] = IA[j][i],IA[i][j]

    return IA

def lup\_determinant(U, rowExc):

    n = len(U[0])

    det = U[0][0]

    for j in range(1,n):

        det \*= U[j][j]

    det \*= (-1)\*\*rowExc

    return det

def matmult(a,b):

   zip\_b = zip(\*b)

   zip\_b = list(zip\_b)

   return [[sum(elem\_a\*elem\_b for elem\_a, elem\_b in zip(row\_a, col\_b)) for col\_b in zip\_b] for row\_a in a]

def main():

    print("Matrix A:")

    A = []

    A.append([int(j) for j in input().strip().split(" ")])

    for i in range(1,len(A[0])) :

        A.append([int(j) for j in input().strip().split(" ")])

    print("Vector b:")

    b = [int(j) for j in input().strip().split(" ")]

    P, L, U, rowExc = lup\_decomposition(A)

    print("Matrix P:")

    for elem in P:

        print(elem)

    print("Matrix L:")

    for elem in L:

        print(elem)

    print("Matrix U:")

    for elem in U:

        print(elem)

    LU = matmult(L, U)

    print("Matrix L\*U")

    for elem in LU:

        print(elem)

    x = lup\_solve(P,L,U,b)

    print("X:")

    print(x)

    IA = lup\_invert(P,L,U)

    print("Inverted A:")

    for elem in IA:

        print(elem)

    AAIA = matmult(A, IA)

    print("Matrix A\*IA")

    for elem in AAIA:

        print(elem)

    detA = lup\_determinant(U, rowExc)

    print("Determinant of A: %d" % detA)

if \_\_name\_\_ == '\_\_main\_\_':

    main()

**Вывод программы:**

Matrix A:

-7 -9 1 -9

-6 -8 -5 2

-3 6 5 -9

-2 0 -5 -9

Vector b:

29 42 11 75

Matrix P:

[1.0, 0.0, 0.0, 0.0]

[0.0, 1.0, 0.0, 0.0]

[0.0, 0.0, 1.0, 0.0]

[0.0, 0.0, 0.0, 1.0]

Matrix L:

[1.0, 0.0, 0.0, 0.0]

[0.8571428571428571, 1.0, 0.0, 0.0]

[0.42857142857142855, -34.49999999999991, 1.0, 0.0]

[0.2857142857142857, -8.999999999999975, 0.29367088607594943, 1.0]

Matrix U:

[-7.0, -9.0, 1.0, -9.0]

[0.0, -0.2857142857142865, -5.857142857142857, 9.714285714285714]

[0.0, 0.0, -197.49999999999943, 329.99999999999903]

[0.0, 0.0, 0.0, -15.911392405063282]

Matrix L\*U

[-7.0, -9.0, 1.0, -9.0]

[-6.0, -8.0, -5.0, 2.0]

[-3.0, 6.000000000000001, 5.0, -9.0]

[-2.0, 0.0, -4.999999999999993, -9.0]

X:

[-4.00000000000003, 2.000000000000025, -8.000000000000005, -3.0000000000000018]

Inverted A:

[0.062052505966587415, -0.18042959427207686, -0.14749403341288794, 0.045346062052506075]

[-0.10103420843277668, 0.06300715990453493, 0.09912490055688153, 0.015910898965791523]

[0.06682577565632457, -0.0558472553699283, 0.02577565632458235, -0.10501193317422441]

[-0.05091487669053304, 0.07112171837708839, 0.018456642800318234, -0.06284805091487673]

Matrix A\*IA

[1.0, -1.1102230246251565e-16, 2.7755575615628914e-17, 0.0]

[-4.163336342344337e-17, 1.0, 1.8735013540549517e-16, -2.7755575615628914e-17]

[-2.1094237467877974e-15, 3.219646771412954e-15, 1.0000000000000007, -5.551115123125783e-16]

[-3.3306690738754696e-16, -2.220446049250313e-16, 2.7755575615628914e-17, 1.0000000000000004]

Determinant of A: 6284

**Вывод:**

В результате выполнения лабораторной работы мной был изучен численный метод LU разложения матрицы и решения системы линейных уравнений этим методом. Было установлено, что LU разложение матрицы позволяет представить матрицу в виде произведения верхнетреугольной и нижнетреугольной матрицы, что упрощает решение системы линейных уравнений. Были разработаны алгоритмы LU разложения матрицы и решения системы линейных уравнений с использованием этого метода, а также проведено сравнение этого метода с методом Гаусса на примере решения системы линейных уравнений.

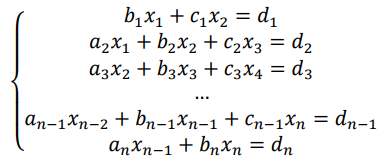
**Задание 1.2**

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.



**Теоретические сведения**

Метод прогонки является частным случаем метода Гаусса. Он применяется для решения СЛАУ с трехдиагональными матрицами. Рассмотрим СЛАУ:



При этом будем полагать, что . Решение системы можно искать в виде:

Здесь – прогоночные коэффициенты, определяемы по формулам:

После того как будут найдены прогоночные коэффициенты, можно вычислить значения неизвестных путем обратной подстановки:

Достаточным условием корректности метода прогонки и устойчивости его к погрешностям вычислений является условие преобладания диагональных коэффициентов:

**Код программы:**

def progon(a, b, c, d):

    l = len(a)

    P, Q, X = [], [], [0]\*l

    P.append(-c[0] / b[0])

    Q.append(d[0] / b[0])

    for i in range(1, l):

        if i == l - 1:

            p = 0

        else:

            p = - c[i] / (b[i] + a[i] \* P[i-1])

        q = (d[i] - a[i] \* Q[i-1]) / (b[i] + a[i] \* P[i-1])

        P.append(p)

        Q.append(q)

    X[l-1] = Q[l-1]

    for i in range(l - 2, -1, -1):

        X[i] = P[i] \* X[i+1] + Q[i]

    return X

print("Введите коэффициенты А:")

A = [0] + list(map(int, input().split()))

print("Введите коэффициенты В:")

B = list(map(int, input().split()))

print("Введите коэффициенты С:")

C = list(map(int, input().split())) + [0]

print("Введите правую часть:")

D = list(map(int, input().split()))

otvet = progon(A, B, C, D)

print("\nОтвет:")

if len(otvet) != 1:

    for i in range(len(otvet)):

        print("x{0} = {1}".format(i+1, otvet[i]))

else:

    print(otvet[0])

**Вывод программы:**

Введите коэффициенты А:

-2 2 -8 -7

Введите коэффициенты В:

8 12 -9 17 13

Введите коэффициенты С:

-4 -7 1 -4

Введите правую часть:

32 15 -10 133 -76

Ответ: x1 = 6.0 x2 = 4.0 x3 = 3.0 x4 = 9.0 x5 = -1.0000000000000002

**Вывод:**

Результаты выполнения лабораторной работы было показано, что метод прогонки является эффективным численным методом для решения систем линейных уравнений с трехдиагональной матрицей; а также были разработаны алгоритмы решения системы линейных уравнений методом прогонки, а также проведено сравнение этого метода с методом Гаусса на примере решения системы линейных уравнений. Этот метод широко используется в практических приложениях, где необходимо решить системы линейных уравнений с большим количеством неизвестных, и он может значительно ускорить вычислительный процесс.

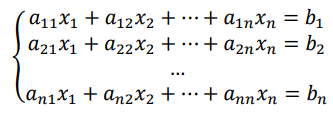
**Задание 1.3**

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.

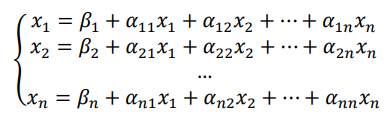


**Теоретические сведения**

Рассмотрим СЛАУ с невырожденной матрицей:



Приведем СЛАУ к эквивалентному виду:

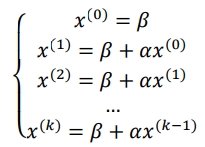


или в векторно-матричной форме

Такое приведение может быть выполнено различными способами. Одним из наиболее распространенных является следующий. Разрешим систему относительно неизвестных при ненулевых диагональных элементах (если какой-либо коэффициент на главной диагонали равен нулю, достаточно соответствующее уравнение поменять местами с любым другим уравнением). Получим следующие выражения для компонентов вектора и матрицы эквивалентной системы:

В качестве нулевого приближения вектора неизвестных примем вектор

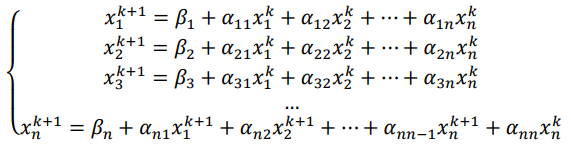
правых частей . Тогда метод простых итераций примет вид:



Достаточным условием сходимости является диагональное преобладание матрицы 𝐴 по строкам или по столбцам:

Критерием окончания итерационного процесса может служить неравенство

Метод простых итераций довольно медленно сходится. Для его ускорения существует метод Зейделя, заключающийся в том, что при вычислении компонента вектора неизвестных на (*k+1*)-й итерации используются уже вычисленные на (*k+1*)-й итерации. Тогда метод Зейделя для известного вектора на *k*-ой итерации имеет вид:



**Код программы:**

1. метод простых итераций

import sys

def check\_iter(A):

   n = len(A[0])

   for i in range(n):

      suma = sum(abs(x) for x in A[i]) - abs(A[i][i])

      if abs(A[i][i])<= suma:

         return False

   return True

def simiter(A,b, eps):

   n = len(b)

   x = [0,0]

   x[0] = [b[i] \* 1.0 / A[i][i] for i in range(n)]

   x[1] = [0 for \_ in range(n)]

   alnorm = max(sum(abs(1.0\*A[j][k]/A[j][j]) for k in range(n) if k != j) for j in range(n))

   coef = (alnorm/(1-alnorm)) if alnorm < 1 else 1

   it = 0

   while True:

      it += 1

      for j in range(n):

         x[it%2][j] = (b[j] - sum(A[j][k]\*x[(it+1)%2][k] for k in range(n) if k != j)) \* 1.0 / A[j][j]

      normx = coef \* sum(abs(x[it%2][k]-x[(it+1)%2][k]) for k in range(n))

      if normx <= eps:

         return x[it%2], it

def main():

   print("Matrix A:")

   A = []

   A.append([int(j) for j in input().strip().split(" ")])

   for i in range(1,len(A[0])) :

      A.append([int(j) for j in input().strip().split(" ")])

   print("Vector b:")

   b = [int(j) for j in input().strip().split(" ")]

   if not(check\_iter(A)):

      print("Not applicable")

      return

   print("Epsilon:")

   eps = float(input())

   x, it = simiter(A,b,eps)

   print("X:")

   print(x)

   print("Number of iterations:")

   print(it)

if \_\_name\_\_ == '\_\_main\_\_':

      main()

**Вывод программы:**

Matrix A:

12 -3 -1 3

5 20 9 1

6 -3 -21 -7

8 -7 3 -27

Vector b:

-31 90 119 71

Epsilon:

0.01

X:

[8.353977263286794e-05, 7.0003965988518555, -5.00015606683062, -5.000318070931773]

Number of iterations:

13

1. метод Зейделя

import sys

def check\_iter(A):

   n = len(A[0])

   for i in range(n):

      suma = sum(abs(x) for x in A[i]) - abs(A[i][i])

      if abs(A[i][i])<= suma:

         return False

   return True

def seidel(A,b, eps):

   n = len(b)

   x = [0,0]

   x[0] = [b[i] \* 1.0 / A[i][i] for i in range(n)]

   x[1] = [0 for \_ in range(n)]

   alnorm = max(sum(abs(1.0\*A[j][k]/A[j][j]) for k in range(n) if k != j) for j in range(n))

   cnorm = max(sum(abs(1.0\*A[j][k]/A[j][j]) for k in range(j+1,n)) for j in range(n))

   coef = (cnorm/(1-alnorm)) if alnorm < 1 else 1

   it = 0

   while True:

      it += 1

      for j in range(n):

         x[it%2][j] = (b[j] - sum(A[j][k]\*x[it%2][k] for k in range(j)) - sum(A[j][k]\*x[(it+1)%2][k] for k in range(j+1,n))) \* 1.0 / A[j][j]

      normx = coef \* sum(abs(x[it%2][k]-x[(it+1)%2][k]) for k in range(n))

      if normx <= eps:

         return x[it%2], it

def main():

   print("Matrix A:")

   A = []

   A.append([int(j) for j in input().strip().split(" ")])

   for i in range(1,len(A[0])) :

      A.append([int(j) for j in input().strip().split(" ")])

   print("Vector b:")

   b = [int(j) for j in input().strip().split(" ")]

   if not(check\_iter(A)):

      print("Seidel not applicable")

      return

   print("Epsilon:")

   eps = float(input())

   x, it = seidel(A,b,eps)

   print("X:")

   print(x)

   print("Number of iterations:")

   print(it)

if \_\_name\_\_ == '\_\_main\_\_':

    main()

**Вывод программы:**

Matrix A:

12 -3 -1 3

5 20 9 1

6 -3 -21 -7

8 -7 3 -27

Vector b:

-31 90 119 71

Epsilon:

0.01

X:

[-0.0006925549786691576, 7.0004626456398595, -5.000389421074938, -5.000368415649377]

Number of iterations:

8

**Вывод:**

В результате выполнения лабораторной работы были разработаны алгоритмы решения системы линейных уравнений методом простой итерации и методом Зейделя. Было проведено сравнение этих методов на примере решения системы линейных уравнений, и было выявлено, что метод Зейделя работает быстрее, чем метод простой итерации. Метод Зейделя требует большего объема вычислительных ресурсов, так как он работает с каждым уравнением системы несколько раз за одну итерацию. В целом, оба метода могут быть использованы для решения систем линейных уравнений в зависимости от конкретной задачи и доступных вычислительных ресурсов.

**Задание 1.4**

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.



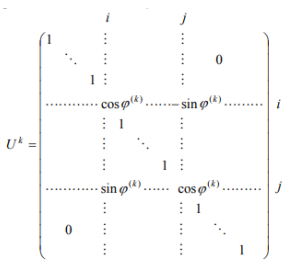
**Теоретические сведения**

Метод вращений Якоби применим только для симметрических матриц и решает полную проблему собственных значений собственных векторов таких матриц. Он основан на отыскании с помощью итерационных процедур матрицы 𝑈 в преобразовании подобия , а поскольку для симметрических матриц матрица преобразования подобия 𝑈 является ортогональной , то , где Λ – диагональная матрица с собственными значениями на главной диагонали.

Пусть дана симметрическая матрица 𝐴. Требуется для нее вычислить с точностью 𝜀 все собственные значения и соответствующие им собственные векторы. Алгоритм метода вращения, следующий:

Пусть известна матрица на k–й итерации, при этом для

1. Выбирается максимальный по модулю недиагональный элемент матрицы
2. Ставится задача найти такую ортогональную матрицу , чтобы в результате преобразования подобия произошло обнуление элемента матрицы . В качестве ортогональной матрицы выбирается матрица вращения, имеющая следующий вид:



Угол вращения определяется из условия

причем если

1. Строится матрица

В качестве критерия окончания итерационного процесса используется условие малости суммы квадратов вне диагональных элементов:

Координатными столбцами собственных векторов матрицы *А* в единичном базисе будут столбцы матрицы

**Код программы:**

import copy, math

def mult\_matrix(A, B):

   C = [[0.0 for col in range(len(B[0]))] for row in range(len(A))]

   for i in range(len(A)):

      for j in range(len(B[0])):

         for k in range(len(B)):

            C[i][j] += A[i][k]\*B[k][j]

   return C

def trans\_matrix(M):

   n = len(M)

   return [[ M[i][j] for i in range(n)] for j in range(n)]

def check\_symmetry(M):

   return M == trans\_matrix(M)

def givens\_rot(M, eps):

   n = len(M)

   A = copy.deepcopy(M)

   it = 0

   Ux = [[float(i == j) for j in range(n)] for i in range(n)]

   while True:

      it += 1

      a, i, j = 0.0, 0, 0

      for l in range(n):

         for m in range(l):

            if abs(A[l][m]) > a:

               a, i, j = abs(A[l][m]), l, m

      phi = 0.5 \* (math.atan2(2\*A[i][j], (A[i][i]-A[j][j]))) if A[i][i]-A[j][j] != 0 else math.pi / 4

      U = [[float(l == m) for m in range(n)] for l in range(n)]

      U[i][i] = math.cos(phi)

      U[j][j] = U[i][i]

      U[j][i] = math.sin(phi)

      U[i][j] = - U[j][i]

      A = mult\_matrix(trans\_matrix(U), mult\_matrix(A, U))

      Ux = mult\_matrix(Ux, U)

      t = math.sqrt(sum(sum(A[l][m]\*\*2 for m in range(l)) for l in range(n)))

      if t <= eps :

         return [A[l][l] for l in range(n)], trans\_matrix(Ux), it

def main():

   print("Matrix A:")

   A = []

   A.append([int(j) for j in input().strip().split(" ")])

   for i in range(1,len(A[0])) :

      A.append([int(j) for j in input().strip().split(" ")])

   if not(check\_symmetry(A)):

      print("not solvable")

      return

   print("Epsilon:")

   eps = float(input())

   Lambdas, Ux, it = givens\_rot(A,eps)

   print("Lambdas:")

   print(Lambdas)

   print("All X:")

   for elem in Ux:

      print(elem)

   print("Number of iterations:")

   print(it)

if \_\_name\_\_ == '\_\_main\_\_':

      main()

**Вывод программы:**

Matrix A:

4 7 -1

7 -9 -6

-1 -6 -4

Epsilon:

0.01

Lambdas:

[-14.623607766264342, -2.4483107542352425, 8.071918520499581]

All X:

[0.29309373687715623, -0.8440481813931533, -0.44908654944263804]

[0.4490987287685852, -0.2931282516513245, 0.8440297150588233]

[-0.8440417011154279, -0.4490640216837111, 0.2931469106223472]

Number of iterations:

6

**Вывод:**

В результате выполнения лабораторной работы был реализован метод вращений Якоби для поиска собственных значений и собственных функций симметричной матрицы. Этот метод является итеративным и позволяет добиться высокой точности результатов. Полученные результаты подтвердили правильность работы реализованного метода.

**Задание 1.5**

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.



**Теоретические сведения**

В основе 𝑄𝑅-алгоритма лежит представление матрицы в виде 𝐴 = 𝑄𝑅, где Q – ортогональная матрица , а *R* – верхняя треугольная. Такое разложение существует для любой квадратной матрицы. Одним из возможных подходов к построению 𝑄𝑅-разложения является использование преобразования Хаусхолдера, позволяющего обратить в нуль группу поддиагональных элементов столбца матрицы.

Преобразование Хаусхолдера осуществляется с использованием матрицы Хаусхолдера, имеющей следующий вид:

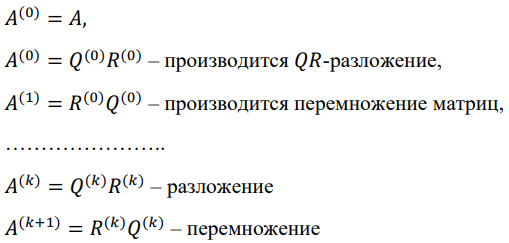
, где – произвольный ненулевой столбец

Рассмотрим случай, когда необходимо обратить в нуль все элементы какого-либо вектора кроме первого, т.е. построить матрицу Хаусхолдера такую, что

Тогда вектор 𝜈 определится следующим образом:

Применяя описанную процедуру с целью обнуления поддиагональных элементов каждого из столбцов исходной матрицы, можно за фиксированное число шагов получить ее QR – разложение.

Процедура 𝑄𝑅-разложения многократно используется в 𝑄𝑅-алгоритме вычисления собственных значений. Строится следующий итерационный процесс:



Таким образом, каждая итерация реализуется в два этапа. На первом этапе осуществляется разложение матрицы в произведение ортогональной и верхней треугольной матриц, а на втором – получение матрицы перемножаются в обратном порядке.

При отсутствии у матрицы кратных собственных значений последовательность сходится к верхней треугольной матрице (в случае, когда все собственные значения вещественны) или к верхней квазитреугольной матрице (если имеются комплексно-сопряженные пары собственных значений).

Таким образом, каждому вещественному собственному значению будет соответствовать столбец со стремящимися к нулю поддиагональными элементами и в качестве критерия сходимости итерационного процесса для таких собственных значений можно использовать следующее неравенство:

При этом соответствующее собственное значение принимается равным диагональному элементу данного столбца.

**Код программы:**

import copy, math, cmath

def mult\_matrix(A, B):

   C = [[0.0 for col in range(len(B[0]))] for row in range(len(A))]

   for i in range(len(A)):

      for j in range(len(B[0])):

         for k in range(len(B)):

            C[i][j] += A[i][k]\*B[k][j]

   return C

def trans\_matrix(M):

   n = len(M)

   return [[ M[i][j] for i in range(n)] for j in range(n)]

def norm(x):

    return math.sqrt(sum([x\_i\*\*2 for x\_i in x]))

def Q\_i(Q\_min, i, j, k):

    if i < k or j < k:

        return float(i == j)

    else:

        return Q\_min[i-k][j-k]

def QR\_decomp(A):

    n = len(A)

    R = copy.deepcopy(A)

    Q = [[0.0] \* n for i in range(n)]

    for k in range(n-1):

        I = [[float(i == j) for i in range(n)] for j in range(n)]

        x = [row[k] for row in R[k:]]

        e = [row[k] for row in I[k:]]

        alpha = -((x[0] > 0) - (x[0] < 0))\* norm(x)

        u = list(map(lambda p,q: p + alpha \* q, x, e))

        norm\_u = norm(u)

        v = list(map(lambda p: p/norm\_u, u))

        Q\_min = [ [float(i==j) - 2.0 \* v[i] \* v[j] for i in range(n-k)] for j in range(n-k) ]

        Q\_t = [[ Q\_i(Q\_min,i,j,k) for i in range(n)] for j in range(n)]

        if k == 0:

            Q = Q\_t

            R = mult\_matrix(Q\_t,A)

        else:

            Q = mult\_matrix(Q\_t,Q)

            R = mult\_matrix(Q\_t,R)

    return trans\_matrix(Q), R

def QR\_solve(A, eps):

   n = len(A)

   M = copy.deepcopy(A)

   it = 0

   while True:

      it += 1

      Q, R = QR\_decomp(M)

      M = mult\_matrix(R, Q)

      La = [M[l][l] for l in range(n)]

      t = True

      i = -1

      while i < n-2:

         i += 1

         if math.sqrt(sum(M[i][j]\*\*2 for j in range(i+1, n))) > eps:

            if math.sqrt(sum(M[i][j]\*\*2 for j in range(i+2, n))) <= eps:

               a = 1

               b = -(M[i][i]+M[i+1][i+1])

               c = M[i][i]\*M[i+1][i+1] - M[i][i+1]\*M[i+1][i]

               disc = b\*\*2 - 4 \* a \* c

               l1 = (-b + cmath.sqrt(disc))/ (2 \* a)

               l2 = (-b - cmath.sqrt(disc))/ (2 \* a)

               La[i] = l1

               La[i+1] = l2

               i += 1

            else:

               t = False

               break

      if t:

         return La, it

def main():

   print("Matrix A:")

   A = []

   A.append([int(j) for j in input().strip().split(" ")])

   for i in range(1,len(A[0])) :

      A.append([int(j) for j in input().strip().split(" ")])

   print("Epsilon:")

   eps = float(input())

   Lambdas, it = QR\_solve(A,eps)

   print("Lambdas:")

   print(Lambdas)

   print("Number of iterations:")

   print(it)

if \_\_name\_\_ == '\_\_main\_\_':

      main()

**Вывод программы:**

Matrix A:

-5 -8 4

4 2 6

-2 5 -6

Epsilon:

0.01

Lambdas:

[(-6.223764161951541+4.7093487723523655j), (-6.223764161951541-4.7093487723523655j), 3.4475283239029517]

Number of iterations:

531

**Вывод:**

В результате выполнения лабораторной работы мной были изучены основы метода QR-разложения для поиска собственных значений матрицы. Была реализована программа, основанная на преобразовании Хаусхолдера, которая позволяет получить QR-разложение матрицы и на его основе найти собственные значения. Также был проведен анализ эффективности метода в зависимости от размерности матрицы, что позволило сделать вывод о применимости метода для решения задач большой размерности.