Московский авиационный институт (национальный исследовательский университет)

Институт №8 «Информационные технологии и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

Лабораторные работы по курсу «Численные методы»

Студент: М. А. Субботина
Преподаватель: Д. Е. Пивоваров
Группа: М8О-303Б-21

Дата:
Оценка:
Подпись:

Москва, 2024

1.5 QR – разложение матриц

1 Постановка задачи

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: 22

$$\begin{pmatrix} -1 & 8 & 5 \\ 8 & -4 & 4 \\ 2 & 9 & -2 \end{pmatrix}$$

2 Результаты работы

Рис. 1: Вывод программы в консоли

3 Исходный код

```
#include <vector>
#include <iostream>
#include <locale.h>
#include "matrix.h"
#include <cmath>
#include <algorithm>

using namespace std;

#define value_pair pair<pair<double, double>, pair<double, double>>


int size_init() {
    int size;
    cin >> size;
    return size;
}

void matrix_init(Matrix& A, int size) {
    A = Matrix(size, size);
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            cin >> A[i][j];
        }
    }
}



void print_vector_x(const vector<pair<double, double>>& x) {
    for (unsigned i = 0; i < x.size(); ++i) {
        cout << 'l' << i + 1 << " = " << x[i].first;
        if (x[i].second) {
            if (x[i].second > 0) {
                cout << " + ";
            }
            else {
                cout << " - ";
            }
            cout << abs(x[i].second) << "i";
        }
        cout << endl;
    }
    cout << endl;
}


```

```
48 double mult_1xn_nx1_vecs(const vector<double>& left, const vector<double>& right) {
49     double ans = 0.0;
50     if (left.size() != right.size()) {
51         throw "Wrong sizes of vectors!";
52     }
53     for (unsigned i = 0; i < left.size(); ++i) {
54         ans += right[i] * left[i];
55     }
56     return ans;
57 }
58
59 Matrix mult_nx1_1xn_vecs(const vector<double>& left, const vector<double>& right) {
60     Matrix ans(left.size(), right.size());
61     for (int i = 0; i < ans.get_n(); ++i) {
62         for (int j = 0; j < ans.get_m(); ++j) {
63             ans[i][j] = left[i] * right[j];
64         }
65     }
66     return ans;
67 }
68
69 double sign(double num) {
70     if (!num) {
71         return 0.0;
72     }
73     return num > 0 ? 1.0 : -1.0;
74 }
75
76 void solve_eq(double a, double b, double c, pair<pair<double, double>, pair<double,
       double>>& ans) {
77     double D = b * b - 4.0 * a * c;
78     if (D >= 0.0) {
79         ans.first.first = (-b + sqrt(D)) / (2 * a);
80         ans.first.second = 0.0;
81         ans.second.first = (-b - sqrt(D)) / (2 * a);
82         ans.second.second = 0.0;
83     }
84     else {
85         ans.first.first = -b / (2 * a);
86         ans.first.second = sqrt(-D) / (2 * a);
87         ans.second.first = -b / (2 * a);
88         ans.second.second = -sqrt(-D) / (2 * a);;
89     }
90 }
91
92 double complex_check(const value_pair& last, const value_pair& cur) {
93     pair<double, double> r1, r2;
94     r1.first = cur.first.first - last.first.first;
95     r1.second = cur.first.second - last.first.second;
```

```
96
97      r2.first = cur.second.first - last.second.first;
98      r2.second = cur.second.second - last.second.second;
99      return max(sqrt(r1.first * r1.first + r1.second * r1.second), sqrt(r2.first * r2.
            first + r2.second * r2.second));
100 }
101
102 void QRseparate_method(const Matrix& A, Matrix& Q, Matrix& R) {
103     Matrix E(A.get_n(), A.get_m());
104     E.make_ones();
105     Q = E;
106     R = A;
107
108     for (int j = 0; j < R.get_m() - 1; ++j) {
109         vector<double> v(R.get_n(), 0.0);
110         double norm = 0.0;
111
112         v[j] = R[j][j];
113         for (int i = j; i < R.get_n(); ++i) {
114             norm += R[i][j] * R[i][j];
115         }
116         norm = sqrt(norm);
117         v[j] += sign(R[j][j]) * norm;
118
119         for (int i = j + 1; i < R.get_n(); ++i) {
120             v[i] = R[i][j];
121         }
122
123         Matrix H = E - (2.0 / mult_1xn_nx1_vecs(v, v)) * mult_nx1_1xn_vecs(v, v);
124
125         Q = Q * H;
126         R = H * R;
127     }
128 }
129
130 int QRmethod_values(const Matrix& A, vector<pair<double, double>>& x, double alfa) {
131     Matrix Q, R, A_k = A;
132     x.resize(A_k.get_m());
133     int itter = 0;
134     double check;
135     value_pair curr;
136     bool flag = true;
137
138     for (itter = 0; flag; ++itter) {
139         QRseparate_method(A_k, Q, R);
140         A_k = R * Q;
141
142         flag = false;
143         for (int j = 0; j < A_k.get_m(); ++j) {
```

```
            check = 0.0;
            for (int i = j + 1; i < A_k.get_n(); ++i) {
                check += A_k[i][j] * A_k[i][j];
            }
            check = sqrt(check);

            if (check > alfa) {
                solve_eq(1.0, -A_k[j][j] - A_k[j + 1][j + 1], A_k[j][j] * A_k[j + 1][j +
                     1] - A_k[j + 1][j] * A_k[j][j + 1], curr);
                if (complex_check(curr, value_pair(x[j], x[j + 1])) > alfa) {
                    flag = true;
                }
                x[j] = curr.first;
                x[j + 1] = curr.second;
                ++j;
            }
            else {
                x[j].first = A_k[j][j];
                x[j].second = 0.0;
            }
        }
    }
    return itter;
}

int main() {
    setlocale(0, "");
    Matrix A;
    vector<pair<double, double>> x;
    double accuracy = 0.01;
    cout << "  :";
    int size = size_init();
    cout << " : \n";
    matrix_init(A, size);
    cout << "  : \n";
    cin >> accuracy;

    cout << "QR :" << endl;
    int itter = QRmethod_values(A, x, accuracy);
    cout << " : \n";
    print_vector_x(x);
    cout << " : " << itter << endl;

    return 0;
}
```

```
#include "matrix.h"

const Matrix& Matrix::operator=(const Matrix& right) {
    _matrix = right._matrix;
```

```
5       n_size = right.n_size;
6       m_size = right.m_size;
7       return *this;
8   }
9
10
11  vector<double>& Matrix::operator[](const int index) {
12      return _matrix[index];
13  }
14
15  const vector<double>& Matrix::operator[](const int index) const {
16      return _matrix[index];
17  }
18
19
20  std::ostream& operator<<(std::ostream& os, const Matrix& matrix) {
21      for (int i = 0; i < matrix.n_size; ++i) {
22          os << endl;
23          os.width(8);
24          os << matrix[i][0];
25          for (int j = 1; j < matrix.m_size; ++j) {
26              os << '\t';
27              os.width(8);
28              os << matrix[i][j];
29          }
30      }
31      os << endl;
32      return os;
33  }
34
35
36  const Matrix operator+(const Matrix& left, const Matrix& right) {
37      if (left.n_size != right.n_size || left.m_size != right.m_size) {
38          throw "   !";
39      }
40      Matrix ans(left.n_size, left.m_size);
41      for (int i = 0; i < ans.n_size; ++i) {
42          for (int j = 0; j < ans.m_size; ++j) {
43              ans[i][j] = left._matrix[i][j] + right._matrix[i][j];
44          }
45      }
46      return ans;
47  }
48
49  const Matrix operator-(const Matrix& left, const Matrix& right) {
50      if (left.n_size != right.n_size || left.m_size != right.m_size) {
51          throw "   !";
52      }
53      Matrix ans(left.n_size, left.m_size);
```

```
54      for (int i = 0; i < ans.n_size; ++i) {
55          for (int j = 0; j < ans.m_size; ++j) {
56              ans[i][j] = left._matrix[i][j] - right._matrix[i][j];
57          }
58      }
59      return ans;
60  }
61
62  const Matrix operator*(double left, const Matrix& right) {
63      Matrix ans = right;
64      for (int i = 0; i < ans.n_size; ++i) {
65          for (int j = 0; j < ans.m_size; ++j) {
66              ans[i][j] *= left;
67          }
68      }
69      return ans;
70  }
71
72  const Matrix operator*(const Matrix& left, double right) {
73      return right * left;
74  }
75
76
77
78
79  const Matrix operator*(const Matrix& left, const Matrix& right) {
80      if (left.m_size != right.n_size) {
81          throw "   !";
82      }
83      Matrix ans(left.n_size, right.m_size);
84      for (int i = 0; i < ans.n_size; ++i) {
85          for (int j = 0; j < ans.m_size; ++j) {
86              for (int k = 0; k < left.m_size; ++k) {
87                  ans[i][j] += left._matrix[i][k] * right._matrix[k][j];
88              }
89          }
90      }
91      return ans;
92  }
93
94  const vector<double> operator*(const Matrix& left, const vector<double>& right) {
95      if (left.m_size != (int)right.size()) {
96          throw "   !";
97      }
98      vector<double> ans(left.n_size, 0.0);
99      for (int i = 0; i < left.n_size; ++i) {
100         for (int j = 0; j < left.m_size; ++j) {
101             ans[i] += left._matrix[i][j] * right[j];
102         }
```

```
103     }
104     return ans;
105 }
106
107
108
109 int Matrix::get_m() const {
110     return m_size;
111 }
112
113 int Matrix::get_n() const {
114     return n_size;
115 }
116
117 void Matrix::make_ones() {
118     if (!is_quadratic()) {
119         throw "  ";
120     }
121     _matrix.assign(n_size, vector<double>(m_size, 0.0));
122     for (int i = 0; i < n_size; ++i) {
123         _matrix[i][i] = 1.0;
124     }
125 }
126
127 void Matrix::transpose() {
128     vector<vector<double>> temp(m_size, vector<double>(n_size));
129     for (int i = 0; i < n_size; ++i) {
130         for (int j = 0; j < m_size; ++j) {
131             temp[j][i] = _matrix[i][j];
132         }
133     }
134     swap(n_size, m_size);
135     _matrix.swap(temp);
136 }
137
138 Matrix::Matrix() {
139     _matrix.assign(1, vector<double>(1, 0));
140     n_size = m_size = 1;
141 }
142
143 Matrix::Matrix(int n, int m) {
144     _matrix.assign(n, vector<double>(m, 0));
145     n_size = n;
146     m_size = m;
147 }
148
149
150 bool Matrix::is_quadratic() const {
151     return n_size == m_size;
```

```
152 }
153
154
155 bool Matrix::is_three_diagonal() const {
156     if (!is_quadratic()) {
157         return false;
158     }
159     for (int i = 0; i < n_size; ++i) {
160         for (int j = 0; j < m_size; ++j) {
161             if ((abs(i - j) > 1) && _matrix[i][j]) {
162                 return false;
163             }
164         }
165     }
166     return true;
167 }
168
169 bool Matrix::is_simmetric() const {
170     if (!is_quadratic()) {
171         return false;
172     }
173     for (int i = 0; i < n_size; ++i) {
174         for (int j = i + 1; j < m_size; ++j) {
175             if (_matrix[i][j] != _matrix[j][i]) {
176                 return false;
177             }
178         }
179     }
180     return true;
181 }
182
183 double Matrix::get_norm() const {
184     double max = 0.0;
185     for (int i = 0; i < n_size; ++i) {
186         double ans = 0.0;
187         for (int j = 0; j < m_size; ++j) {
188             ans += abs(_matrix[i][j]);
189         }
190         max = max > ans ? max : ans;
191     }
192     return max;
193 }
194
195 double Matrix::get_upper_norm() const {
196     double max = 0.0;
197     for (int i = 0; i < n_size; ++i) {
198         double ans = 0.0;
199         for (int j = 0; j <= i; ++j) {
200             ans += abs(_matrix[i][j]);
```

```
        }
        max = max > ans ? max : ans;
    }
    return max;
}
```

```
#pragma once
#ifndef MATRIX_H
#define MATRIX_H

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


using namespace std;

class Matrix {
public:
    Matrix();
    Matrix(int n, int m);

    void make_ones();
    void transpose();

    vector<double>& operator[](const int index);
    const vector<double>& operator[](const int index) const;

    friend const Matrix operator+(const Matrix& left, const Matrix& right);
    friend const Matrix operator-(const Matrix& left, const Matrix& right);

    friend const Matrix operator*(const Matrix& left, const Matrix& right);
    friend const vector<double> operator*(const Matrix& left, const vector<double>&
        right);
    friend const Matrix operator*(const Matrix& left, double right);
    friend const Matrix operator*(double left, const Matrix& right);

    const Matrix& operator=(const Matrix& right);

    friend std::ostream& operator<<(std::ostream& os, const Matrix& matrix);

    double get_norm() const;
    double get_upper_norm() const;

    int get_n() const;
    int get_m() const;

    bool is_three_diagonal() const;
    bool is_simmetric() const;

```

```
44      bool is_quadratic() const;
45
46  private:
47      vector<vector<double>> _matrix;
48      int n_size;
49      int m_size;
50  };
51
52  #endif
```