Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №1
Задание №5 по курсу «Численные методы»

Студент: С. М. Власова
Преподаватель: И. Э. Иванов
Группа: М8О-306Б
Дата:
Оценка:
Подпись:

Москва, 2020

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.
Вариант: 12

$$\begin{pmatrix} 5 & -1 & -2 \\ -4 & 3 & -3 \\ -2 & -1 & 1 \end{pmatrix}$$

1 Описание метода решения

При решении полной проблемы собственных значений для несимметричных матриц эффективным является подход, основанный на приведении матриц к подобным, имеющим треугольный или квазитреугольный вид. Одним из наиболее распространенных методов этого класса является **QR-алгоритм**, позволяющий находить как *вещественные*, так и *комплексные* собственные значения.

В основе $QR-$алгоритма лежит представление матрицы в виде $A = Q \cdot R$,
где Q — ортогональная матрица ($Q^{-1} = Q^T$), а R — верхняя треугольная матрица. 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:

$$H = E - \frac{2}{\nu^T \cdot \nu} \cdot \nu \cdot \nu^T,$$

где ν — произвольный ненулевой вектор-столбец, E — единичная матрица, $\nu \cdot \nu^T$ — квадратная матрица того же размера.

Легко убедиться, что любая матрица такого вида является симметричной и ортогональной. При этом произвол в выборе вектора ν дает возможность построить матрицу, отвечающую некоторым дополнительным требованиям.
Рассмотрим случай, когда необходимо обратить в нуль все элементы какого-либо вектора кроме первого, т.е. построить матрицу Хаусхолдера такую, что

$$\widetilde{b} = H \cdot b, \quad b = (b_1, b_2, \ldots, n)^T, \quad \widetilde{b} = (\widetilde{b_1}, 0, \ldots, 0)^T.$$

Тогда вектор ν определится следующим образом:

$$\nu = b + sign(b_1) \cdot ||b||_2 \cdot e_1,$$

где $||b||_2 = (\sum_i b_i^2)^{\frac{1}{2}}$ — евклидова норма вектора, $e_1 = (1, 0, \ldots, 0)^T$.

Применяя описанную процедуру с целью обнуления поддиагональных элементов каждого из столбцов исходной матрицы, можно за фиксированное число шагов получить ее $QR-$разложение.
Рассмотрим подробнее реализацию данного процесса. Положим $A_0 = A$ и построим

преобразование Хаусхолдера H_1 ($A_1 = H_1 \cdot A_0$), переводящее матрицу A_0 в матрицу A_1 с нулевыми элементами первого столбца под главной диагональю:

$$A_0 = \begin{pmatrix} a_{11}^0 & a_{12}^0 & \dots & a_{1n}^0 \\ a_{21}^0 & a_{22}^0 & \dots & a_{2n}^0 \\ \dots & \dots & \dots & \dots \\ a_{n1}^0 & a_{n2}^0 & \dots & a_{nn}^0 \end{pmatrix} \underset{H_1}{\longrightarrow} A_1 = \begin{pmatrix} a_{11}^1 & a_{12}^1 & \dots & a_{1n}^1 \\ 0 & a_{22}^1 & \dots & a_{2n}^1 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & a_{nn}^1 \end{pmatrix}$$

Ясно, что матрица Хаусхолдера H_1 должна определяться по первому столбцу матрицы A_0, т.е в качестве вектора b в формуле ν берется вектор $(a_{11}^0, a_{a1}^0, \dots, n1^0)^T$.
Тогда компоненты вектора ν вычисляются следующим образом:

$$\nu_1^1 = a_{11}^0 + sign(a_{11}^0) \cdot (\sum_{j=1}^{n} (a_{j1}^0)^2)^{\frac{1}{2}}$$

$$\nu_i^1 = a_{i1}^0, \quad i = \overline{2, n}.$$

Матрица Хаусхолдера H_1 вычисляется согласна формуле:

$$H_1 = E - 2 \cdot \frac{\nu^1 \cdot \nu^{1^T}}{\nu^{1^T} \cdot \nu}.$$

На следующем, втором, шаге рассматриваемого процесса строится преобразование Хаусхолдера H_2 ($A_2 = H_2 \cdot A_1$), обнуляющее расположенные ниже главной диагонали элементы второго столбца матрицы A_1. Взяв в качестве вектора b вектор $(a_{22}^1, a_{32}^1, \dots, a_{n2}^1)^T$ размерности $n-1$, получим следующие выражения для компонентов вектора ν :

$$\nu_1^2 = 0,$$

$$\nu_2^2 = a_{22}^1 + sign(a_{22}^1) \cdot (\sum_{j=2}^{n} (a_{j2}^1)^2)^{\frac{1}{2}}$$

$$\nu_i^2 = a_{i2}^1, \quad i = \overline{3, n}.$$

Повторяя процесс $n-1$ раз, получим искомое разложение $A = Q \cdot R$, где
$Q = (H_{n-1} \cdot H_{n-2} \cdot \ldots \cdot H_0)^T = H_1 \cdot H_2 \cdot \ldots \cdot H_{n-1}, \quad R = A_{n-1}$.

Следует отметить определенное сходство рассматриваемого процесса с алгоритмом Гаусса. Отличие заключается в том, что здесь обнуление поддиагональных элементов соответствующего столбца осуществляется с использованием ортогонального преобразования. Процедура $QR-$разложения многократно используется в $QR-$алгоритме вычисления собственных значений.
Строится следующий итерационный процесс.

$A^{(0)} = A$,

$A^{(0)} = Q^{(0)} \cdot R^{(0)}$ — производится $QR-$разложение,

$A^{(1)} = R^{(0)} \cdot Q^{(0)}$ — производится перемножение матриц,

. . . . . .

$A^{(k)} = Q^{(k)} \cdot R^{(k)}$ — разложение,

$A^{(k+1)} = R^{(k)} \cdot Q^{(k)}$ — перемножение.

Таким образом, каждая итерация реализуется в два этапа. На первом этапе осуществляется разложение матрицы $A^{(k)}$ в произведение ортогональной $Q^{(k)}$ и верхней треугольной $R^{(k)}$ матриц, а на втором — полученные матрицы перемножаются в обратном порядке.
Нетрудно показать подобие матриц $A^{(k+1)}$ и $A^{(k)}$. Действительно, учитывая ортогональность $Q^{(k)}$ $(Q^{(k)T} \cdot Q^{(k)} = E)$, можно записать:

$$A^{(k+1)} = R^{(k)} \cdot Q^{(k)} = Q^{(k)T} \cdot Q^{(k)} \cdot R^{(k)} \cdot Q^{(k)} = Q^{(k)T} \cdot A^{(k)} \cdot Q^{(k)}.$$

Аналогично можно показать, что любая из матриц $A^{(k)}$ ортогонально подобна матрице A.
При отсутствии у матрицы кратных собственных значений последовательность $A^{(k)}$ 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:

$$(\sum_{l=m+1}^{n} (a_{lm}^{(k)})^2)^{\frac{1}{2}} \leq \varepsilon.$$

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

Каждой комплексно-сопряженной паре соответствует диагональный блок размерностью 2x2, т.е. матрица $A^{(k)}$ имеет блочно-диагональную структуру. Принципиально то, что элементы этих блоков изменяются от итерации к итерации без видимой закономерности, в то время как комплексно-сопряженные собственные значения, определяемые каждым блоком, имеют тенденцию к сходимости. Это обстоятельство необходимо учитывать при формировании критерия выхода из итерационного процесса. Если в ходе итераций прослеживается комплексно-сопряженная пара собственных значений, соответствующая блоку, образуемому элементами $j-$го и $(j+1)-$го столбцов

$a_{jj}^{(k)}, a_{jj+1}^{(k)}, a_{j+1j}^{(k)}, a_{j+1j+1}^{(k)}$, то, несмотря на значительное изменение в ходе итераций самих этих элементов, собственные значения, соответствующие данному блоку и определяемые из решения квадратного уравнения $(a_{jj}^{(k)} - \lambda^{(k)}) \cdot (a_{j+1j+1}^{(k)} - \lambda^{(k)}) = a_{jj+1}^{(k)} \cdot a_{j+1j}^{(k)}$, начиная с некоторого k, отличаются незначительно.

В качестве критерия окончания итераций для таких блоков может быть использовано следующее условие

$$|\lambda^{(k)} - \lambda^{(k-1)}| \leq \varepsilon.$$

2 Протокол

Входные данные я храню в файле *data5*: первая строка — размерность матрицы, на следующих строках — матрица и заданная точность.
Выходные данные я записываю в файл *res5*.
Скриншот консоли:
$\varepsilon = 0.1$.

```
(base) vlasochka@vlasochka-VPCSB11FX:~/Документы/ЧМ$ g++ 1.5.cpp -o 1.5
(base) vlasochka@vlasochka-VPCSB11FX:~/Документы/ЧМ$ cat data5
3
5 -1 -2
-4 3 -3
-2 -1 1
0.1
(base) vlasochka@vlasochka-VPCSB11FX:~/Документы/ЧМ$ ./1.5 < data5 > res5
(base) vlasochka@vlasochka-VPCSB11FX:~/Документы/ЧМ$ cat res5
Собственные значения матрицы A:
lyambda_0 = 6.42504
lyambda_1 = 3.8026
lyambda_2 = -1.22764
(base) vlasochka@vlasochka-VPCSB11FX:~/Документы/ЧМ$
```

Алгоритм нашел собственные значения матрицы с точностью $\varepsilon = 0.1$.

$$\lambda_1 = 6.42504, \quad \lambda_2 = 3.8026, \quad \lambda_3 = -1.22764.$$

Найдем собственные значения, уменьшив значение оценки погрешности.
Скриншот консоли:
$\varepsilon = 0.01$.

```
(base) vlasochka@vlasochka-VPCSB11FX:~/Документы/ЧМ$ cat data5
3
5 -1 -2
-4 3 -3
-2 -1 1
0.01
(base) vlasochka@vlasochka-VPCSB11FX:~/Документы/ЧМ$ ./1.5 < data5 > res5
(base) vlasochka@vlasochka-VPCSB11FX:~/Документы/ЧМ$ cat res5
Собственные значения матрицы A:
lyambda_0 = 6.38992
lyambda_1 = 3.83398
lyambda_2 = -1.2239
(base) vlasochka@vlasochka-VPCSB11FX:~/Документы/ЧМ$
```

Алгоритм нашел собственные значения матрицы с точностью $\varepsilon = 0.01$.

$$\lambda_1 = 6.38992, \quad \lambda_2 = 3.83398, \quad \lambda_3 = -1.2239.$$

Скриншот консоли:
$\varepsilon = 0.001$.

```
(base) vlasochka@vlasochka-VPCSB11FX:~/Документы/ЧМ$ cat data5
3
5 -1 -2
-4 3 -3
-2 -1 1
0.001
(base) vlasochka@vlasochka-VPCSB11FX:~/Документы/ЧМ$ ./1.5 < data5 > res5
(base) vlasochka@vlasochka-VPCSB11FX:~/Документы/ЧМ$ cat res5
Собственные значения матрицы A:
lyambda_0 = 6.38486
lyambda_1 = 3.839
lyambda_2 = -1.22386
(base) vlasochka@vlasochka-VPCSB11FX:~/Документы/ЧМ$
```

Алгоритм нашел собственные значения матрицы с точностью $\varepsilon = 0.001$.

$$\lambda_1 = 6.38486, \quad \lambda_2 = 3.839, \quad \lambda_3 = -1.22386.$$

3 Исходный код

Листинг 1: QR-метод

```
#include <iostream>
#include <vector>
#include <cmath>

typedef struct Complex{
    double real;
    double imag;
} Complex;

void PrintMatrix(std::vector<std::vector<double>>& Matrix)
{
    for(int i = 0; i < Matrix.size(); i++){
        for(int j = 0; j < Matrix[0].size(); j++)
            std::cout << Matrix[i][j] << " ";
        std::cout << std::endl;
    }
}

void MultMatrix(std::vector<std::vector<double>>& F, std::vector<std::vector<double>>& S)
{
    int n = F.size();
    int m = S[0].size();
    int l = S.size();
    double a = 0;
    std::vector<double> H(n);

    for(int i = 0; i < m; i++)
    {
        for(int j = 0; j < n; j++)
        {
            for(int k = 0; k < l; k++)
                a += F[j][k]*S[k][i];
            H[j] = a;
            a = 0;
        }
        for(int j = 0; j < n; j++)
            S[j][i] = H[j];
    }
}
```

```
40
41 void MultMatrix(std::vector<std::vector<double>>& F, std::vector<std::vector<double>>& S,
       std::vector<std::vector<double>>& R)
42 {
43     int n = F.size();
44     int m = S.size();
45     int l = S[0].size();
46     int k = R.size();
47     int h = R[0].size();
48     double a = 0;
49
50     if(n - k > 0)
51         for(int i = 0; i < n - k; i++)
52             R.insert(R.end(), std::vector<double>(l, 0));
53     if(l - h > 0)
54         for(int i = 0; i < n; i++)
55             for(int j = 0; j < l - h; j++)
56                 R[i].insert(R[i].end(), 0);
57     for(int i = 0; i < l; i++)
58         for(int j = 0; j < n; j++)
59         {
60             for(int k = 0; k < m; k++)
61                 a += F[j][k]*S[k][i];
62             R[j][i] = a;
63             a = 0;
64         }
65     if(k - n > 0)
66         R.erase(R.end()- k + n, R.end());
67     if(h - l > 0)
68         for(int i = 0; i < n; i++)
69             R[i].erase(R[i].end() - h + l, R[i].end());
70 }
71
72 int Sign(double a)
73 {
74     return
75         (a < 0) ? -1
76         : (a > 0) ? 1
77         : 0;
78 }
79
80 void Transporant(std::vector<std::vector<double>>& M)
81 {
82     double h = 0;
```

```
83      int n = M.size();
84      for(int i = 0; i < n; i++)
85          for(int j = i + 1; j < n; j++)
86          {
87              h = M[i][j];
88              M[i][j] = M[j][i];
89              M[j][i] = h;
90          }
91  }
92
93  void HouseHolder(std::vector<std::vector<double>>& A, std::vector<std::vector<double>>& Q
        )
94  {
95      int n = A.size();
96      double sum = 0;
97      double kaf = 0;
98      std::vector<std::vector<double>> Nju(n, std::vector<double>(1, 0));
99      std::vector<std::vector<double>> NjuTransp(1, std::vector<double>(n, 0));
100     std::vector<std::vector<double>> H(n, std::vector<double>(n, 0));
101     std::vector<std::vector<double>> R(1, std::vector<double>(1, 0));
102
103     for(int i = 0; i < n - 1; i ++)
104     {
105         for(int j = 0; j < i; j++)
106         {
107             Nju[j][0] = 0;
108             NjuTransp[0][j] = 0;
109         }
110         for(int j = i; j < n; j++)
111         {
112             Nju[j][0] = A[j][i];
113             NjuTransp[0][j] = Nju[j][0];
114             sum += pow(A[j][i], 2);
115         }
116         Nju[i][0] += Sign(Nju[i][0])*pow(sum, 0.5);
117         NjuTransp[0][i] = Nju[i][0];
118         MultMatrix(NjuTransp, Nju, R);
119         kaf = -2/R[0][0];
120         MultMatrix(Nju, NjuTransp, R);
121         for(int i = 0; i < n; i++)
122             for(int j = 0; j < n; j++)
123                 if(i == j)
124                     H[i][j] = 1 + R[i][j]*kaf;
125                 else
```

```
126                         H[i][j] = R[i][j]*kaf;
127             MultMatrix(H, A);
128             if(i == 0)
129                 Q = H;
130             else
131                 MultMatrix(Q, H);
132             Q = H;
133             sum = 0;
134         }
135         H = A;
136         MultMatrix(H, Q, A);
137     }
138
139     void GetComplexSV(std::vector<std::vector<double>>& A, std::vector<Complex>& Lyambda,
        int i)
140     {
141         double Disc = pow(A[i+1][i+1] + A[i][i], 2) - 4*(A[i][i]*A[i+1][i+1] - A[i][i+1]*A[i+1][i]);
142         Lyambda[i].real = (A[i+1][i+1] + A[i][i])/2;
143         Lyambda[i].imag = pow(fabs(Disc), 0.5)/2;
144         Lyambda[i+1].real = Lyambda[i].real;
145         Lyambda[i+1].imag = -Lyambda[i].imag;
146     }
147
148     int main()
149     {
150         int n, it_count = 0;
151         double eps, kr = 1;
152         std::cin >> n;
153         std::vector<std::vector<double>> A(n, std::vector<double>(n));
154         std::vector<std::vector<double>> Q(n, std::vector<double>(n));
155         std::vector<Complex> Lyambda(n, {0, 0});
156
157         for( int i = 0; i < n; i++)
158             for( int j = 0; j < n; j++)
159                 std::cin >> A[i][j];
160         std::cin >> eps;
161         while(kr > eps)
162         {
163             HouseHolder(A, Q);
164             for(int i = 0; i < n; i++ )
165             {
166                 kr = 0;
167                 for(int j = i + 1; j < n; j++)
168                     kr += pow(A[j][i], 2);
```

```
169                 kr = pow(kr, 0.5);
170                 if(kr <= eps)
171                     Lyambda[i].real = A[i][i];
172                 else if( i == 0)
173                     break;
174                 else if(i != n - 1)
175                 {
176                     double real = Lyambda[i].real;
177                     double img = Lyambda[i].imag;
178                     GetComplexSV(A, Lyambda, i);
179                     kr = pow(pow(real - Lyambda[i].real, 2) + pow(img - Lyambda[i].imag, 2), 0.5)
                            ;
180                     i++;
181                     if(kr > eps)
182                         break;
183                 }
184             }
185             it_count ++;
186         }
187         std::cout << "Eigenvalues of matrix A:" << std::endl;
188         for(int i = 0; i < n; i++)
189             if(Lyambda[i].imag == 0)
190                 std::cout << "lyambda_" << i << " = " << Lyambda[i].real << std::endl;
191             else
192                 std::cout << "lyambda_" << i << " = " << Lyambda[i].real << " + " <<
                        Lyambda[i].imag << "i" << std::endl;
193         return 0;
194 }
```

4 Выводы

Выполнив пятое задание первой лабораторной работы, я познакомилась с QR-алгоритмом. В отличие от метода вращений Якоби, этот алгоритм решает полную проблему собственных значений для несимметричных матриц. Он находит как вещественные, так и комплексные собственные значения. Алгоритм основан на приведении исходной матрицы к подобной, треугольной (в случае с вещественными собственными значениями) или квазитруегольной (когда среди собственных значений есть комплексные). Построение QR-разложения можно осуществить разными способами, в данной лабораторной я реализую разложение с помощью преобразования Хаусхолдера. На каждой итерации мы ищем разложение исходной матрицы и путем перемножения находим новую матрицу, тем самым приближая ее к подобной. Существенным недостатком рассмотренного выше алгоритма является большое число операций (пропорционально n^3, где n — размерность матрицы), необходимое для $QR-$ факторизации матрицы на каждой итерации.