Metdy numeryczne – projekt 2
„Układy równań liniowych”

Michał Soja

175793
Informatyka
semestr 4
grupa 5

15.05.2022r

1. **Wstęp**

   Projekt polega na stworzeniu programu rozwiązujące większych rozmiarów układy równań.

   W moim projekcie został wykorzystany język C++, a rysowanie wykresów zostało wykonane za pomocą programu LibreOffice Calc.

2. **Implementacja**

   Program napisany został w oparciu o klasę *Matrix* reprezentującą macierz.

```
typedef double containerType;

class Matrix
{
public:
    Matrix(int size_x, int size_y);
    Matrix(Point size);
    Matrix(int size);
    ~Matrix();

    Point GetSize() const;
    std::shared_ptr<Matrix> GetCopy();
    void Fill(containerType value);
    void SetDiagonal(containerType value);

    containerType*& operator[](int index);
private:
    void allocateMemory();

    containerType** container;
    Point size;
};

using MatrixPtr = std::shared_ptr<Matrix>;

MatrixPtr operator*(Matrix& left, Matrix& right);
MatrixPtr operator-(Matrix& left, Matrix& right);
```

   Jako konstruktor klasa przyjmuje rozmiar i alokuje pamięć na elementy. Elementy są typu *containerType*, aby w każdej chwili móc zmienić typ double ukryty pod nim na inny. Klasa również pozwala na podstawowe operacje na macierzach takie jak dodawanie i odejmowanie poprzez przeciążenie operatorów. Dodatkowo zawiera opcje wypełnienia całej macierzy konkretną wartością lub wypełnienie diagonali konkretną wartością. Funkcja *GetCopy()* zwraca nam wskaźnik na obiekt tej klasy zawierający te same dane co obiekt na którym została wywołana ta metoda. W destruktorze zwalniana jest pamięć kontenera.

Podpunkt A)

W tym podpunkcie należało utworzyć macierze A oraz wektory x oraz b. Ich rozmiar oraz wypełnienie miało zależeć od numeru indeksu autora. Jest to realizowane przez makra na początku programu przechowujące indeks, w dalszych częściach programu wykorzystywane są makra ID1, ID2, … ID6 odpowiadające kolejnym cyfrom.

```
#define ID1 1
#define ID2 7
#define ID3 5
#define ID4 7
#define ID5 9
#define ID6 3
```

Tworzenie macierzy A polega na wpisaniu wartości a_1 na diagonali, a_2 na drugiej diagonali oraz a_3 na trzeciej diagonali. Macierz po prawidłowym wypełnieniu powinna mieć rozmiar 993x993 oraz wyglądać następująco:

$$\begin{bmatrix} 12 & -1 & -1 & 0 & 0 & \cdots & 0 \\ -1 & 12 & -1 & -1 & 0 & \cdots & 0 \\ -1 & -1 & 12 & -1 & -1 & \cdots & 0 \\ 0 & -1 & -1 & 12 & -1 & \cdots & 0 \\ 0 & 0 & -1 & -1 & 12 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & 12 \end{bmatrix}$$

zrealizowane w poniżej pokazany sposób:

```
MatrixPtr Create_A(int size, containerType a1, containerType a2, containerType a3) {
    auto result = std::make_shared<Matrix>(size);

    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            if (i == j) {
                (*result)[i][j] = a1;
            }
            else if (i + 1 == j || i - 1 == j) {
                (*result)[i][j] = a2;
            }
            else if (i + 2 == j || i - 2 == j) {
                (*result)[i][j] = a3;
            }
            else {
                (*result)[i][j] = 0;
            }
        }
    }
    return result;
}
```

```
MatrixPtr Create_x(int size) {
    auto result = std::make_shared<Matrix>(1, size);
    result->Fill(1);
    return result;
}
```

Wektor b natomiast jest wyliczony ze wzoru $a_n = sin(n * (ID3 + 1))$.

```
MatrixPtr Create_b(int size) {
    auto result = std::make_shared<Matrix>(1, size);

    for (int n = 0; n < size; n++)
    {
        (*result)[0][n] = sin(n * (ID3 + 1));
    }
    return result;
}
```

Podpunkt B)

Dla podanej macierzy należało znaleźć taki wektor x, który będzie spełniał równanie A * x = b. Wykorzystujemy do tego dwie metody: Jacobiego oraz Gaussa-Seidla. Oprócz tego używamy funkcji wypisujących wyniki i tworzących macierze. Jako parametry metody te przyjmują wskaźniki na macierz A i wektory x, b oraz wartość wymaganej normy z residuum.

```
MatrixPtr CalculateResiduum(MatrixPtr A, MatrixPtr B, MatrixPtr X) {
    auto result = (*A) * (*X);
    result = (*result) - (*B);
    return result;
}
containerType CalculateNorm(MatrixPtr residuum) {
    containerType result = 0;
    for (int i = 0; i < residuum->GetSize().y; i++)
    {
        result += pow((*residuum)[0][i], 2);
    }
    return sqrt(result);
}
containerType CalculateNormOfResiduum(MatrixPtr A, MatrixPtr B, MatrixPtr X) {
    auto residuum = CalculateResiduum(A, B, X);
    return CalculateNorm(residuum);
}
```

Metoda Jacobiego

Metoda ta na początku tworzy pomocniczy wektor lastX który przyjmuje wartości 1. Następnie w dwóch pętlach for wyliczamy nowy wektor X korzystając ze wzoru:

$$\begin{cases} x_1^{(k+1)} &= (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)})/a_{11} \\ x_2^{(k+1)} &= (b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)})/a_{22} \\ x_3^{(k+1)} &= (b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)})/a_{33} \end{cases}$$

w każdej iteracji zwiększamy licznik iteracji oraz liczymy normę z residuum. Czynności powtarzamy tak długo aż norma będzie mniejsza niż zadana jako parametr funkcji. Po spełneniu tego warunku zwracamy wynik funkcji składający się z czasu trwania, ilości iteracji oraz normy residuum.

Metoda Gaussa-Seidla

Metoda działa podobnie do metody Jacobiego. Różnią się jedynie sposobem wyliczania nowego wektora X. W tej metodzie wykorzystujemy wektor lastX oraz X do wyliczenia kolejnych elementów w tej samej iteracji, w odróżnieniu od metody Jacobiego.

```cpp
Result Jacobiego(MatrixPtr A, MatrixPtr B, MatrixPtr X, const double NORM) {
    Timer timer;
    timer.Start();

    Result result;

    auto lastX = std::make_shared<Matrix>(X->GetSize());
    lastX->Fill(1);

    do
    {
        for (int y = 0; y < X->GetSize().y; y++)
        {
            (*X)[0][y] = (*B)[0][y];
            for (int x = 0; x < A->GetSize().x; x++)
            {
                if (x == y) {
                    continue;
                }
                (*X)[0][y] -= (*A)[x][y] * (*lastX)[0][x];
            }
            (*X)[0][y] /= (*A)[y][y];
        }
        lastX = X->GetCopy();
        result.iterations++;

        result.norm = CalculateNormOfResiduum(A, B, X);
    } while (result.norm > NORM);

    timer.Stop();
    result.time = timer.GetTime();
    return result;
}
```

```cpp
Result GaussaSeidla(MatrixPtr A, MatrixPtr B, MatrixPtr X, const double NORM) {
    Timer timer;
    timer.Start();

    Result result;

    auto lastX = std::make_shared<Matrix>(X->GetSize());
    lastX->Fill(1);

    do
    {
        for (int i = 0; i < X->GetSize().y; i++)
        {
            (*X)[0][i] = (*B)[0][i];
            for (int j = 0; j < A->GetSize().x; j++)
            {
                if (j < i) {
                    (*X)[0][i] -= (*A)[i][j] * (*X)[0][j];
                }
                else if (j > i) {
                    (*X)[0][i] -= (*A)[i][j] * (*lastX)[0][j];
                }
            }
            (*X)[0][i] /= (*A)[i][i];
        }
        lastX = X->GetCopy();
        result.iterations++;

        result.norm = CalculateNormOfResiduum(A, B, X);
    } while (result.norm > NORM);

    timer.Stop();
    result.time = timer.GetTime();

    return result;
}
```

Podpunkt C)

W tym zadaniu należy obliczyć układ równań dla a1 = 3, a2 = -1 i a3 = -1. Niestety wykorzystując metody iteracyjne nie osiągniemy wyniku, gdyż wartości nie zbiegają do niego. Ta metoda dla tych danych jest rozbieżna, w kolejnych iteracjach wektor X oddala się od docelowego.

```
void ZadanieC() {
    printf("Zadanie C\n");

    MatrixPtr A, b, x;
    CreateData data;
    Result result;

    data.N = 9 * 100 + ID5 * 10 + ID6;
    data.a1 = 3;
    data.a2 = -1;
    data.a3 = -1;

    PrintMethod("Jacobi", data);
    PrepareMatrixes(data, A, b, x);
    result = Jacobiego(A, b, x, 1e-9);
    PrintResult(result);

    PrintMethod("Gauss-Seidl", data);
    PrepareMatrixes(data, A, b, x);
    result = GaussaSeidla(A, b, x, 1e-9);
    PrintResult(result);
}
```

Podpunkt D)

Aby rozwiązać problem w podpunkcie C trzeba wykorzystać bezpośrednią metode rozwiązywania układów liniowych. Zaletą jest możliwość rozwiązania każdego układu, minusem zaś dużo większa złożoność obliczeniowa w porównaniu do rozwiązań iteracyjnych. W moim projekcie wykorzystałem faktoryzacje LU. Dzięki temu udało się uzyskać wynik. Norma residuum spełnia również warunek i jest mniejsza niż wymagana. W idealnej sytuacji powinna być ona równa 0, pojawia się ona w związku z zaokrągleniami. Metoda ta na początku tworzy macierze L oraz U, które wymnożone przez siebie dadzą macierz A. Następnie obliczane są równania L * y = b oraz U * x = y.

```
Result LU(MatrixPtr A, MatrixPtr B, MatrixPtr X) {
    Timer timer;
    timer.Start();

    Result result;

    auto size = A->GetSize().x;

    //PrepareMatrixes U & L Matrixes
    auto U = A->GetCopy();
    auto L = std::make_shared<Matrix>(A->GetSize());
    L->Fill(0);
    L->SetDiagonal(1);

    for (int k = 0; k < size - 1; k++)
    {
        for (int j = k + 1; j < size; j++)
        {
            (*L)[j][k] = (*U)[j][k] / (*U)[k][k];

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

```
    auto Y = std::make_shared<Matrix>(1, size);

    //L * y = b
    for (int i = 0; i < size; i++)
    {
        (*Y)[0][i] = (*B)[0][i];

        for (int j = 0; j < i; ++j)
            (*Y)[0][i] -= (*L)[i][j] * (*Y)[0][j];

        (*Y)[0][i] = (*Y)[0][i] / (*L)[i][i];
    }
    //U * x = y
    for (int i = size - 1; i >= 0; --i)
    {
        (*X)[0][i] = (*Y)[0][i];
        for (int j = i + 1; j < size; ++j)
            (*X)[0][i] -= (*U)[i][j] * (*X)[0][j];

        (*X)[0][i] = (*X)[0][i] / (*U)[i][i];
    }

    result.norm = CalculateNormOfResiduum(A, B, X);

    timer.Stop();
    result.time = timer.GetTime();
    result.iterations = 1;

    return result;
}
```

Podpunkt E)

Na koniec aby zobaczyć różnice między tymi metodami wykonywane są one dla macierzy o rozmiarze N = 100, 500, 1000, 2000, 3000. Zostało to zaimplementowane w pętli wykonując te trzy metody na zmianę i zapisane do pliku o rozszerzeniu csv, aby za pomocą programu LibreOffice Calc wyrysować wykres.

```
data.N = 9 * 100 + ID5 * 10 + ID6;
int N[] = { 100, 500, 1000, 2000, 3000 };

outputFile << "N,Jacobi,GaussSeidl,LU";
for (int i = 0; i < sizeof N / sizeof N[0]; i++)
{
    data.N = N[i];
    outputFile << data.N << ",";
    PrintMethod("Jacobi", data);
    PrepareMatrixes(data, A, b, x);
    result = Jacobiego(A, b, x, 1e-9);
    outputFile << result.time.min << ":" << result.time.sec << ":" << result.time.ms << ",";
    PrintResult(result);

    PrintMethod("Gauss-Seidl", data);
    PrepareMatrixes(data, A, b, x);
    result = GaussaSeidla(A, b, x, 1e-9);
    outputFile << result.time.min << ":" << result.time.sec << ":" << result.time.ms << ",";
    PrintResult(result);

    PrintMethod("LU", data);
    PrepareMatrixes(data, A, b, x);
    result = LU(A, b, x);
    outputFile << result.time.min << ":" << result.time.sec << ":" << result.time.ms << "\n";
    PrintResult(result);
}
outputFile.close();
}
```

3. Wyniki

```
Microsoft Visual Studio Debug Console
Zadanie B
        Jacobi Method (N=993, a1=12, a2=-1, a3=-1)
                iterations: 24
                norm of residuum: 8.856464e-10
                time: 0min 2sec 931ms

        Gauss-Seidl Method (N=993, a1=12, a2=-1, a3=-1)
                iterations: 17
                norm of residuum: 3.271765e-10
                time: 0min 1sec 821ms
```

```
Microsoft Visual Studio Debug Console
Zadanie D
        LU Method (N=993, a1=3, a2=-1, a3=-1)
                iterations: 1
                norm of residuum: 2.518577e-12
                time: 0min 17sec 329ms
```

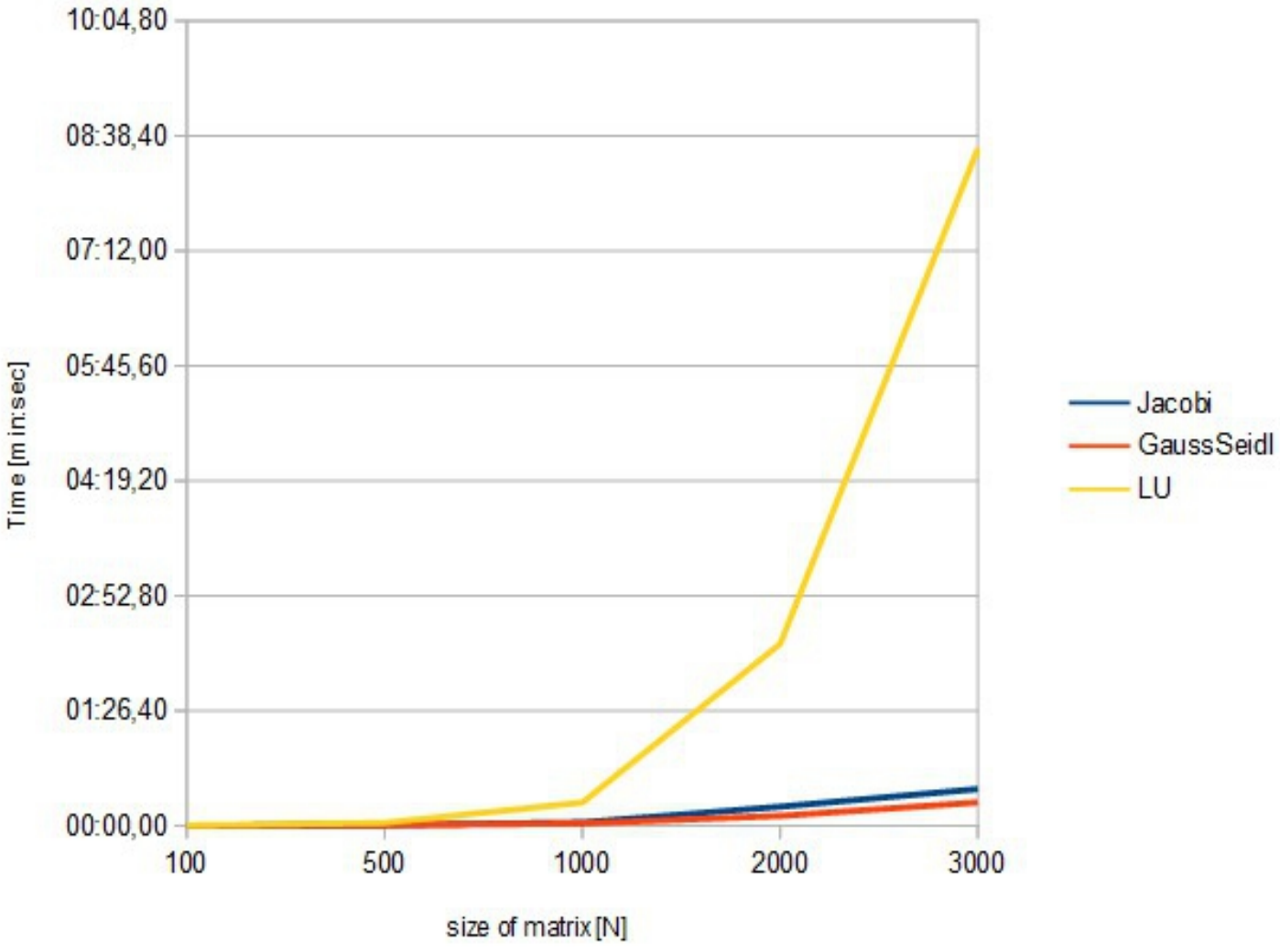
```
Microsoft Visual Studio Debug Console
Zadanie E
        Jacobi Method (N=100, a1=12, a2=-1, a3=-1)
                iterations: 23
                norm of residuum: 7.820622e-10
                time: 0min 0sec 19ms

        Gauss-Seidl Method (N=100, a1=12, a2=-1, a3=-1)
                iterations: 16
                norm of residuum: 4.679273e-10
                time: 0min 0sec 13ms

        LU Method (N=100, a1=12, a2=-1, a3=-1)
                iterations: 1
                norm of residuum: 1.341434e-15
                time: 0min 0sec 19ms

        Jacobi Method (N=500, a1=12, a2=-1, a3=-1)
                iterations: 24
                norm of residuum: 6.234153e-10
                time: 0min 0sec 544ms

        Gauss-Seidl Method (N=500, a1=12, a2=-1, a3=-1)
                iterations: 17
                norm of residuum: 2.296224e-10
                time: 0min 0sec 372ms
```



4. Wnioski

Metoda Jacobiego w najgorszym przypadku (dla N = 3000) wyniosła około 30sek, a Gaussa-Seidla 15sek, z czego wynika że metoda Gaussa-Seidla jest około dwukrotnie szybsza. Wykresy obu metod są w przybliżeniu liniowe. Natomiast metoda faktoryzacji LU wyniosła aż 8,5min, a czas wykonywania rośnie wykładniczo. Wyniki otrzymane jasno wskazują, że dla większych rozmiarów macierzy metody iteracyjne znacząco przewyższają faktoryzacje LU, a różnica między tymi metodami rośnie wraz ze zwiększaniem rozmiaru macierzy.