Metody Numeryczne

Projekt 2 – Układy równań liniowych

Krzysztof Nazar, 184698

16 kwietnia 2022

1. Wstęp

Celem projektu jest implementacja metod iteracyjnych (Jacobiego i Gaussa-Seidla) i bezpośrednich (faktoryzacja LU) rozwiazywania układów równań liniowych. Układy równań są kluczowe w dalszym rozwoju nauki w obszarze wielu dziedzin, na przykład biomechanika, symulacje odkształceń, dynamika płynów i wiele innych. Mimo, że w praktyce stosuje się macierze przechowywane w tak zwanym rzadkim formacie, w tym projekcie będę stosował macierz zapisaną w formacie pełnym.

Implementacja metod rozwiązywania układów równań została wykonana w języku C++ w środowisku Visual Studio. Wykresy zostały stworzone z wykorzystaniem programu MS Excel.

2. Podstawa teoretyczna

2.1 Konstrukcja układu równań

Układ równań liniowych przedstawiony jest formułą:

$$Ax = b$$

gdzie:

A – macierz systemowa, np. obwód elektroniczny, karoserię samochodu, turbinę, itp.

x – wektor pobudzenia, np. impuls elektroniczny, wektor siły, fala dźwiękowa itp.

b – wektor rozwiązań reprezentujących szukaną wielkość fizyczną, np. rozkład pola w przestrzeni, natężenie dźwięku itp.

2.2 Wektor residuum

Podczas wykorzystywania algorytmów iteracyjnych istotne jest określenie iteracji, w której algorytm powinien przestać się wykonywać. W tym celu korzysta się z tak zwanego wektora residuum. Dla k-tej iteracji wektor residuum można wyznaczyć za pomocą poniższego wzoru:

$$res^{(k)} = Ax^{(k)} - b$$

Aby obliczyć jaki błąd wnosi wektor $x^{(k)}$ należy wyznaczyć normę euklidesową z wektora residuum w k-tej iteracji. Wektor powinien być wektorem zerowym, jeśli algorytm zbiegnie do dokładnego rozwiązania. Zwykle jako kryterium stopu przyjmuje się normę euklidesową z wektora residuum o wartość mniejszej niż 10^{-6}.

3. Zadanie projektowe

3.1 Zadanie A

W tym zadaniu macierz A jest macierzą kwadratową o rozmiarze $N \times N$ gdzie $N = 998$. Macierz A składa się z wyrazów a_1, a_2 oraz a_3, gdzie

$$a_1 = 5 + 4 = 9$$

$$a_2 = a_3 = -1$$

Macierz A przedstawiona została poniżej.

$$A = \begin{bmatrix} 9 & -1 & -1 & 0 & 0 & 0 & 0 & \dots & 0 \\ -1 & 9 & -1 & -1 & 0 & 0 & 0 & \dots & 0 \\ -1 & -1 & 9 & -1 & -1 & 0 & 0 & \dots & 0 \\ 0 & -1 & -1 & 9 & -1 & -1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 & 0 & -1 & -1 & 9 \end{bmatrix}$$

Wektor b ma długość N. Jego n-ty element ma wartość

$$b_n = \sin(n \cdot (4 + 1)) = \sin(5 \cdot n)$$

Wektor b przedstawiony został poniżej.

$$b = \begin{bmatrix} \sin(5 \cdot 0) \\ \sin(5 \cdot 1) \\ \sin(5 \cdot 2) \\ \sin(5 \cdot 3) \\ \vdots \\ \sin(5 \cdot 997) \end{bmatrix}$$

3.2 Zadanie B

Celem tego zadania była implementacja metod iteracyjnych rozwiązywania układów równań liniowych: Jacobiego i Gaussa–Seidla.

Podczas obliczeń przy pomocy obydwu metod wykorzystuje funkcje wyznaczające wartość residuum oraz normę wektora. Ich implementacje umieściłem poniżej.

```
double* calcResiduum(Matrix A, Vector b, Vector x) {
	Vector c = A.matrixMultiplicationVector(x.valuesArr);
	for (int i = 0; i < A.getSize(); i++) {
		c.valuesArr[i] -= b.valuesArr[i];
	}
	return c.valuesArr;
}


double vectorEuclNorm(int size,double* x) {
	double tmp = 0;
	for (int i = 0; i < size; i++) {
		tmp += (x[i] * x[i]);
	}
	return sqrt(tmp);
}
```

Najpierw zaimplementowałem metodę Jacobiego.

```
double* runJacobiFormula(Matrix* A, Vector* b, double* x1) {
	int size = A->getSize();
	double* x = new double[size];
	double sum = 0;
	for (int i = 0; i < size; i++) {
		sum = 0;
		for (int j = 0; j < size; j++) {
			if (i != j)sum += A->valuesArr[i][j] * x1[j];
		}
		sum = b->valuesArr[i] - sum;
		x[i] = sum / A->valuesArr[i][i];
	}
	return x;
}
void solveJacobi(Matrix* A, Vector* b, CSVWriter *csv = NULL) {
	int size = A->getSize();
	clock_t begin, finish;
	double final_time;
	begin = clock();

	int iterCounter = 0;
	double currRes = 0, finalRes = pow(10, -9);
	Vector* x = new Vector(size, 1);

	while (true) {
		iterCounter++;
		x->valuesArr = runJacobiFormula(A, b, x->valuesArr);
		currRes = vectorEuclNorm(A->getSize(),calcResiduum(*A, *b, *x));
		if (currRes < finalRes || iterCounter == 5000)
			break;
		//cout << iterCounter << "\n";
	}
	finish = clock();
	final_time = (double)(finish - begin);

	equResult* result = new equResult("Jacobi", currRes, iterCounter, final_time);
	result->printResult();
	cout << size << "\t" << final_time << endl;
	if(csv != NULL)
	csv->newRow() << size << final_time;

}
```

Następnie zaimplementowałem metodę Gaussa-Seidla.

```
double* runGaussSeidlFormula(Matrix* A, Vector* b, double* x1) {
	int size = A->getSize();

	double sum = 0;
	for (int i = 0; i < size; i++) {
		sum = 0;
		for (int j = 0; j < i; j++) {
			sum += A->valuesArr[i][j] * x1[j];
		}
		for (int j = i + 1; j < size; j++) {
			sum += A->valuesArr[i][j] * x1[j];
		}
		sum = b->valuesArr[i] - sum;
		x1[i] = sum / A->valuesArr[i][i];
	}
	return x1;
}
void solveGaussSeidl(Matrix* A, Vector* b, CSVWriter* csv = NULL) {
	clock_t begin, finish;
	double final_time;
	begin = clock();

	int size = A->getSize();
	int iterCounter = 0;
	double currRes = 0, finalRes = pow(10, -9);
	Vector* x = new Vector(size, 1);

	while (true) {
		iterCounter++;
		x->valuesArr = runGaussSeidlFormula(A, b, x->valuesArr);
		currRes = vectorEuclNorm(A->getSize(),calcResiduum(*A, *b, *x));
		if (currRes < finalRes|| iterCounter == 5000)
			break;
	}
	finish = clock();
	final_time = (double)(finish - begin);

	equResult* result = new equResult("Gauss-Seidl", currRes, iterCounter,
final_time);
	result->printResult();
	//cout << size << "\t" << final_time << endl;

	if (csv != NULL)
		csv->newRow() << size << final_time;
}
```

Program zwraca poniższe informacje.

```
Method: Jacobi: Norm: 8.63803e-10.      Iterations: 23. Time: 166 miliseconds.

Method: Gauss-Seidl:    Norm: 4.12187e-10.      Iterations: 17. Time: 129 miliseconds.
```

Rozwiązanie układu równań uzyskuje się szybciej przy użyciu metody Gaussa-Seidla. Istotny jest także fakt, że przy użyciu metody Jacobiego potrzebna jest większa liczba iteracji aby obliczyć wynik. W tym przypadku, różnice pomiędzy metodami są mało znaczące.

3.3 Zadanie C

W tym zadaniu macierz A składa się z wyrazów a_1, a_2 oraz a_3, gdzie

$$a_1 = 3$$

$$a_2 = a_3 = -1$$

Macierz A przedstawiona została poniżej.

$$A = \begin{bmatrix} 3 & -1 & -1 & 0 & 0 & 0 & 0 & \dots & 0 \\ -1 & 3 & -1 & -1 & 0 & 0 & 0 & \dots & 0 \\ -1 & -1 & 3 & -1 & -1 & 0 & 0 & \dots & 0 \\ 0 & -1 & -1 & 3 & -1 & -1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 & 0 & -1 & -1 & 3 \end{bmatrix}$$

Wartości w wektorze b o długości N nie zmieniają się.

$$b = \begin{bmatrix} \sin(5 \cdot 0) \\ \sin(5 \cdot 1) \\ \sin(5 \cdot 2) \\ \sin(5 \cdot 3) \\ \vdots \\ \sin(5 \cdot 997) \end{bmatrix}$$

W funkcjach obliczających rozwiązanie układu równań zgodnie z metodą Jacobiego oraz Gaussa-Seidlera wprowadziłem warunek, aby kończyły się gdy liczba iteracji osiągnie 5000. Dzięki temu można uniknąć nieskończonej pętli w przypadku gdy układ nie ma dokładnego rozwiązania.

```
Method: Jacobi: Norm: -nan(ind).        Iterations: 5000.       Time: 36274 miliseconds.

Method: Gauss-Seidl:    Norm: -nan(ind).        Iterations: 5000.       Time: 32358 miliseconds.
```

Metody iteracyjne dla analizowanego układu nie zbiegają się. Na konsoli widoczne jest, że norma wektora osiąga wartość *-nan(ind)*. Skrót *NAN* w języku angielskim oznacza „Not A Number”, a więc nie otrzymaliśmy dokładnego rozwiązania. Można z tego wyciągnąć wniosek, że pętla została przerwana przez warunek sprawdzający ilość wykonanych iteracji.

3.4 Zadanie D

Do wyznaczenia rozwiązania układu równań została wykorzystana implementacja metody faktoryzacji LU. Kod umieszczam poniżej.

```
void LU(Matrix* L, Matrix* U) {
	int size = L->getSize();
	for (int k = 0; k < size - 1; k++) {
		for (int j = k + 1; j < size; j++) {
			L->valuesArr[j][k] = U->valuesArr[j][k] / U->valuesArr[k][k];

			for (int r = k; r < size; r++) {
				U->valuesArr[j][r] = U->valuesArr[j][r] - L->valuesArr[j][k] *
U->valuesArr[k][r];
			}
		}
	}
}

void solveLUFactorisation(Matrix *A, Vector *b) {
	clock_t begin, finish;
	double final_time;
	begin = clock();
	int size = A->getSize();
	Matrix* U = new Matrix(size, 3, -1, -1);
	Matrix* L = new Matrix(size, 1, 0, 0);
	LU(L, U);
	Vector* y = new Vector(size, 0);

	for (int i = 0; i < size; i++) {
		double sum = 0;
		for (int j = 0; j < i; j++) {
			sum += L->valuesArr[i][j] * y->valuesArr[j];
		}
		y->valuesArr[i] = b->valuesArr[i] - sum;
	}
	double* x = new double[size];
	for (int i = 0; i < size; i++) {
		x[i] = 0;
	}

	for (int i = size - 1; i >= 0; i--) {
		double sum = 0;
		for (int j = i; j < size; j++) {
			sum += U->valuesArr[i][j] * x[j];
		}
		x[i] = (y->valuesArr[i] - sum) / U->valuesArr[i][i];
	}
	Vector* z = new Vector(size, 0);
	z->valuesArr = x;
	double res = vectorEuclNorm(A->getSize(), calcResiduum(*A, *b, *z));
	finish = clock();
	final_time = (double)(finish - begin) / CLOCKS_PER_SEC;

	equResult *result = new equResult();
	result->printResult("LUFact", res, final_time);
}
```

Analizowane były macierze takie same jak w zadaniu C, w którym metody iteracyjne nie zbiegły się. Jednak używając metody faktoryzacji LU otrzymaliśmy dokładny wynik.

```
Method: LUFact: Result: 6.1141e-13.     Time: 1.554 miliseconds.
```

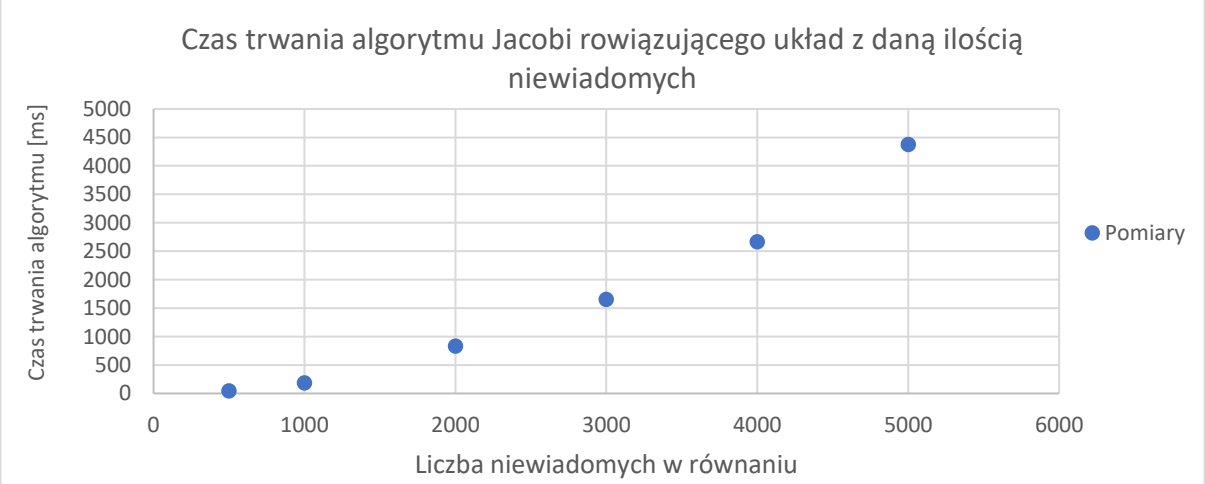
Wartość residuum osiągnęła wartość około $6{,}11 \cdot 10^{-13}$. Czas wykonywania algorytmu był relatywnie krótki – około 1,5 sekundy.

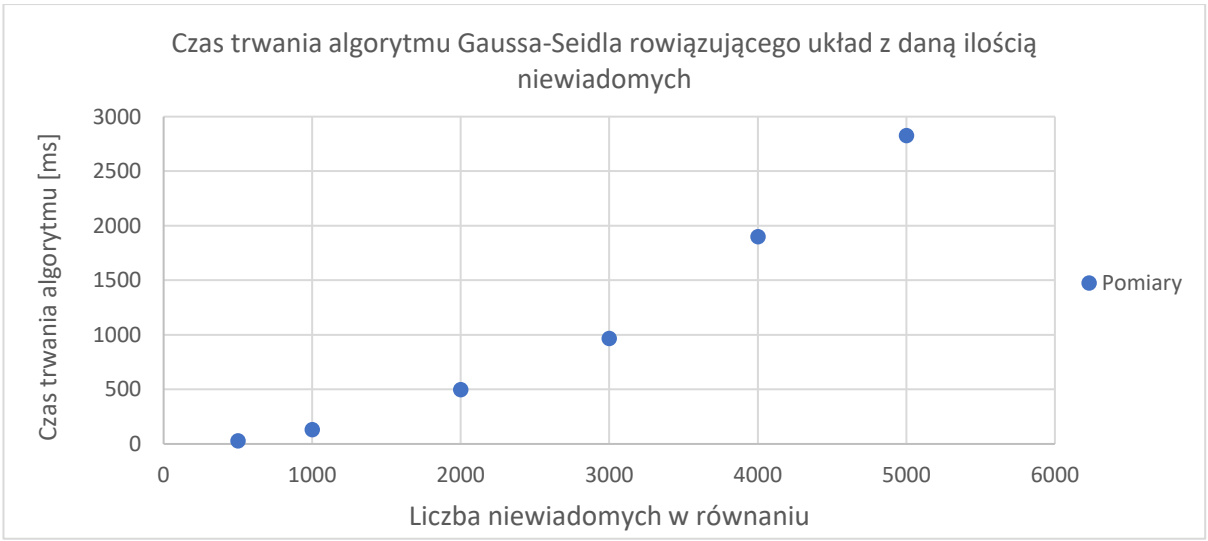
3.5 Zadanie E

Stworzyłem tablice *NArray* zawierającą kilka liczb całkowitych – są to kolejno analizowane rozmiary macierzy A.

```
int NArray[] = { 500, 1000, 2000, 3000, 4000, 5000};
```

Dzięki klasie *CSVWriter* informacje o ilości niewiadomych w danym przypadku oraz czasie rozwiązywania układu równań zapisywane są do plików CSV: *JacobiTimeVsSize.csv* oraz *GaussSeidlTimeVsSize.csv*. Na podstawie uzyskanych danych, w programie Microsoft Excel stworzyłem w wykresy przedstawiające zależności pomiędzy rozmiarem macierzy a czasem rozwiązywania układu równań. Wykresy zamieściłem poniżej.





3.6 Zadanie F - Wnioski

Po wykonaniu zadań nie potrafię stwierdzić która metoda jest najlepsza. Zadanie B pokazuje, że metody iteracyjne wykonują się bardzo szybko. Jednak zadania D oraz E ilustrują przypadek gdy metoda bezpośrednia ma znaczącą przewagę nad metodami iteracyjnymi. Uważam, że równania powinno się najpierw rozwiązywać metodami iteracyjnymi, a jeśli nie będą zbiegać, wtedy zastosować metodę bezpośrednią, na przykład metodę faktoryzacji LU. Na podstawie wyników funckji „Trendilne" w MS Excel, można powiedzieć, że zależność pomiędzy czasem wykonywania algorytmu a liczba niewiaodmych w równaniu rośnie potęgowo.