Kacper Korta,

16.05.2023

Laboratorium 09

Układy równań – metody bezpośrednie

1. Treść zadań

Korzystając z przykładu napisz program, który:

1. Jako parametr pobiera rozmiar układu równań n
2. Generuje macierz układu A(nxn) i wektor wyrazów wolnych b(n)
3. Rozwiązuje układ równań na trzy sposoby:

a. poprzez dekompozycję LU macierzy A: A=LU, posługując się funkcjami GSL: gsl_linalg_LU_decomp () i gsl_linalg_LU_solve (); wydrukować na ekran wyjściową macierz (parametr pierwszy) i wektor (parametr drugi)

b. poprzez odwrócenie macierzy A: x=A-1 b, posługując się funkcją GSL: gsl_linalg_LU_invert (); sprawdzić czy AA-1=I i A-1A=I (macierz jednostkowa)

c. poprzez dekompozycję QR macierzy A: A=QR, posługując się funkcjami GSL: gsl_linalg_QR_decomp () i gsl_linalg_QR_solve (); wydrukować na ekran wyjściową macierz (parametr pierwszy) i wektor (parametr drugi)

4. Sprawdzić poprawność rozwiązania (tj., czy Ax=b)
5. Zmierzyć całkowity czas rozwiązania układu - do mierzenia czasu można skorzystać z przykładowego programu dokonującego pomiaru czasu procesora spędzonego w danym fragmencie programu.
6. Porównać czasy z trzech sposobów: poprzez dekompozycję LU, poprzez odwrócenie macierzy i poprzez dekompozycję QR

Zadanie domowe: Narysuj wykres zależności całkowitego czasu rozwiązywania układu (LU, QR, odwrócenie macierzy) od rozmiaru układu równań. Wykonaj pomiary dla 5 wartości z przedziału od 10 do 100.

2. Rozwiązania

Zadanie 1.

```c
#include <stdio.h>
#include <gsl/gsl_linalg.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_vector.h>
```

```c
#include <gsl/gsl_blas.h>
#include <time.h>

void generateRandomMatrix(gsl_matrix *matrix) {
    int n = matrix->size1;
    int m = matrix->size2;
    srand(time(NULL));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            double value = 10 + (double)rand() / RAND_MAX * 90;
            gsl_matrix_set(matrix, i, j, value);
        }
    }
}

void generateRandomVector(gsl_vector *vector) {
    int n = vector->size;
    srand(time(NULL));
    for (int i = 0; i < n; i++) {
        double value = 10 + (double)rand() / RAND_MAX * 90;
        gsl_vector_set(vector, i, value);
    }
}

void verifySolution(gsl_matrix *A, gsl_vector *b, gsl_vector *x) {

    int n = A->size1;

    gsl_vector *Ax = gsl_vector_alloc(n);
    gsl_blas_dgemv(CblasNoTrans, 1.0, A, x, 0.0, Ax);

    // printf("Verification: ... \n");

    double tolerance = 1e-6;
    int verified = 1;
    for (int i = 0; i < n; i++) {
        if (fabs(gsl_vector_get(Ax, i) - gsl_vector_get(b, i)) > tolerance) {
            verified = 0;
            break;
        }

    }

    if (verified) {
        // pass
        // printf("Solutions verified. Ax equals b within the tolerance.\n");

    } else {
```

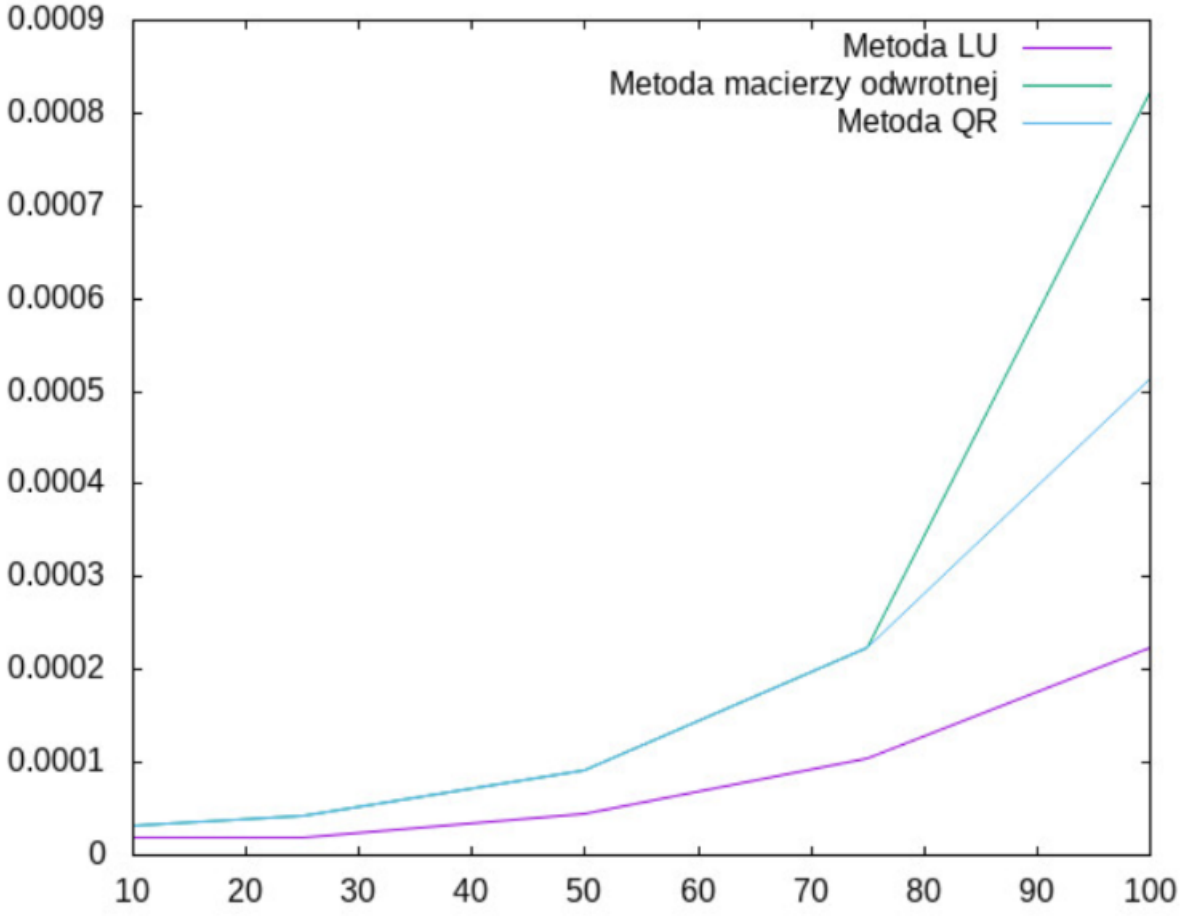
```
        printf("Solutions not verified. Ax does not equal b within the
tolerance.\n");
    }

    gsl_vector_free(Ax);

}

void printMatrix(gsl_matrix *matrix) {
    int n = matrix->size1;
    int m = matrix->size2;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            printf("%f ", gsl_matrix_get(matrix, i, j));
        }
        printf("\n");
    }
}

void printVector(gsl_vector *vector) {
    int n = vector->size;
    for (int i = 0; i < n; i++) {
        printf("%f\n", gsl_vector_get(vector, i));
    }
}

void verifyInverseMatrix(gsl_matrix *A, gsl_matrix *invA) {
    int n = A->size1;

    gsl_matrix *prod1 = gsl_matrix_alloc(n, n);
    gsl_matrix *prod2 = gsl_matrix_alloc(n, n);
    gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 1.0, A, invA, 0.0, prod1);
    gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 1.0, invA, A, 0.0, prod2);

    double tolerance = 1e-6;
    int verified = 1;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            double value1 = gsl_matrix_get(prod1, i, j);
            double value2 = gsl_matrix_get(prod2, i, j);
            if (i == j) {
                if (fabs(value1 - 1.0) > tolerance || fabs(value2 - 1.0) >
tolerance) {
                    verified = 0;
                    break;
                }
            } else {
                if (fabs(value1) > tolerance || fabs(value2) > tolerance) {
                    verified = 0;
```

```
                    break;
                }
            }
        }
        if (!verified) {
            break;
        }
    }

    if (verified) {
        printf("Inverse matrix verified. AA-1 = I and A-1A = I.\n");
    } else {
        printf("Inverse matrix not verified. AA-1 != I or A-1A != I.\n");
    }

    gsl_matrix_free(prod1);
    gsl_matrix_free(prod2);
}

int main(void) {

    int start_n = 25;
    int end_n = 1200;
    int step = 25;

    FILE *file = fopen("execution_times.csv", "a");
    if (file == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    fprintf(file, "n,time_LU,time_inv,time_QR\n");

    for (int n = start_n; n <= end_n; n += step) {
        gsl_matrix *A = gsl_matrix_alloc(n, n);
        gsl_vector *b = gsl_vector_alloc(n);
        gsl_vector *x = gsl_vector_alloc(n);

        generateRandomMatrix(A);
        generateRandomVector(b);

        gsl_matrix *LU = gsl_matrix_alloc(n, n);
        gsl_matrix_memcpy(LU, A);

        // Solve the linear system using LU decomposition
        gsl_permutation *p = gsl_permutation_alloc(n);
        int signum;
        clock_t start = clock();
        gsl_linalg_LU_decomp(LU, p, &signum);
```

```
        gsl_linalg_LU_solve(LU, p, b, x);
        clock_t end = clock();
        double time_LU = (double) (end - start) / CLOCKS_PER_SEC;

        // Verification
        verifySolution(A, b, x);

        // Solve the linear system by matrix inversion
        gsl_matrix *invA = gsl_matrix_alloc(n, n);
        gsl_matrix_memcpy(invA, A);
        start = clock();
        gsl_linalg_LU_invert(invA, p, invA);
        gsl_blas_dgemv(CblasNoTrans, 1.0, invA, b, 0.0, x);
        end = clock();
        double time_inv = (double) (end - start) / CLOCKS_PER_SEC;

        // Verification
        verifySolution(A, b, x);
//        verifyInverseMatrix(A, invA);

        // Solve the linear system using QR decomposition
        gsl_matrix *QR = gsl_matrix_alloc(n, n);
        gsl_matrix_memcpy(QR, A);
        gsl_vector *tau = gsl_vector_alloc(n);
        start = clock();
        gsl_linalg_QR_decomp(QR, tau);
        gsl_linalg_QR_solve(QR, tau, b, x);
        end = clock();
        double time_QR = (double) (end - start) / CLOCKS_PER_SEC;

        // Verification
        verifySolution(A, b, x);

        fprintf(file, "%d,%.6f,%.6f,%.6f\n", n, time_LU, time_inv, time_QR);

        gsl_matrix_free(A);
        gsl_vector_free(b);
        gsl_vector_free(x);
        gsl_matrix_free(LU);
        gsl_permutation_free(p);
        gsl_matrix_free(invA);
        gsl_matrix_free(QR);
        gsl_vector_free(tau);
    }

    fclose(file);

    return 0;
}
```

Zadanie 2. Korzystając z wyników które otrzymaliśmy przy wykonywaniu programu

Otrzymujemy wykres:



Wykres zależności czasu rozwiązywania a ilości zmiennych

Wnioski: Z wykresu możemy zauważyć, że metoda LU jest najszybsza i znacząco się różni od 2 pozostałych.

3. Bibliografia

Wykład

https://pl.wikipedia.org/wiki/Metoda_LU

https://pl.wikipedia.org/wiki/Rozk%C5%82ad_QR

https://www.gnu.org/software/gsl/doc/html/vectors.html

https://www.gnu.org/software/gsl/doc/html/linalg.html

https://www.gnu.org/software/gsl/doc/html/blas.html