Федеральное государственное автономное образовательное учреждение высшего образования «Санкт-Петербургский политехнический университет Петра Великого»
Институт компьютерных наук и технологий
Программная инженерия

ПОЛИТЕХ
Санкт-Петербургский политехнический университет Петра Великого

Отчёт по лабораторным работам
по дисциплине «Вычислительная математика»

Студент гр. 5130904/30008 Ребдев П.А
Руководитель: Скуднева Е.В

Санкт-Петербург
2025

Оглавление

Лабораторная работа №2

1. Задание

Вариант №16

Написать процедуру формирования матрицы B по формулам:

$$B_{ik}=\begin{cases}\dfrac{0.01}{(N-i+k)(i+1)} & для \quad i=k\\ 0 & для \quad i<k\\ i(N-K) & для \quad i>k\end{cases}$$

Вычислить матрицу B^{-1}, используя процедуры DECOMP и SOLVE, и найти норму матрицы $R = BB^{-1} - E$ для N = 3, 6, 9 . Объяснить результаты.

$$\|R\|=\sqrt{\sum_{i}^{N}\sum_{k}^{N}R_{ik}^{2}}$$

2. Решение

2.1 Ход решения

1. Сформировать матрицы по заданному правилу

2. Вычислить обратные матрицы с помощью процедур DECOMP и SOLVE

3. Сформировать R матрицы, которые равны разнице произведений оригинальных и обратных матриц и единичных матриц

4. Вычислить нормы R матриц

2.2 Основная программа

main.cpp

```
#include <iostream>
#include <iomanip>
#include <vector>

#include "matrixFunc.hpp"

int main()
{
  size_t n[] = {3, 6, 9};

  for (size_t q = 0; q < 3; ++q)
  {
    std::vector< double > b(n[q] * n[q]);
    fillMatrix(b);
    std::cout << "\033[1;32mStandart matrix " << n[q] << 'x' << n[q] << ":\033[1;0m\n";
    printMatrix(b, std::cout);

    std::vector< double > antiB(antiMatrix(b));
    std::cout << "\033[1;33mAnti matrix:\033[1;0m\n";
    printMatrix(antiB, std::cout);

    std::vector< double > r(multiMatrix(b, antiB));
    for (int i = 0; i < n[q]; ++i)
    {
      r[i * n[q] + i] -= 1;
    }
    std::cout << "\033[1;34mR matrix:\033[1;0m\n";
    printMatrix(r, std::cout);
    std::cout << "\033[1;35mNorm of R: " << calcNorm(r) << "\033[1;0m\n\n";
  }
  return 0;
}
```

2.3 Вычисление обратной матрицы

matrixFunc.cpp

```
vec antiMatrix(const vec & m)
{
  int dim = std::sqrt(m.size());
  vec copy(m);
  double cond;
  std::vector< int > ipvt(dim);
  vec work(dim);

  decomp_(&dim, &dim, copy.data(), &cond, ipvt.data(), work.data());

  vec result(m.size());
  for (int i = 0; i < dim; ++i)
  {
    vec b(dim, 0.0);
    b[i] = 1.0;
    solve_(&dim, &dim, copy.data(), b.data(), ipvt.data());
    for (int j = 0; j < dim; ++j)
    {
      result[i * dim + j] = b[j];
    }
  }
  return result;
}
```

2.4 Перемножение матриц

matrixFunc.cpp

```
vec multiMatrix(const vec & first, const vec & second) noexcept
{
  vec result(first.size());
  int n = std::sqrt(first.size());
  for (size_t i = 0; i < n; ++i)
  {
    for (size_t j = 0; j < n; ++j)
    {
      result[i * n + j] = 0;
      for (size_t q = 0; q < n; ++q)
      {
        result[i * n + j] += first[i * n + q] * second[q * n + j];
      }
    }
  }
  return result;
}
```

2.5 Вычисление нормы матрицы

matrixFunc.cpp

```
double calcNorm(const vec & m)
{
  int n = std::sqrt(m.size());
  double norm = 0;
  for (size_t i = 0; i < n; ++i)
  {
    for (size_t j = 0; j < n; ++j)
```

```
    {
      norm += std::pow(m[i * n + j], 2);
    }
  }
  norm = std::sqrt(norm);
  return norm;
}
```

3. Результаты

3.1 Оригинальные матрицы

N = 3

3,33E-03	0	0
3	1,67E-03	0
6	4	1,11E-03

N = 6

1,67E-03	0	0	0	0	0
6	8,33E-04	0	0	0	0
12	10	5,56E-04	0	0	0
18	15	12	4,17E-04	0	0
24	20	16	12	3,33E-04	0
30	25	20	15	10	2,78E-04

N = 9

1,11E-03	0	0	0	0	0	0	0	0
9	5,56E-04	0	0	0	0	0	0	0
18	16	3,70E-04	0	0	0	0	0	0
27	24	21	2,78E-04	0	0	0	0	0
36	32	28	24	2,22E-04	0	0	0	0
45	40	35	30	25	1,85E-04	0	0	0
54	48	42	36	30	24	1,59E-04	0	0
63	56	49	42	35	28	21	1,39E-04	0
72	64	56	48	40	32	24	16	1,23E-04

3.2 Обратные матрицы

N = 3

300	0	0
-5,40E+05	600	0
1,94E+09	-2,16E+06	900

N = 6

600	0	0	0	0	0
-4,32E+06	1200	0	0	0	0
7,77E+10	-2,16E07	1800	0	0	0
-2,24E+15	6,22E+11	-5,18E+07	2400	0	0
8,06E+19	-2,24E+16	1,87E+12	-8,64E+07	3000	0
-2,90E+24	8,06E+20	-6,72E+16	3,11E+12	-1,08E+08	3600

N = 9

900	0	0	0	0	0	0	0	0
-1,46E+07	1800	0	0	0	0	0	0	0
6,30E+11	-7,78E+07	2700	0	0	0	0	0	0
-4,76E+16	5,88E+12	-2,04E+08	3600	0	0	0	0	0
5,14E+21	-6,35E+17	2,20E+13	-3,89E+08	4500	0	0	0	0
-6,94E+26	8,57E+22	-2,98E+18	5,25E+13	-6,08E+08	5400	0	0	0
1,05E+32	-1,230E+28	4,50E+23	-7,97E+18	9,19E+13	-8,16E+08	6300	0	0
-1,59E+37	1,96E+33	-6,80E+28	1,20E+24	-1,39E+19	1,23E+14	-9,53E+08	7200	0
2,06E+42	-2,54E+38	8,82E+33	-1,56E+29	1,80E+24	-1,60E+19	1,23E+14	-9,33E+08	8100

3.3 R матрицы

N = 3

0	0	0
-1,14E-13	0	0
-4,66E-10	0	0

N = 6

0	0	0	0	0	0
-4,55E-13	0	0	0	0	0
0	2,98E-08	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
1,31E+05	0	0	0	0	0

N = 9

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
-128	0	0	0	0	0	0	0	0
-5,03E+07	2048	0	0	0	0	0	0	0
-4,40E+12	0	0	0	0	0	0	0	0
0	-3,52E+13	0	0,25	0	0	0	0	0
7,56E+22	-9,22E+18	0	-2,15E+09	3,27E+04	-0,25	0	0	0

3.4 Нормы

N = 3; ||R|| = 4,66E-10

N = 6; ||R|| = 1,31E+05

N = 9; ||R|| = 7,56E+22

4. Выводы

В ходе выполнения лабораторной работы был изучен метод нахождения обратной матрицы с помощью процедур DECOMP и SOLVE. Было обнаружено, что с увеличением стороны квадратной матрицы в 2 раза, среднеквадратичная норма R вырастает на десятки порядков, что может свидетельствует о том, что с ростом размера матрицы значительно возрастают погрешности при вычисление обратной матрицы, а так же погрешность при перемножение матриц. В ходе выполнения лабораторный работы для хранения элементов матрицы использовался тип данных double, что может являться причиной резкого роста погрешности, так как double поддерживает лишь 16 чисел после мантиссы. Можно с уверенностью заключить, что с ростом размера матрицы значительно вырастают и погрешности при вычислениях

5. Дополнения

5.1 Полный код всех программ

Описание файлов:

- main.cpp является связующим звеном между функциями реализованными в matrixFunc.*
- matrixFunc.* содержит функции заполнения матрицы, вывода матрицы, нахождения нормы матрицы, перемножения матриц и нахождения обратной матрицы с использованием процедур DECOMP и SOLVE
- *.f файлы содержащие оригинальные Фортран процедуры DECOMP и SOLVE

main.cpp

```
#include <iostream>
#include <iomanip>
#include <vector>

#include "matrixFunc.hpp"

int main()
{
  size_t n[] = {3, 6, 9};

  for (size_t q = 0; q < 3; ++q)
  {
    std::vector< double > b(n[q] * n[q]);
    fillMatrix(b);
    std::cout << "\033[1;32mStandart matrix " << n[q] << 'x' << n[q] << ":\033[1;0m\n";
    printMatrix(b, std::cout);

    std::vector< double > antiB(antiMatrix(b));
    std::cout << "\033[1;33mAnti matrix:\033[1;0m\n";
    printMatrix(antiB, std::cout);

    std::vector< double > r(multiMatrix(b, antiB));
    for (int i = 0; i < n[q]; ++i)
    {
      r[i * n[q] + i] -= 1;
    }
    std::cout << "\033[1;34mR matrix:\033[1;0m\n";
    printMatrix(r, std::cout);
    std::cout << "\033[1;35mNorm of R: " << calcNorm(r) << "\033[1;0m\n\n";
  }
  return 0;
}
```

matrixFunc.hpp

```cpp
#ifndef MATRIXFUNC_HPP
#define MATRIXFUNC_HPP

#include <iostream>
#include <vector>

using vec = std::vector< double >;

void fillMatrix(vec & m) noexcept;
void printMatrix(const vec & m, std::ostream & out);
vec antiMatrix(const vec & m);
vec multiMatrix(const vec & first, const vec & second) noexcept;
double calcNorm(const vec & m);

#endif
```

matrixFunc.cpp

```cpp
#include "matrixFunc.hpp"

#include <cmath>
#include <iomanip>

extern "C" {
  int decomp_(int *ndim, int *n, double *a, double *cond, int *ipvt, double *work);
  int solve_(int *ndim, int *n, double *a, double *b, int *ipvt);
}

void fillMatrix(vec & m) noexcept
{
  int n = std::sqrt(m.size());
  for (int i = 0; i < n; ++i)
  {
    for (int j = 0; j < n; ++j)
    {
      if (i == j)
      {
        m[i * n + j] = 0.01 / ((n - i + j) * (i + 1));
      }
      else if (i < j)
      {
        m[i * n + j] = 0;
      }
      else
      {
        m[i * n + j] = i * (n - j);
      }
    }
  }
}
void printMatrix(const vec & m, std::ostream & out)
{
  int n = std::sqrt(m.size());
  for (size_t i = 0; i < n; ++i)
  {
    for (size_t j = 0; j < n; ++j)
    {
```

```
      out << m[i * n + j] << '\t';
    }
    out << '\n';
  }
}
vec antiMatrix(const vec & m)
{
  int dim = std::sqrt(m.size());
  vec copy(m);
  double cond;
  std::vector< int > ipvt(dim);
  vec work(dim);

  decomp_(&dim, &dim, copy.data(), &cond, ipvt.data(), work.data());

  vec result(m.size());
  for (int i = 0; i < dim; ++i)
  {
    vec b(dim, 0.0);
    b[i] = 1.0;
    solve_(&dim, &dim, copy.data(), b.data(), ipvt.data());
    for (int j = 0; j < dim; ++j)
    {
      result[i * dim + j] = b[j];
    }
  }
  return result;
}
vec multiMatrix(const vec & first, const vec & second) noexcept
{
  vec result(first.size());
  int n = std::sqrt(first.size());
  for (size_t i = 0; i < n; ++i)
  {
    for (size_t j = 0; j < n; ++j)
    {
      result[i * n + j] = 0;
      for (size_t q = 0; q < n; ++q)
      {
        result[i * n + j] += first[i * n + q] * second[q * n + j];
      }
    }
  }
  return result;
}
double calcNorm(const vec & m)
{
  int n = std::sqrt(m.size());
  double norm = 0;
  for (size_t i = 0; i < n; ++i)
  {
    for (size_t j = 0; j < n; ++j)
    {
      norm += std::pow(m[i * n + j], 2);
    }
  }
```

```
  norm = std::sqrt(norm);
  return norm;
}
```

decomp.f

```
c  Code from http://www.netlib.org/fmm/
      subroutine decomp(ndim,n,a,cond,ipvt,work)
c
      integer ndim,n
      double precision a(ndim,n),cond,work(n)
      integer ipvt(n)
c
c     decomposes a double precision matrix by gaussian elimination
c     and estimates the condition of the matrix.
c
c     use solve to compute solutions to linear systems.
c
c     input..
c
c        ndim = declared row dimension of the array containing  a.
c
c        n = order of the matrix.
c
c        a = matrix to be triangularized.
c
c     output..
c
c        a  contains an upper triangular matrix  u  and a permuted
c          version of a lower triangular matrix  i-l  so that
c          (permutation matrix)*a = l*u .
c
c        cond = an estimate of the condition of  a .
c           for the linear system  a*x = b, changes in  a  and  b
c           may cause changes  cond  times as large in  x .
c           if  cond+1.0 .eq. cond , a is singular to working
c           precision.  cond  is set to  1.0d+32  if exact
c           singularity is detected.
c
c        ipvt = the pivot vector.
c           ipvt(k) = the index of the k-th pivot row
c           ipvt(n) = (-1)**(number of interchanges)
c
c     work space..  the vector  work  must be declared and included
c                   in the call.  its input contents are ignored.
c                   its output contents are usually unimportant.
c
c     the determinant of a can be obtained on output by
c        det(a) = ipvt(n) * a(1,1) * a(2,2) * ... * a(n,n).
c
      double precision ek, t, anorm, ynorm, znorm
      integer nm1, i, j, k, kp1, kb, km1, m
      double precision dabs, dsign
c
      ipvt(n) = 1
      if (n .eq. 1) go to 80
      nm1 = n - 1
```

```
c
c     compute 1-norm of a
c
      anorm = 0.0d0
      do 10 j = 1, n
         t = 0.0d0
         do 5 i = 1, n
            t = t + dabs(a(i,j))
    5    continue
         if (t .gt. anorm) anorm = t
   10 continue
c
c     gaussian elimination with partial pivoting
c
      do 35 k = 1,nm1
         kp1= k+1
c
c        find pivot
c
         m = k
         do 15 i = kp1,n
            if (dabs(a(i,k)) .gt. dabs(a(m,k))) m = i
   15    continue
         ipvt(k) = m
         if (m .ne. k) ipvt(n) = -ipvt(n)
         t = a(m,k)
         a(m,k) = a(k,k)
         a(k,k) = t
c
c        skip step if pivot is zero
c
         if (t .eq. 0.0d0) go to 35
c
c        compute multipliers
c
         do 20 i = kp1,n
             a(i,k) = -a(i,k)/t
   20    continue
c
c        interchange and eliminate by columns
c
         do 30 j = kp1,n
             t = a(m,j)
             a(m,j) = a(k,j)
             a(k,j) = t
             if (t .eq. 0.0d0) go to 30
             do 25 i = kp1,n
                a(i,j) = a(i,j) + a(i,k)*t
   25        continue
   30    continue
   35 continue
c
c     cond = (1-norm of a)*(an estimate of 1-norm of a-inverse)
c     estimate obtained by one step of inverse iteration for the
c     small singular vector.  this involves solving two systems
c     of equations, (a-transpose)*y = e  and  a*z = y  where  e
```

```
c     is a vector of +1 or -1 chosen to cause growth in y.
c     estimate = (1-norm of z)/(1-norm of y)
c
c     solve (a-transpose)*y = e
c
      do 50 k = 1, n
        t = 0.0d0
        if (k .eq. 1) go to 45
        km1 = k-1
        do 40 i = 1, km1
          t = t + a(i,k)*work(i)
   40   continue
   45   ek = 1.0d0
        if (t .lt. 0.0d0) ek = -1.0d0
        if (a(k,k) .eq. 0.0d0) go to 90
        work(k) = -(ek + t)/a(k,k)
   50 continue
      do 60 kb = 1, nm1
        k = n - kb
        t = 0.0d0
        kp1 = k+1
        do 55 i = kp1, n
          t = t + a(i,k)*work(i)
   55   continue
        work(k) = t + work(k)
        m = ipvt(k)
        if (m .eq. k) go to 60
        t = work(m)
        work(m) = work(k)
        work(k) = t
   60 continue
c
      ynorm = 0.0d0
      do 65 i = 1, n
        ynorm = ynorm + dabs(work(i))
   65 continue
c
c     solve a*z = y
c
      call solve(ndim, n, a, work, ipvt)
c
      znorm = 0.0d0
      do 70 i = 1, n
        znorm = znorm + dabs(work(i))
   70 continue
c
c     estimate condition
c
      cond = anorm*znorm/ynorm
      if (cond .lt. 1.0d0) cond = 1.0d0
      return
c
c     1-by-1
c
   80 cond = 1.0d0
      if (a(1,1) .ne. 0.0d0) return
```

```
c
c     exact singularity
c
   90 cond = 1.0d+32
      return
      end
```

solve.f

```
c  Code from http://www.netlib.org/fmm/
      subroutine solve(ndim, n, a, b, ipvt)
c
      integer ndim, n, ipvt(n)
      double precision a(ndim,n),b(n)
c
c   solution of linear system, a*x = b .
c   do not use if decomp has detected singularity.
c
c   input..
c
c     ndim = declared row dimension of array containing a .
c
c     n = order of matrix.
c
c     a = triangularized matrix obtained from decomp .
c
c     b = right hand side vector.
c
c     ipvt = pivot vector obtained from decomp .
c
c   output..
c
c     b = solution vector, x .
c
      integer kb, km1, nm1, kp1, i, k, m
      double precision t
c
c     forward elimination
c
      if (n .eq. 1) go to 50
      nm1 = n-1
      do 20 k = 1, nm1
         kp1 = k+1
         m = ipvt(k)
         t = b(m)
         b(m) = b(k)
         b(k) = t
         do 10 i = kp1, n
             b(i) = b(i) + a(i,k)*t
   10    continue
   20 continue
c
c     back substitution
c
      do 40 kb = 1,nm1
         km1 = n-kb
         k = km1+1
```

```
      b(k) = b(k)/a(k,k)
      t = -b(k)
      do 30 i = 1, km1
         b(i) = b(i) + a(i,k)*t
30    continue
40 continue
50 b(1) = b(1)/a(1,1)
   return
   end
```