Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Wydział Inżynierii Metali i Informatyki Przemysłowej

Sprawozdanie z Laboratorium:

Rozwiązywanie układów równań liniowych – metoda LU, Choleskiego.

Przedmiot: Metody Numeryczne

Kierunek: Inżynieria Obliczeniowa

Autor: Filip Rak

Prowadzący przedmiot: dr hab. inż. Marcin Hojny

Data: 24 maj 2024

Numer lekcji: 10

Grupa laboratoryjna: 4

Wstęp teoretyczny

Dekompozycja macierzy polega na przekształceniu pojedynczej macierzy na iloczyn kilku macierzy. W analizie numerycznej różne rodzaje dekompozycji są używane do implementacji wydajnych algorytmów.

Jedną z takich metod jest dekompozycja LU, która służy do rozwiązywania układów równań liniowych. Metoda ta polega na rozkładzie macierzy kwadratowej na dwie macierze trójkątne: dolną L (ang. *lower*) i górną U (ang. *upper*).

Proces dekompozycji LU składa się z dwóch etapów:

1. **Eliminacja w przód** - Celem jest przekształcenie macierzy współczynników układu równań do postaci trójkątnej górnej, tj. macierzy U. Proces ten polega na wprowadzeniu zer poniżej głównej przekątnej macierzy, co osiąga się poprzez odpowiednie modyfikacje wierszy macierzy. Eliminację w przód wykonujemy, obliczając współczynniki.

$$m_{ij} = \frac{a_{ij}^{k}}{a_{jj}^{k}} \tag{1}$$

   Gdzie:

   i, j są indeksami elementu, który chcemy wyzerować w macierzy

   k jest indeksem obecnego kroku w eliminacji, gdzie element a_{jj}^{k} jest elementem diagonalnym, względem którego wykonujemy eliminację w i-tym wierszu.

2. **Wypełnienie macierzy L** – Współczynniki obliczone w procesie eliminacji są używane do wypełnienia odpowiednich pozycji w macierzy L, która początkowo jest zainicjowana jako macierz jednostkowa (tożsamościowa). Każdy element jest ustawiany na wartość obliczoną podczas eliminacji dla macierzy U.

Rozkład Choleskiego jest metodą rozkładu symetrycznej, dodatnio określonej macierzy A na iloczyn postaci: $A = LL^T$, gdzie L jest dolną macierzą trójkątną, a L^T jej transpozycją, tj. macierzą górną U

Macierz L jesteśmy w stanie obliczyć poprzez zastosowanie następującego wzoru dla elementów diagonalnych:

$$l_{jj} = \sqrt{a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2} \tag{2}$$

Oraz poniższego wzoru dla elementów nie znajdujących się na diagonali:

$$l_{ij} = \frac{a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk}}{l_{jj}} \tag{3}$$

Obliczenie macierzy U jest bardzo proste wystarczy transponować otrzymaną macierz dolną L. Możemy to opisać poniższym wzorem.

$$A_{ji}^T = A_{ij} \tag{4}$$

W obu tych przypadkach dekompozycji obliczanie wyniku sprowadza się do wykonania dwóch czynności.

Podstawienie w przód – Obliczamy wartości Y poprzez przeprowadzenie podstawienia w przód na macierzy L i wektorze B.

$$x_i = \frac{b_i^n - \sum_{k=1}^{i-1} a_{ik}^n x_k}{a_{ii}^n} \tag{5}$$

Podstawienie wsteczne – Obliczamy wartości X poprzez przeprowadzenie podstawienia w przód na macierzy U i wektorze będący wynikiem z podstawienia w przód z macierzą L i wektorem B.

Podstawienie wsteczne opisane jest następującym wzorem:

$$x_i = \frac{b_i^n - \sum_{k=i+1}^{n} a_{i,k}^n x_k}{a_{ii}^n} \tag{6}$$

Implementacja

Metody dekompozycji zostały zaimplementowane w języku C++ w postaci dwóch osobnych funkcji głównych i kilku funkcji pomocniczych.

- `LU_decomposition` – metoda dekompozycji LU.
- `cholesky_decomposition`- metoda Choleskiego.

Do cech wspólnych tych funkcji głównych należą argumenty:

- Wskaźnik do macierzy liczb zmiennoprzecinkowych: `double** matrix`.
- Referencja do wskaźnika macierzy `double**& L`.
- Referencja do wskaźnika macierzy `double**& U`.
- Liczba całkowitą będąca stopniem macierzy: `int size`.

Pełna definicja funkcji `LU_decomposition`

```
void LU_decomposition(double** A, double**& L, double**& U, int size)
{
    // Initialize U as a copy of A
    U = copy_matrix(A, size);

    // Initialize L as an identity matrix
    L = identity_matrix(size);

    // Perform elimination process for each column
    for (int k = 0; k < size; k++)
        elimination(U, L, size, k);
}
```

Omówienie elementów funkcji `LU_decomposition`

Funkcja najpierw inicjalizuje macierze L i U. Macierz U przechowuje kopie macierzy początkowej A.

```
U = copy_matrix(A, size);
```

Macierz L jest zainicjowana jako macierz tożsamościowa, tj. taka która na swojej diagonali ma jedynki, pozostałe miejsca są zerami.

```
L = identity_matrix(size);
```

Następnie następuje proces eliminacji w przód dla każdej kolejnej kolumny macierzy U

```
for (int k = 0; k < size; k++)
    elimination(U, L, size, k);
```

Funkcja `copy_matrix`

```
double** copy_matrix(double** source, int size)
{
    double** copy = new double* [size];
    for (int i = 0; i < size; i++)
    {
        copy[i] = new double[size];
        for (int j = 0; j < size; j++)
            copy[i][j] = source[i][j];
    }
    return copy;
}
```

Ze względu na to, że funkcja **`copy_matrix`** jest standardową funkcją do wykonywania głębokiej kopii macierzy jej działanie nie będzie omawiane.

Funkcja `identity_matrix`

```
double** identity_matrix(int size)
{
    double** matrix = create_matrix(size);
    for (int i = 0; i < size; i++)
        matrix[i][i] = 1;

    return matrix;
}
```

Funkcja **`identity_matrix`** wykorzystuje funkcje **`create_matrix`**, która tworzy macierz kwadratową zadanego rozmiaru, a następnie wstawia wartości 1 na pozycje diagonalne.

Funkcja `create_matrix`

```
double** create_matrix(int size)
{
    double** matrix = new double* [size];
    for (int i = 0; i < size; i++)
    {
        matrix[i] = new double[size];
        for (int j = 0; j < size; j++)
            matrix[i][j] = 0;
    }

    return matrix;
```

Ze względu na to, że funkcja **`create_matrix`** jest standardową funkcją do tworzenia macierzy kwadratowej i wypełniania jej zerami, nie będzie ona dokładniej omawiana.

Funkcja `elimination`

```
void elimination(double** U, double** L, int size, int k)
{
    for (int i = k + 1; i < size; i++)
    {
        // Calculate the multiplier for the current row
        // Store it in L
        double m = U[i][k] / U[k][k];
        L[i][k] = m;

        // Update the elements of the current row in
        // U matrix
        for (int j = k; j < size; j++)
            U[i][j] -= m * U[k][j];
    }
}
```

Omówienie elementów funkcji `elimination`

Obliczany jest współczynnik wymagany do wyzerowania kolumn poniżej badanego wiersza k (wzór 1).

```
double m = U[i][k] / U[k][k];
```

Ten współczynnik jest następnie zapisywany w odpowiedniej komórce macierzy L.

```
double m = U[i][k] / U[k][k];
```

Wiersze poniżej badanej kolumny macierzy U są zerowane.

```
for (int j = k; j < size; j++)
    U[i][j] -= m * U[k][j];
```

Pełna definicja funkcji `cholesky_decomposition`

```
void cholesky_decomposition(double** A, double**& L, double**& U, int size)
{
    // Allocate memory for the lower triangular matrix L
    L = create_matrix(size);

    // Perform the Cholesky decomposition for L matrix
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j <= i; j++)
        {
            double sum = 0;

            // Diagonal elements
            if (j == i)
            {
                for (int k = 0; k < j; k++)
                    sum += L[j][k] * L[j][k];

                double diag = A[j][j] - sum;
                L[j][j] = sqrt(diag);
            }

            // Off-diagonal elements
            else
            {
                for (int k = 0; k < j; k++)
                    sum += L[i][k] * L[j][k];

                L[i][j] = (A[i][j] - sum) / L[j][j];
            }
        }
    }

    // U matrix is transpose of L
    U = copy_matrix(L, size);
    transpose_square_matrix(U, size);
}
```

Omówienie elementów funkcji `cholesky_decomposition`

Funkcja najpierw alokuje pamięć dla macierzy L i inicjalizuje ją zerami, używając wcześniej omówionej funkcji **`create_matrix`**

```
L = create_matrix(size);
```

Pętla iteruje po każdej komórce macierzy, **`i`** oznacza wiersz, **`j`** kolumnę. Wewnątrz pętli realizowane są wzory 2 i 3. Wzór drugi dla elementów diagonalnych i wzór trzeci dla pozostałych elementów.

```
double sum = 0;

// Diagonal elements
if (j == i)
{
     for (int k = 0; k < j; k++)
          sum += L[j][k] * L[j][k];

     double diag = A[j][j] - sum;
     L[j][j] = sqrt(diag);
}

// Off-diagonal elements
else
{
     for (int k = 0; k < j; k++)
               sum += L[i][k] * L[j][k];

       L[i][j] = (A[i][j] - sum) / L[j][j];
}
```

W metodzie Choleskiego macierz U otrzymujemy poprzez transponowanie macierzy L. Najpierw wykonujemy głęboką kopię, wcześniej omówioną funkcją **`copy_matrix`**.

```
U = copy_matrix(L, size);
```

Następnie dokonujemy transponowania skopiowanej macierzy z użyciem funkcji **`transpose_square_matrix`**. Funkcja ta zostanie omówiona poniżej.

```
transpose_square_matrix(U, size);
```

Pełna definicja funkcji `transpose_square_matrix`

```
void transpose_square_matrix(double** matrix, int size)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = i + 1; j < size; j++)
            std::swap(matrix[i][j], matrix[j][i]);
    }
}
```

Funkcja jest realizacją wzoru 4-tego, jej zadaniem jest transponowanie przekazanej macierzy. Ze względu na swoją względną prostotę nie będzie ona dalej omawiana.

Rozwiązywanie układu równań liniowych

Obliczanie rozwiązania jest dokładnie takie same bez względu na to, której z metod dekompozycji użyliśmy. W przypadku tej implementacji dzieje się to w ramach funkcji `run_decomposition`. Funkcja ta przyjmuje następujący zestaw argumentów

- wskaźnik do użytej metody dekompozycji `Method method`.
- macierz wejściową `double** matrix`.
- wektor wyrazów wolnych `double* vector`.
- oraz rozmiar macierzy i wektora `int size`.

Pełna definicja funkcji `run_decomposition`

```
void run_decomposition(Method method, double** matrix, double* vector, int size)
{
    // Time measurement for decomposition
    auto start = std::chrono::high_resolution_clock::now();

    // Decomposition method execution
    double** L_matrix = nullptr;
    double** U_matrix = nullptr;
    method(matrix, L_matrix, U_matrix, size);

    // Time measurement finalization
    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

    // Result calculations
    double* y_arr = forward_substitution(L_matrix, vector, size);
    double* x_arr = reverse_substitution(U_matrix, y_arr, size);

    // Prints
    std::cout << "-------------------------------------";

    // Debug prints
    //print_matrix(L_matrix, size, "\nL Matrix: ");
    //print_matrix(U_matrix, size, "\nU Matrix: ");
    //print_array(y_arr, size, "\nY Vector: ");

    // Result prints
    print_array(x_arr, size, "\nX Vector: ");
    std::cout << "Decomposition time: " << duration.count() << "\n";

    // Cleanup
    delete_matrix(L_matrix, size);
    delete_matrix(U_matrix, size);
    delete_array(y_arr);
}
```

Omówienie istotnych elementów funkcji `run_decomposition`

Deklaracja wskaźników L i U oraz wywołanie przekazanej metody dekompozycji, tj. funkcji `LU_decomposition` lub funkcji `cholesky_decomposition`.

```
double** L_matrix = nullptr;
double** U_matrix = nullptr;
method(matrix, L_matrix, U_matrix, size);
```

Wyniki są obliczane poprzez wykonanie podstawienia w przód na macierzy L oraz wektorze wyrazów wolnych – to są nasze Y-ki. X-y obliczamy poprzez podstawienie wsteczne na macierzy U oraz wektorze Y-ków. Funkcje wykonujące podstawienia zostaną omówione w dalszej części sprawozdania.

```
double* y_arr = forward_substitution(L_matrix, vector, size);
double* x_arr = reverse_substitution(U_matrix, y_arr, size);
```

Ostatnim zadaniem funkcji jest posprzątanie po sobie i zwolnienie zalokowanej pamięci

```
delete_matrix(L_matrix, size);
delete_matrix(U_matrix, size);
delete_array(y_arr);
```

Funkcja `delete_matrix`

```
void delete_matrix(double**& matrix, int size)
{
    for (int i = 0; i < size; i++)
        delete[] matrix[i];

    delete[] matrix;
    matrix = nullptr;
}
```

Funkcja `delete_array`

```
void delete_array(double*& array)
{
    delete[] array;
    array = nullptr;
}
```

Funkcja `reverse_substitution`

```
double* reverse_substitution(double** matrix, double* vector, int size)
{
    double* results = new double[size];
    for (int i = size - 1; i >= 0; i--)
    {
        double sum = vector[i];
        for (int j = i + 1; j < size; j++)
            sum -= matrix[i][j] * results[j];

        results[i] = sum / matrix[i][i];
    }

    return results;
}
```

Funkcja oblicza wyniki podstawienia wstecz dla przekazanych macierzy i wektora wyrazów wolnych, realizując wzór 6. Funkcja zwraca tablice wyników.

Funkcja `forward_substitution`

```
double* forward_substitution(double** matrix, double* vector, int size)
{
    double* results = new double[size];
    for (int i = 0; i < size; i++)
    {
        double sum = vector[i];
        for (int j = 0; j < i; j++)
            sum -= matrix[i][j] * results[j];

        results[i] = sum / matrix[i][i];
    }

    return results;
}
```

Funkcja oblicza wyniki dla podstawienia wprzód. Jej działanie jest takie same jak w przypadku funkcji **`reverse_substitution`**. Jedyną różnicą jest realizowany wzór, tj. wzór 5.

Testy na wybranych przykładach

Zaimplementowane metody zostały przetestowane z losowo wygenerowanymi macierzami, oraz porównanie z wynikami dla tych macierzy obliczonymi z wykorzystaniem biblioteki NumPy w języku Python, oraz wcześniej zaimplementowanej metody `gaussian_elimination`. Testy skupiły się na sprawdzeniu efektywności czasowej implementacji, w pracy z coraz to większymi macierzami.

Macierz 2x2

Zgodność wyników z NumPy:

- `gaussian_elimination`: Tak
- `LU_decomposition`: Tak
- `cholesky_decomposition`: Tak

Czas trwania algorytmu (mikrosekundy)

- `gaussian_elimination`: 0
- `LU_decomposition`: 1
- `cholesky_decomposition`: 2

Macierz 4x4

Zgodność wyników z NumPy:

- `gaussian_elimination`: Tak
- `LU_decomposition`: Tak
- `cholesky_decomposition`: Tak

Czas trwania algorytmu (mikrosekundy)

- `gaussian_elimination`: 0
- `LU_decomposition`: 2
- `cholesky_decomposition`: 5

Macierz 8x8

Zgodność wyników z NumPy:

- `gaussian_elimination`: Tak
- `LU_decomposition`: Tak
- `cholesky_decomposition`: Tak

Czas trwania algorytmu (mikrosekundy)

- `gaussian_elimination`: 1
- `LU_decomposition`: 3
- `cholesky_decomposition`: 6

Macierz 16x16

Zgodność wyników z NumPy:

- `gaussian_elimination`: Tak
- `LU_decomposition`: Tak
- `cholesky_decomposition`: Tak

Czas trwania algorytmu (mikrosekundy)

- `gaussian_elimination`: 5
- `LU_decomposition`: 14
- `cholesky_decomposition`: 15

Macierz 32x32

Zgodność wyników z NumPy:

- `gaussian_elimination`: Tak
- `LU_decomposition`: Tak
- `cholesky_decomposition`: Tak

Czas trwania algorytmu (mikrosekundy)

- `gaussian_elimination`: 34
- `LU_decomposition`: 51
- `cholesky_decomposition`: 42

Macierz 64x64

Zgodność wyników z NumPy:

- `gaussian_elimination`: Tak
- `LU_decomposition`: Tak
- `cholesky_decomposition`: Tak

Czas trwania algorytmu (mikrosekundy)

- `gaussian_elimination`: 254
- `LU_decomposition`: 297
- `cholesky_decomposition`: 217

Macierz 128x128

Zgodność wyników z NumPy:

- `gaussian_elimination`: Tak
- `LU_decomposition`: Tak
- `cholesky_decomposition`: Tak

Czas trwania algorytmu (mikrosekundy)

- `gaussian_elimination`: 1881
- `LU_decomposition`: 1980
- `cholesky_decomposition`: 1207

Macierz 256x256

Zgodność wyników z NumPy:

- `gaussian_elimination`: Tak
- `LU_decomposition`: Tak
- `cholesky_decomposition`: Tak

Czas trwania algorytmu (mikrosekundy)

- `gaussian_elimination`: 14558
- `LU_decomposition`: 13823
- `cholesky_decomposition`: 7425

Pozostałe testy sprawdzały tylko i wyłącznie czas pracy algorytmów. Otrzymane wyniki nie były weryfikowane.

Macierz 512x512

Czas trwania algorytmu (mikrosekundy)

- `gaussian_elimination`: 105636
- `LU_decomposition`: 108356
- `cholesky_decomposition`: 56092

Macierz 1024x1024

Czas trwania algorytmu (mikrosekundy)

- `gaussian_elimination`: 853360
- `LU_decomposition`: 876511
- `cholesky_decomposition`: 408912

Macierz 2048x2048

Czas trwania algorytmu (mikrosekundy)

- `gaussian_elimination`: 6885037
- `LU_decomposition`: 7263507
- `cholesky_decomposition`: 3437097

Macierz 4096x4096

Czas trwania algorytmu (mikrosekundy)

- `gaussian_elimination`: 55025021
- `LU_decomposition`: 65303282
- `cholesky_decomposition`: 27981733

Macierz 8192x8192

Czas trwania algorytmu (mikrosekundy)

- `gaussian_elimination`: 451171613
- `LU_decomposition`: 452804476
- `cholesky_decomposition`: 206117421

Czas pracy algorytmów został wpisany do poniższych tabelek z podziałem na rozmiar badanej macierzy. Dodatkowo, na podstawie tych danych utworzony został wykres.

Stopień macierzy		*2*	*4*	*8*	*16*	*32*
Czas (mikrosekundy)	*Gauss*	0	0	1	5	34
	LU	1	2	3	14	51
	Cholesky	2	5	6	15	42

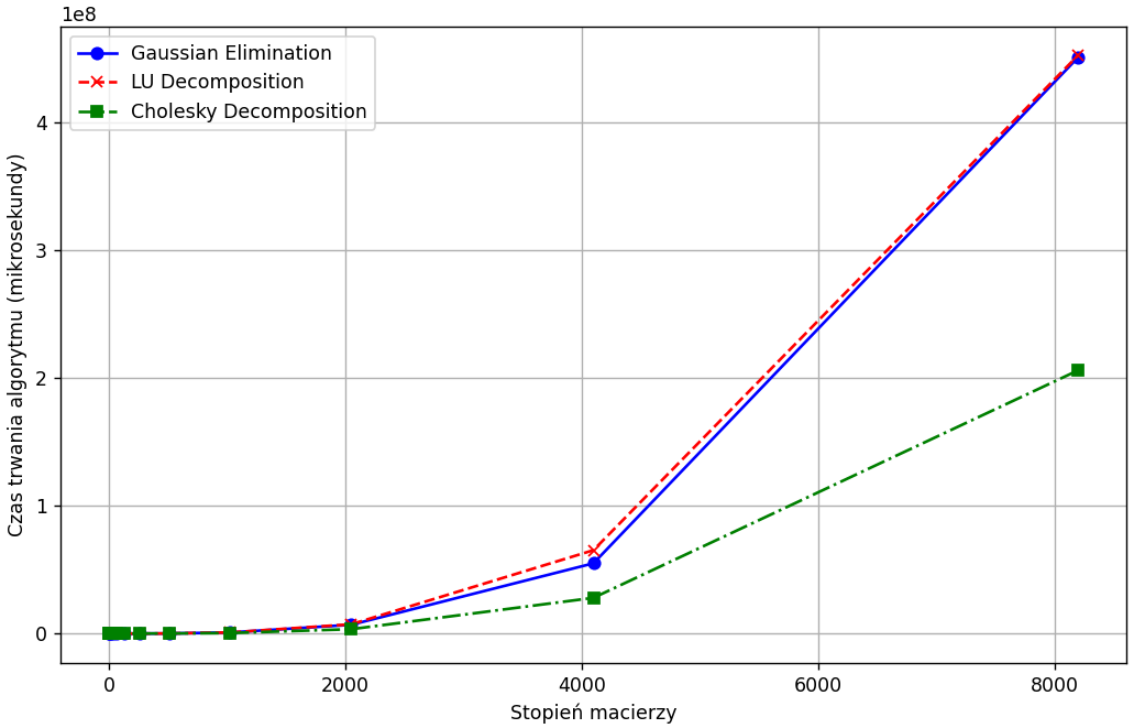
Tabela 1: Czas dla małych macierzy

Stopień macierzy		*64*	*128*	*256*	*512*
Czas (mikrosekundy)	*Gauss*	254	1881	14558	105636
	LU	297	1980	13823	108356
	Cholesky	217	1207	7425	56092

Tabela 2: Czas dla średnich macierzy

Stopień macierzy		*1024*	*2048*	*4096*	*8192*
Czas (mikrosekundy)	*Gauss*	853360	6885037	55025021	451171613
	LU	876511	7263507	65303282	452804476
	Cholesky	408912	3437097	27981733	206117421

Tabela 3: Czas dla dużych macierzy



Wykres 1: Porównanie czasu trwania algorytmu do stopnia macierzy

Analiza wyników

Małe Macierze (do 32x32):

- **Eliminacja Gaussa** oraz **dekompozycja LU** wykazują podobne czasy wykonania, z niewielką przewagą eliminacji Gaussa w niektórych przypadkach.
- **Dekompozycja Choleskiego** początkowo zajmuje więcej czasu niż pozostałe metody, co prawdopodobnie wynika z faktu, że w obecnej implementacji musi ona dwukrotnie dokonywać głębokiej kopii macierzy.

Średnie Macierze (64x64 do 512x512):

- **Dekompozycja Choleskiego** zaczyna wykazywać znaczącą przewagę nad pozostałymi metodami, co sugeruje wyższą efektywność przy większym obciążeniu obliczeniowym.

Duże Macierze (1024x1024 i większe):

- **Dekompozycja Choleskiego** systematycznie osiąga lepsze czasy wykonania, nawet dwukrotnie szybsze w przypadku dużych macierzy. Dla macierzy 8192x8192 czas wyniósł 206 sekund, co stanowi zaledwie 46% czasu drugiej najszybszej metody.
- Skalowalność **dekompozycji Choleskiego** jest znacznie lepsza, co czyni ją preferowaną metodą w przypadku dużych systemów równań.
- Metoda dekompozycji LU nadal osiąga gorsze wyniki niż metoda eliminacji Gaussa, jednak różnica pomiędzy nimi staje się mniej zauważalna.

Podsumowanie

Rezultaty jasno wskazują na przewagę **dekompozycji Choleskiego** w kontekście czasowym, zwłaszcza dla większych macierzy. Wydaje się być ona korzystna dla zastosowań, które wymagają częstego rozwiązywania układów równań, gdzie macierze są symetryczne i dodatnio określone. W przypadku, gdy wymagana jest większa elastyczność odnośnie akceptowalnych danych wejściowych, **metoda Eliminacji Gaussa** okazała się być nieznacznie lepsza od metody **Dekompozycji LU** w przeprowadzonych testach.