Министерство образования и науки Российской Федерации
Санкт-Петербургский политехнический университет Петра Великого
—
Институт компьютерных наук и технологий
Высшая школа программной инженерии

Лабораторная работа №2
по дисциплине «Вычислительная математика»

Выполнил
Студент группы 5130904/20004 Шелковников Д.С.

Преподаватель Устинов С.М.

Санкт-Петербург
2024

Оглавление

Задание

Семейство линейных систем, представленных в следующем виде зависит от p:

$$\begin{pmatrix} p-29 & 6 & -6 & -4 & -3 & -8 & -5 & 5 \\ 6 & -13 & -3 & 5 & 4 & 3 & 1 & 7 \\ 5 & -5 & -1 & 7 & 2 & 0 & 7 & 1 \\ 5 & -5 & 5 & 6 & 4 & -7 & 4 & 0 \\ 4 & 4 & 7 & -4 & 9 & -8 & -8 & -4 \\ -4 & 5 & -4 & 1 & 0 & 12 & 0 & 6 \\ -3 & -2 & -4 & 2 & -8 & -3 & 16 & 4 \\ 7 & 5 & 0 & 2 & 0 & -6 & 8 & -12 \end{pmatrix} \begin{pmatrix} x^1 \\ x^2 \\ x^3 \\ x^4 \\ x^5 \\ x^6 \\ x^7 \\ x^8 \end{pmatrix} = \begin{pmatrix} 4p-175 \\ 133 \\ 110 \\ 112 \\ 17 \\ 32 \\ 13 \\ -18 \end{pmatrix}$$

Решить линейные системы, используя программы DECOMP и SOLVE, при $p = 1.0, 0.1, 0.01, 0.0001, 0.000001$. Сравнить решение системы $Ax_1 = b$ с решением системы $A^TAx_2 = A^Tb$, полученной из исходной, левой трансформацией Гаусса.
Проанализировать связь числа обусловленности cond и величины $\delta = \|x_1 - x_2\| / \|x_1\|$.

Санкт-Петербург
2024

Результаты

```
p = 1:
Difference between two parts:
        x0: 5.14033e-12
        x1: 4.99396e-12
        x2: -4.83125e-12
        x3: 4.90896e-12
        x4: -5.11235e-12
        x5: -4.90763e-12
        x6: -4.96936e-12
        x7: 5.03508e-12
cond:   12837.1         3.56911e+07
norm:   6.42542e-13
```

```
p = 0.1:
Difference between two parts:
        x0: -3.73777e-11
        x1: -3.72698e-11
        x2: 3.71538e-11
        x3: -3.72067e-11
        x4: 3.73568e-11
        x5: 3.72101e-11
        x6: 3.72551e-11
        x7: -3.73035e-11
cond:   133903          4.01846e+09
norm:   4.6696e-12
```

```
p = 0.01:
Difference between two parts:
        x0: 2.02713e-10
        x1: 2.02657e-10
        x2: -2.02589e-10
        x3: 2.02625e-10
        x4: -2.027e-10
        x5: -2.0262e-10
        x6: -2.02647e-10
        x7: 2.02673e-10
cond:   1.34462e+06             4.06563e+11
norm:   2.53391e-11
```

```
p = 0.0001:
Difference between two parts:
        x0: 5.62011e-09
        x1: 5.6201e-09
        x2: -5.62007e-09
        x3: 5.62009e-09
        x4: -5.62011e-09
        x5: -5.62009e-09
        x6: -5.6201e-09
        x7: 5.6201e-09
cond:   1.34524e+08             4.07085e+15
norm:   7.02514e-10
```

```
p = 1e-06:
Difference between two parts:
        x0: 5.39109e-07
        x1: 5.39109e-07
        x2: -5.39109e-07
        x3: 5.39109e-07
        x4: -5.39109e-07
        x5: -5.39109e-07
        x6: -5.39109e-07
        x7: 5.39109e-07
cond:   1.34524e+10             1e+32
norm:   6.73886e-08
```

Выводы

Решения двух систем максимально схожи, что видно по разности между соответствующими значениями x. Причем с уменьшением значения параметра P увеличивается разница между ними. Так же с уменьшением параметра P увеличивается число обусловленности и уменьшается подсчитываемая величина. Это происходит потому что:

- Матрица близка к вырождению (определитель стремится к 0 с изменением параметра)
- Различия между элементами растет

Код программы

<DIR>/computational_mathematics/second_lab/Decomp.cpp

```
#include "Decomp.h"

#include <stdexcept>
#include <cmath>
#include "Solve.h"

int dimkashelk::details::decomp(int n, int ndim,
                                double *a, double *cond,
                                int pivot[], int *flag)

/* Purpose ...
   -------
   Decomposes a real matrix by gaussian elimination
   and estimates the condition of the matrix.
```

```
Use Solve to compute solutions to linear systems.

Input ...
-----
n    = order of the matrix
ndim = row dimension of matrix as defined in the calling program
*a   = pointer to matrix to be triangularized

Output ...
------
*a          pointer to  an upper triangular matrix U and a
        permuted version of a lower triangular matrix I-L
        so that
        (permutation matrix) * a = L * U
cond      = an estimate of the condition of a .
        For the linear system a * x = b, changes in a and b
        may cause changes cond times as large in x.
        If cond+1.0 .eq. cond , a is singular to working
        precision, cond is set to 1.0e+32 if exact (or near)
        singularity is detected.
pivot     = the pivot vector.
pivot[k]  = the index of the k-th pivot row
pivot[n-1]= (-1)**(number of interchanges)
flag      = Status indicator
            0 : successful execution
            1 : could not allocate memory for workspace
            2 : illegal user input n < 1, a == NULL,
                pivot == NULL, n > ndim.
            3 : matrix is singular

Work Space ...
----------
The vector work[0..n] is allocated internally by decomp().

This C code written by ...  Peter & Nigel,
----------------------      Design Software,
                            42 Gubberley St,
                            Kenmore, 4069,
                            Australia.

Version  ... 1.1 ,  2-Dec-87
-------      2.0 , 11-Feb-89  (pointer used for a)
             2.1 , 15-Apr-89  (work[] allocated internally)
             2.2 , 14-Aug-89  (fixed pivoting)
             2.3 , 3 -Sep-89  (face lift)
             3.0 , 30-Sep-89  (optimize for rowwise storage)

Notes ...
-----
(1) Subscripts range from 0 through (ndim-1).
```

```
   (2) The determinant of a can be obtained on output by
       det(a) = pivot[n-1] * a[0][0] * a[1][1] * ... * a[n-1][n-1].

   (3) This routine has been adapted from that in the text
       G.E. Forsythe, M.A. Malcolm & C.B. Moler
       Computer Methods for Mathematical Computations.

   (4) Uses the functions fabs(), free() and malloc().
*/

{
    /* --- function decomp() --- */
    double EPSILON = 2.2e-16;
    double ek, t, pvt, anorm, ynorm, znorm;
    int i, j, k, m;
    double *pa, *pb; /* temporary pointers */
    double *work;

    *flag = 0;
    work = (double *) NULL;

    if (a == NULL || pivot == NULL || n < 1 || ndim < n) {
        *flag = 2;
        return (0);
    }

    pivot[n - 1] = 1;
    if (n == 1) {
        /* One element only */
        *cond = 1.0;
        if (*a == 0.0) {
            *cond = 1.0e+32; /* singular */
            *flag = 3;
            return (0);
        }
        return (0);
    }

    work = (double *) malloc(n * sizeof(double));
    if (work == NULL) {
        *flag = 1;
        return (0);
    }

    /* --- compute 1-norm of a --- */

    anorm = 0.0;
    for (j = 0; j < n; ++j) {
        t = 0.0;
        for (i = 0; i < n; ++i) t += fabs(a[(i * ndim + j)]);
        if (t > anorm) anorm = t;
    }
```

```
/* Apply Gaussian elimination with partial pivoting. */

for (k = 0; k < n - 1; ++k) {
    /* Find pivot and label as row m.
       This will be the element with largest magnitude in
       the lower part of the kth column. */
    m = k;
    pvt = fabs(a[(m * ndim + k)]);
    for (i = k + 1; i < n; ++i) {
        t = fabs(a[(i * ndim + k)]);
        if (t > pvt) {
            m = i;
            pvt = t;
        }
    }
    pivot[k] = m;
    pvt = a[(m * ndim + k)];

    if (m != k) {
        pivot[n - 1] = -pivot[n - 1];
        /* Interchange rows m and k for the lower partition. */
        for (j = k; j < n; ++j) {
            pa = a + (m * ndim + j);
            pb = a + (k * ndim + j);
            t = *pa;
            *pa = *pb;
            *pb = t;
        }
    }
    /* row k is now the pivot row */

    /* Bail out if pivot is too small */
    if (fabs(pvt) < anorm * EPSILON) {
        /* Singular or nearly singular */
        *cond = 1.0e+32;
        *flag = 3;
        goto DecompExit;
    }

    /* eliminate the lower matrix partition by rows
       and store the multipliers in the k sub-column */
    for (i = k + 1; i < n; ++i) {
        pa = a + (i * ndim + k); /* element to eliminate */
        t = -(*pa / pvt); /* compute multiplier   */
        *pa = t; /* store multiplier     */
        for (j = k + 1; j < n; ++j) /* eliminate i th row */
        {
            if (fabs(t) > anorm * EPSILON)
                a[(i * ndim + j)] += a[(k * ndim + j)] * t;
        }
    }
```

```
} /* End of Gaussian elimination. */

/* cond = (1-norm of a)*(an estimate of 1-norm of a-inverse)
   estimate obtained by one step of inverse iteration for the
   small singular vector. This involves solving two systems
   of equations, (a-transpose)*y = e and a*z = y where e
   is a vector of +1 or -1 chosen to cause growth in y.
   estimate = (1-norm of z)/(1-norm of y)

   Solve (a-transpose)*y = e   */

for (k = 0; k < n; ++k) {
    t = 0.0;
    if (k != 0) {
        for (i = 0; i < k; ++i) t += a[(i * ndim + k)] * work[i];
    }
    if (t < 0.0) ek = -1.0;
    else ek = 1.0;
    pa = a + (k * ndim + k);
    if (fabs(*pa) < anorm * EPSILON) {
        /* Singular */
        *cond = 1.0e+32;
        *flag = 3;
        goto DecompExit;
    }

    work[k] = -(ek + t) / *pa;
}

for (k = n - 2; k >= 0; --k) {
    t = 0.0;
    for (i = k + 1; i < n; i++)
        t += a[(i * ndim + k)] * work[i];
    /* we have used work[i] here, however the use of work[k]
   makes some difference to cond */
    work[k] = t;
    m = pivot[k];
    if (m != k) {
        t = work[m];
        work[m] = work[k];
        work[k] = t;
    }
}

ynorm = 0.0;
for (i = 0; i < n; ++i) ynorm += fabs(work[i]);

/* --- solve a * z = y */
solve(n, ndim, a, work, pivot);

znorm = 0.0;
for (i = 0; i < n; ++i) znorm += fabs(work[i]);
```

```
    /* --- estimate condition --- */
    *cond = anorm * znorm / ynorm;
    if (*cond < 1.0) *cond = 1.0;
    if (*cond + 1.0 == *cond) *flag = 3;

DecompExit:
    if (work != NULL) {
        free(work);
        work = (double *) NULL;
    }
    return (0);
} /* --- end of function decomp() --- */

dimkashelk::Decomp::Decomp(): cond_(0.0),
                              size_(0),
                              data_(nullptr),
                              pivot_(nullptr),
                              flag_(0) {
}

void dimkashelk::Decomp::operator()(const std::vector<std::vector<double> >
&matrix) {
    if (matrix.empty()) {
        throw std::logic_error("Check matrix");
    }
    if (matrix.size() != matrix[0].size()) {
        throw std::logic_error("Check size of matrix");
    }
    free();
    size_ = static_cast<int>(matrix.size());
    data_ = new double[size_ * size_];
    try {
        pivot_ = new int[matrix.size() * matrix.size()];
    } catch (...) {
        delete[] data_;
        throw;
    }
    int ind = 0;
    for (auto &i: matrix) {
        for (double j: i) {
            data_[ind] = j;
            ind++;
        }
    }
    details::decomp(size_, size_, data_, std::addressof(cond_), pivot_,
std::addressof(flag_));
}

dimkashelk::Decomp::~Decomp() {
    free();
}
```

```
void dimkashelk::Decomp::free() const {
    if (data_ != nullptr) {
        delete[] data_;
    }
    if (pivot_ != nullptr) {
        delete[] pivot_;
    }
}
```

<DIR>/computational_mathematics/second_lab/Decomp.h

```
#ifndef DECOMP_H
#define DECOMP_H
#include <vector>

namespace dimkashelk {
    namespace details {
        int decomp(int n, int ndim,
                   double *a, double *cond,
                   int pivot[], int *flag);
    }

    class Solve;

    class Decomp {
        friend class Solve;

    public:
        Decomp();

        void operator()(const std::vector<std::vector<double> > &matrix);

        ~Decomp();

    private:
        double cond_;
        int size_;
        double *data_;
        int *pivot_;
        int flag_;

        void free() const;
    };
}
#endif
```

<DIR>/computational_mathematics/second_lab/Solve.cpp

```
#include "Solve.h"
```

```
#include <stdexcept>

#include "Decomp.h"

int dimkashelk::details::solve(int n, int ndim,
                               double *a, double b[],
                               int pivot[])

/* Purpose :
   -------
   Solution of linear system, a * x = b.
   Do not use if decomp() has detected singularity.

   Input..
   -----
   n     = order of matrix
   ndim  = row dimension of a
   a     = triangularized matrix obtained from decomp()
   b     = right hand side vector
   pivot = pivot vector obtained from decomp()

   Output..
   ------
   b = solution vector, x

*/

{
    /* --- begin function solve() --- */

    int i, j, k, m;
    double t;

    if (n == 1) {
        /* trivial */
        b[0] /= a[0];
    } else {
        /* Forward elimination: apply multipliers. */
        for (k = 0; k < n - 1; k++) {
            m = pivot[k];
            t = b[m];
            b[m] = b[k];
            b[k] = t;
            for (i = k + 1; i < n; ++i) b[i] += a[(i * ndim + k)] * t;
        }

        /* Back substitution. */
        for (k = n - 1; k >= 0; --k) {
            t = b[k];
            for (j = k + 1; j < n; ++j) t -= a[(k * ndim + j)] * b[j];
            b[k] = t / a[(k * ndim + k)];
        }
```

```
    }

    return (0);
} /* --- end function solve() --- */

dimkashelk::Solve::Solve(): size_(0),
                            data_right_(nullptr),
                            cond_(0.0) {
}

void dimkashelk::Solve::operator()(const std::vector<std::vector<double> >
&matrix_left,
                                   const std::vector<double> &matrix_right) {
    if (matrix_left.size() != matrix_right.size()) {
        throw std::logic_error("Check data");
    }
    free();

    size_ = static_cast<int>(matrix_right.size());
    data_right_ = new double[size_];
    for (int i = 0; i < size_; i++) {
        data_right_[i] = matrix_right[i];
    }
    Decomp dec;
    dec(matrix_left);
    cond_ = dec.cond_;
    const int size = static_cast<int>(matrix_left.size());
    details::solve(size, size, dec.data_, data_right_, dec.pivot_);
}

std::vector<double> dimkashelk::Solve::get_result() const {
    std::vector<double> res(size_);
    for (int i = 0; i < size_; i++) {
        res[i] = data_right_[i];
    }
    return res;
}

double dimkashelk::Solve::get_cond() const {
    return cond_;
}

dimkashelk::Solve::~Solve() {
    free();
}

void dimkashelk::Solve::free() const {
    if (data_right_ != nullptr) {
        delete[] data_right_;
    }
}
```

<DIR>/computational_mathematics/second_lab/Solve.h

```
#ifndef SOLVE_H
#define SOLVE_H
#include <vector>

namespace dimkashelk {
    namespace details {
        int solve(int n, int ndim,
                  double *a, double b[],
                  int pivot[]);
    }

    class Decomp;

    class Solve {
        friend class Decomp;

    public:
        Solve();

        void operator()(const std::vector<std::vector<double> > &matrix_left,
const std::vector<double> &matrix_right);

        [[nodiscard]] std::vector<double> get_result() const;
        [[nodiscard]] double get_cond() const;

        ~Solve();

    private:
        int size_;
        double *data_right_;
        double cond_;

        void free() const;
    };
}

#endif
```

<DIR>/computational_mathematics/second_lab/main.cpp

```
#include <algorithm>
#include <iostream>

#include "Solve.h"

std::vector<double> get_difference_of_matrix(const std::vector<double> &left,
                                             const std::vector<double>
&right) {
    std::vector result = left;
    for (int i = 0; i < left.size(); i++) {
```

```
        result[i] -= right[i];
    }
    return result;
}

double get_matrix_norm(const std::vector<double> &matrix) {
    return *std::max_element(matrix.begin(), matrix.end());
}

std::vector<std::vector<double> >
get_gaussian_elimination(std::vector<std::vector<double> > &m) {
    auto matrix = m;
    const int n = static_cast<int>(matrix.size());
    for (int i = 0; i < n; ++i) {
        int maxRow = i;
        for (int k = i + 1; k < n; ++k) {
            if (std::abs(matrix[k][i]) > std::abs(matrix[maxRow][i])) {
                maxRow = k;
            }
        }
        if (maxRow != i) {
            std::swap(matrix[i], matrix[maxRow]);
        }

        for (int k = i + 1; k < n; ++k) {
            const double factor = matrix[k][i] / matrix[i][i];
            for (int j = i; j < n + 1; ++j) {
                matrix[k][j] -= factor * matrix[i][j];
            }
        }
    }

    for (int i = n - 1; i >= 0; --i) {
        for (int j = i + 1; j < n; ++j) {
            matrix[i][n] -= matrix[i][j] * matrix[j][n];
        }
        matrix[i][n] /= matrix[i][i];
    }
    return matrix;
}

std::vector<std::vector<double> > multiply_matrices(const
std::vector<std::vector<double> > &matrix1,
                                                    const
std::vector<std::vector<double> > &matrix2) {
    const int rows1 = static_cast<int>(matrix1.size());
    const int cols1 = static_cast<int>(matrix1[0].size());
    const int cols2 = static_cast<int>(matrix2[0].size());

    std::vector result(rows1, std::vector(cols2, 0.0));

    for (int i = 0; i < rows1; ++i) {
```

```
        for (int j = 0; j < cols2; ++j) {
            for (int k = 0; k < cols1; ++k) {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
    return result;
}

std::vector<double> multiply_matrices(const std::vector<std::vector<double> >
&matrix1,
                                      const std::vector<double> &matrix2) {
    const int rows1 = static_cast<int>(matrix1.size());
    const int cols1 = static_cast<int>(matrix1.size());
    const int cols2 = static_cast<int>(matrix2.size());

    std::vector result(rows1, 0.0);

    for (int i = 0; i < cols2; i++) {
        for (int j = 0; j < rows1; j++) {
            result[i] += matrix1[i][j] * matrix2[j];
        }
    }
    return result;
}

std::vector<std::vector<double> > get_left_matrix(double p) {
    return {
        {p - 29, 6, -6, -4, -3, -8, -5, 5},
        {6, -13, -3, 5, 4, 3, 1, 7},
        {5, -5, -1, 7, 2, 0, 7, 1},
        {5, -5, 5, 6, 4, -7, 4, 0},
        {4, 4, 7, -4, 9, -8, -8, -4},
        {-4, 5, -4, 1, 0, 12, 0, 6},
        {-3, -2, -4, 2, -8, -3, 16, 4},
        {7, 5, 0, 2, 0, -6, 8, -12}
    };
}

std::vector<double> get_right_matrix(double p) {
    return {
        4 * p - 175,
        133,
        110,
        112,
        17,
        32,
        13,
        -18
    };
}
```

```
int main() {
    dimkashelk::Solve solve;

    const auto numbers = {1.0, 0.1, 0.01, 0.0001, 0.000001};
    for (const auto number: numbers) {
        std::cout << "p = " << number << ": \n";
        std::cout << "Part one\t\t\tPart two\n";

        auto left = get_left_matrix(number);
        auto right = get_right_matrix(number);
        solve(left, right);
        auto res1 = solve.get_result();
        const auto cond1 = solve.get_cond();

        auto gaussian = get_gaussian_elimination(left);
        auto new_left = multiply_matrices(gaussian, left);
        auto new_right = multiply_matrices(gaussian, right);
        solve(new_left, new_right);
        auto res2 = solve.get_result();
        const auto cond2 = solve.get_cond();

        for (int i = 0; i < res1.size(); i++) {
            std::cout << "\tx" << i << ": " << res1[i] << "\t\t\t" << res2[i]
<< '\n';
        }
        std::cout << "cond: \t" << cond1 << "\t\t" << cond2 << "\n";
        const auto res3 = get_matrix_norm(get_difference_of_matrix(res1,
res2)) / get_matrix_norm(res1);
        std::cout << "norm:\t" << res3 << "\n";
        std::cout << "\n";
    }
    return 0;
}
```