Łukasz Merynda

Metody Obliczeniowe

Projekt zaliczeniowy 11-4

1. Wstęp

Zadanie polegało na rozwiązaniu i analizie równania różniczkowego cząstkowego

$$\frac{\partial U(x,t)}{\partial t} = D\frac{\partial^2 U(x,t)}{\partial x^2}$$

równanie zostało określone dla $x \in [0,+\infty], t \in [0,2]$ z następującymi warunkami brzegowymi i początkowym:

$U(x,0)=1, U(0,t)=0, U(+\infty,t)=1$

Rozwiązanie analityczne równania ma postać: (erf to tzw. funkcja błędu)

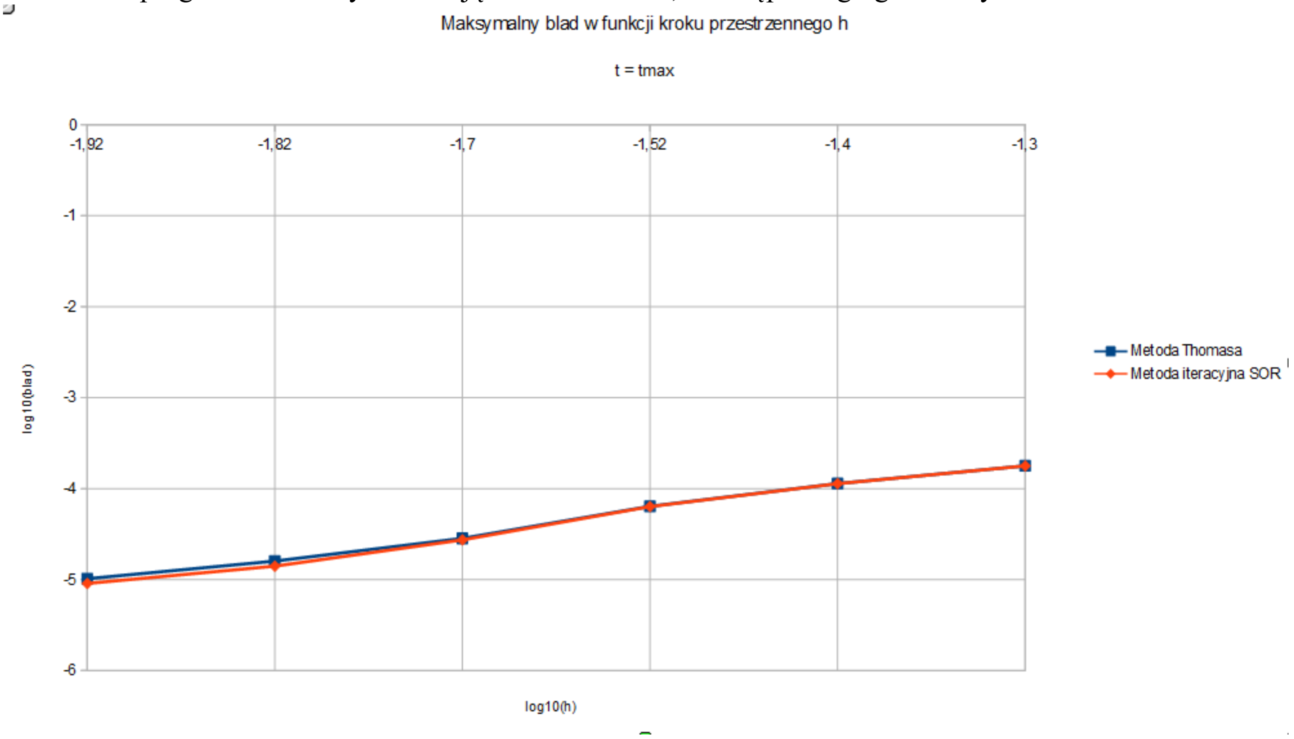
$$U(x,t)=erf\left(\frac{x}{2\sqrt{Dt}}\right)$$

Problem został rozwiązany z użyciem następujących metod:

- Dyskretyzacja – Metoda pośrednia Laasonen
- Rozwiązanie algebraiczne układów równań – Algorytm Thomasa, Metoda iteracyjna SOR

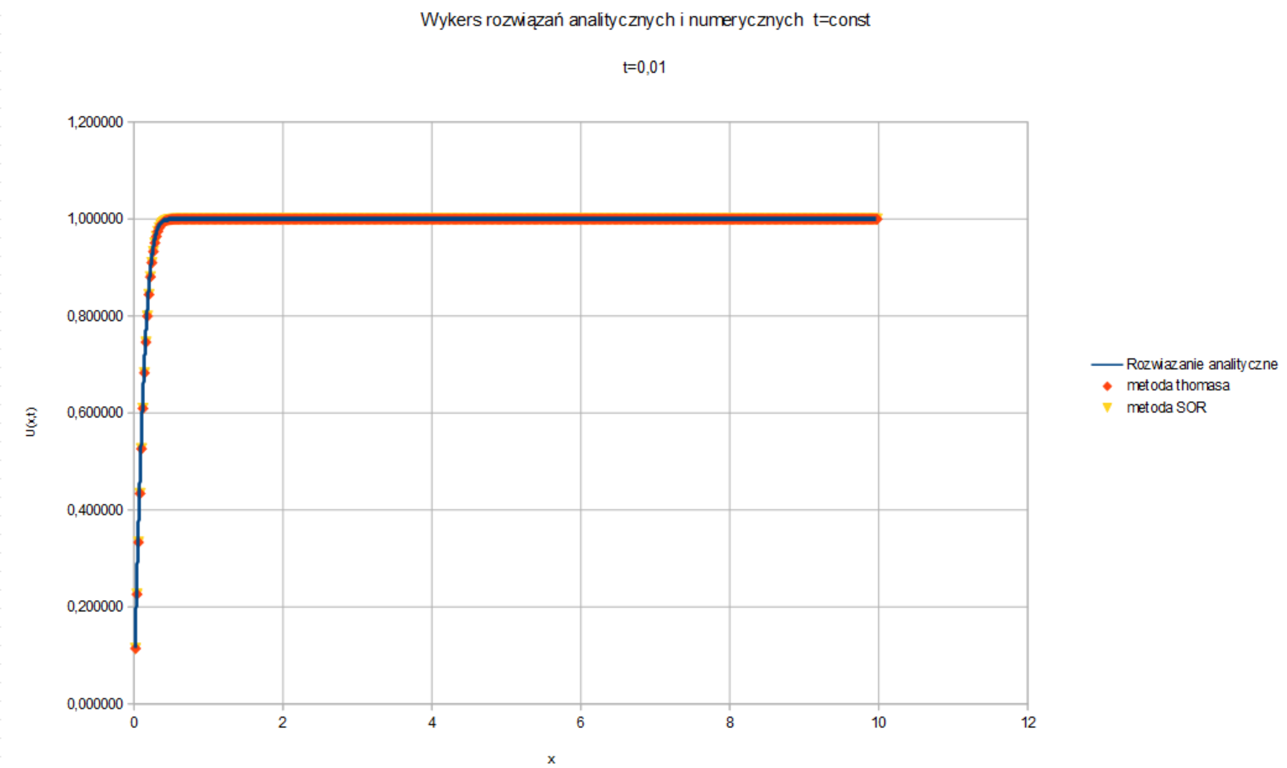
Zad 1

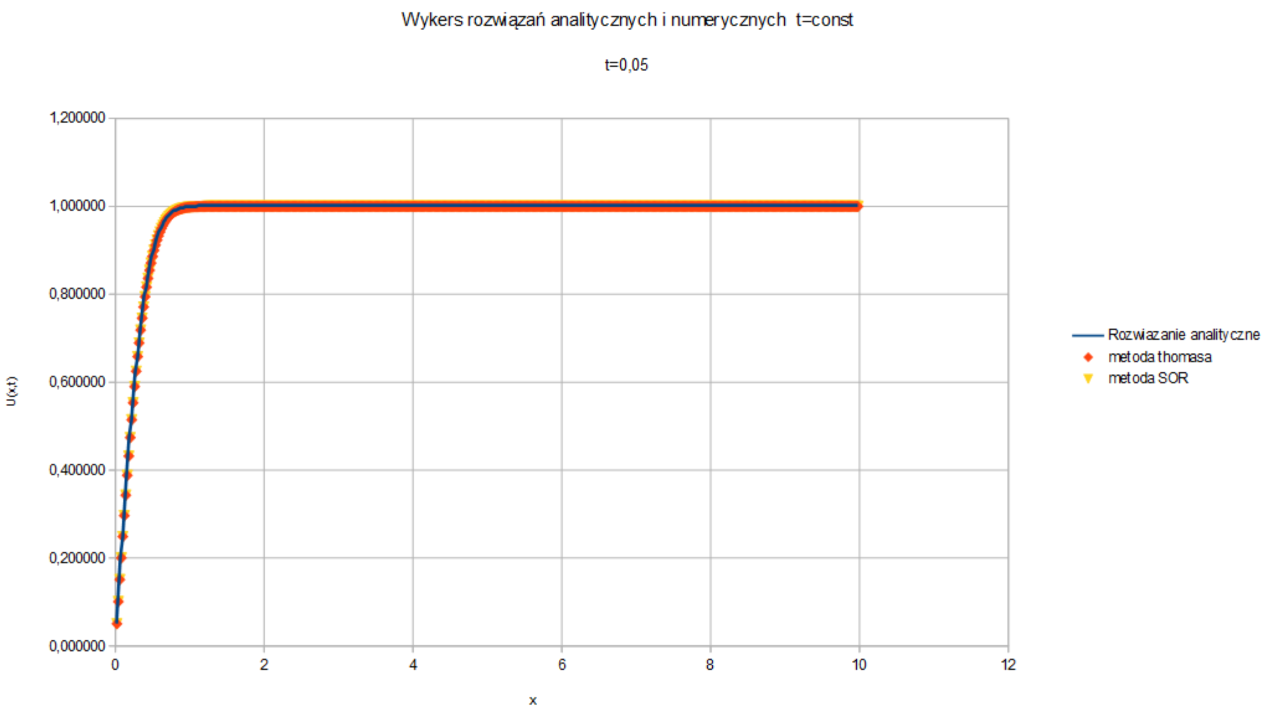
Wykres zależności maksymalnej wartości bezwzględnej błędu obserwowanego dla tmax w funkcji kroku przestrzennego h. Stworzony program zakłada stałą wielkość kroku przestrzennego, dlatego należało uruchomić program kilka razy zmieniając wartość kroku, a następnie agregować wyniki.

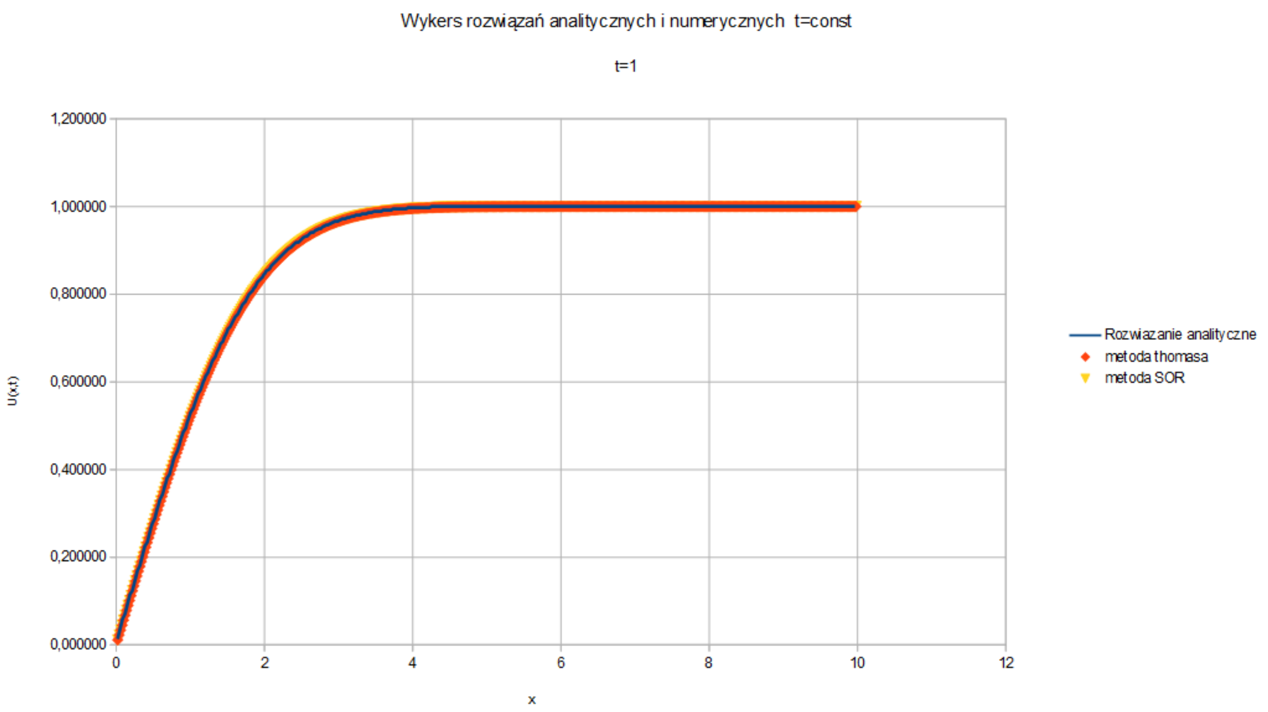


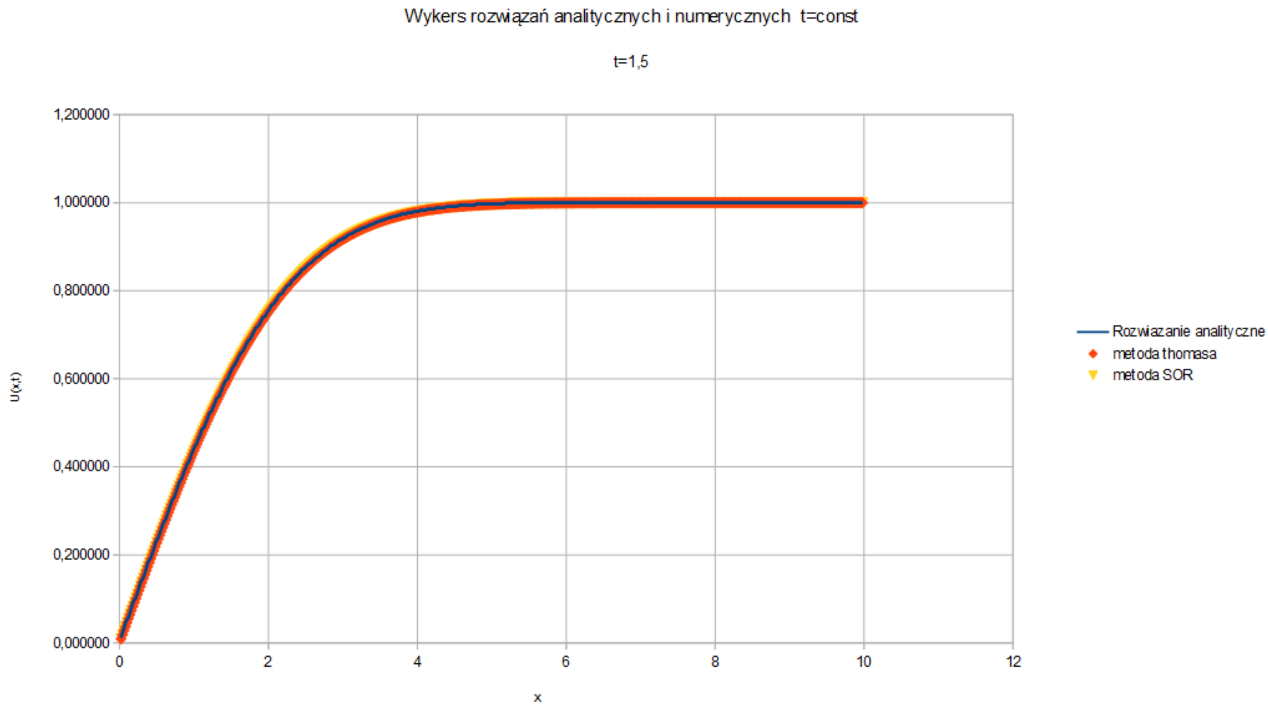
Zad 2

Wykresy rozwiązań numerycznych i analitycznych dla kilku wybranych wartości czasu t.



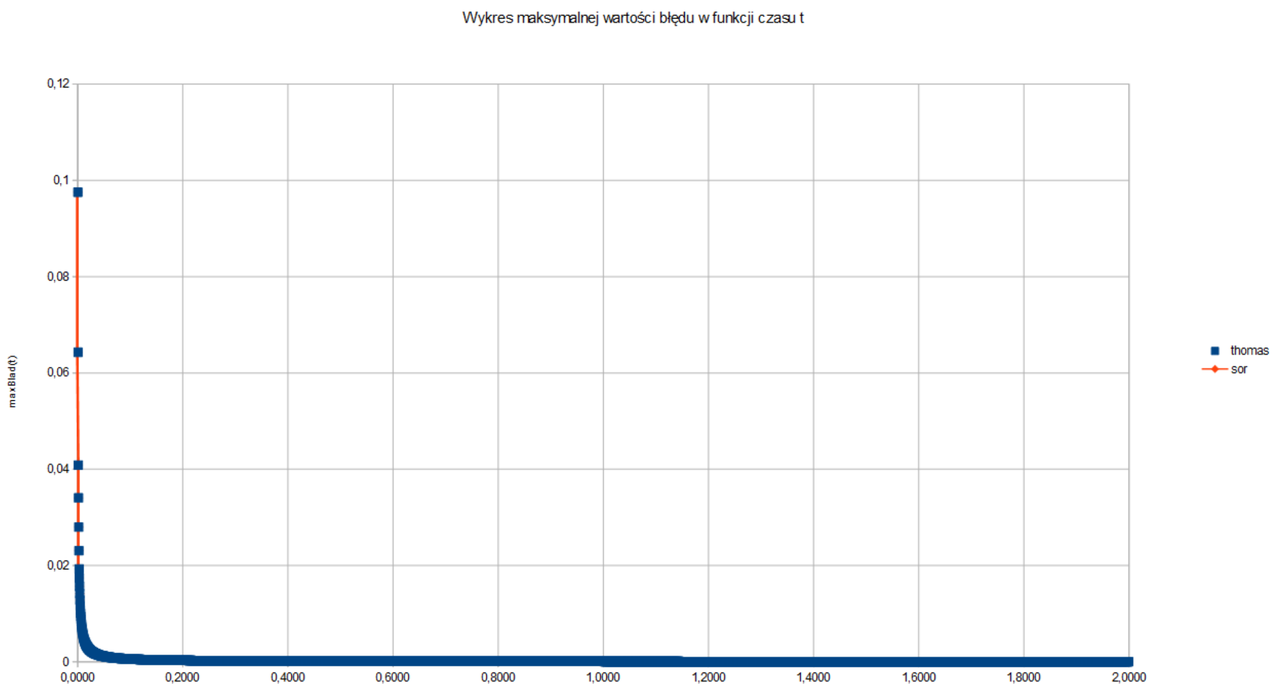

Wykers rozwiązań analitycznych i numerycznych t=const
t=0,05
1,200000
1,000000
0,800000
0,600000
0,400000
0,200000
0,000000
U(x,t)
0
2
4
6
8
10
12
x
Rozwiazanie analityczne
metoda thomasa
metoda SOR


Wykers rozwiązań analitycznych i numerycznych t=const
t=1
1,200000
1,000000
0,800000
0,600000
0,400000
0,200000
0,000000
U(x,t)
0
2
4
6
8
10
12
x
Rozwiazanie analityczne
metoda thomasa
metoda SOR

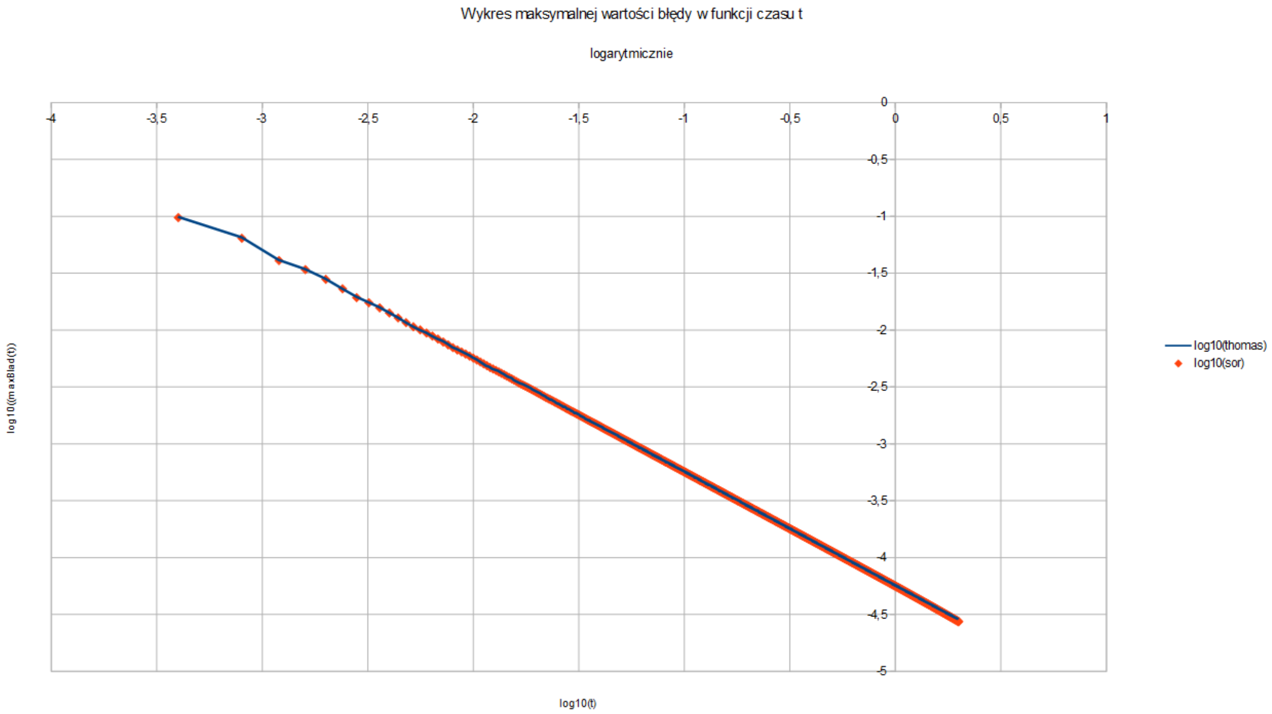


Zad 3

Wykresy zależności maksymalnej wartości bezwzględnej błędu w funkcji czasu t



W celu sprawdzenia charakterystyki propagacji błędu, powyższe dane zostały zaprezentowane w skali logarytmicznej



Wnioski:

1. Rozwiązania numeryczne pokrywają się z pewną dokładnością z rozwiązaniem analitycznym co oznacza że metody zostały prawidłowo zastosowane i zaimplementowane.
2. Program w ciągu ok 4 sekund* jest w stanie obliczyć wynik zarówno dla metody SOR jak i Thomasa, dla kroku przestrzennego 0,01 z dokładnością rzędu 10e-5 co świadczy małej złożoności obliczeniowej (w stosunku do problemu) implementacji
3. Z punktu widzenia analitycznego ważny jest ostatni wykres. Wyraźne widać, że wraz z kolejnymi iteracjami rozwiązań rośnie dokładność (zmniejsza się błąd). Zarówno metoda Thomasa i SOR zachowują się w podobny sposób. Na wielkość błędu ma wpływ zarówno błąd wynikający z poprzedniej iteracji, jak i błąd związany z konkretnymi obliczeniami kolejnej iteracji. Ponieważ błąd maleje w taki sam sposób wraz z kolejnymi iteracjami, a pochodne funkcji błędu maleją wraz ze wzrostem czasu, można wyciągnąć wniosek że na błąd wynikający z poprzedniej iteracji zostaje tłumiony.

Kod programu:

```
#include "stdafx.h"
#include <iostream>
#include <math.h>
#include <fstream>
#include <locale>
#include <iomanip>
#include "calerf.h"

using namespace std;

double analyticSolution(double x, double t);
double getMatrix(int i, int j);
void calculateNextSorX(double* newResult, double* prevSorResult, double omega, int n);
void solveSor(double* prevResult, double* result, int n);
void solveLowerTriangularSet(double* xn, double* B, double omega, int n);
void solveThomas(double* prevResult, double* result, int n);

//macierz trójdiagonalna jako wektory
double* L;
double* U;
double* D;

double* prevResult;
double* result;
double* analyticResult;

double DFactor = 1.0; //współczynnik transportu ciepla
double b = 10;//6.0*sqrt(2.0); //koniec przedziału dla zmiennej przestrzennej

double getMatrix(int i, int j)
{
	if (i == j - 1)
	{
		return U[i];
	}
	else if (i == j)
	{
		return D[i];
	}
	else if (i == j + 1)
	{
		return L[j];
	}
	else
	{
		return 0.0;
	}
}

double analyticSolution(double x, double t)
{
	return erf(x / 2 / sqrt(t));//sqrt(t)*D ale D = 1.0
}

void solveSor(double* prevResult, double* result, int n)
{
	double omega = 2.0;
	int nSor = 100;
	double tolArg = 1e-8;
	double tolRes = 1e-10;

	double* tmpResult = new double[n];
	double* tmpResult2 = new double[n];
```

```
	for (int j = 0; j < n; j++)
	{
		tmpResult[j] = prevResult[j];
		tmpResult2[j] = prevResult[j];
	}

	//ilosc iteracji - dodatkowy max oprocz dokladnosci
	int i = 0;

	do
	{
		for (int j = 0; j < n; j++)
		{
			tmpResult[j] = tmpResult2[j];
		}

		calculateNextSorX(tmpResult2, tmpResult, omega, n);


		i++;

	} while (i < nSor);

	for (int j = 0; j < n; j++)
	{
		result[j] = tmpResult2[j];
	}

	delete[] tmpResult;
	delete[] tmpResult2;
}

void calculateNextSorX(double* newResult, double* prevSorResult, double omega, int n)
{
	double* tmpResult = new double[n];

	tmpResult[n - 1] = prevResult[n - 1] - (prevSorResult[n - 1] * (1 - omega)*D[n - 1]);

	//prawa strona B-U*xn-1
	for (int i = n-2; i >= 0; i--)
	{
		tmpResult[i] = prevResult[i]-(prevSorResult[i]*(1 - omega)*D[i] + 
U[i]*prevSorResult[i+1]);
	}

	solveLowerTriangularSet(newResult, tmpResult, omega, n);

	for (int j = 0; j < n; j++)
	{
		prevSorResult[j] = tmpResult[j];
	}

	delete[] tmpResult;
}

void solveLowerTriangularSet(double* xn, double* B, double omega, int n)
{
	xn[0] = B[0] / (omega*D[0]);

	for (int i = 1; i < n; i++)
	{
		xn[i] = (B[i] - L[i-1]*xn[i-1]) / (omega*D[i]);
	}
}
```

```
void solveThomas(double* prevResult, double* result, int n)
{
	double* vectU = new double[n]; // eta
	double* vectB = new double[n]; // theta
	vectU[0] = D[0];
	for (int i = 1; i < n; ++i)
		vectU[i] = D[i] - (L[i - 1] * U[i - 1] / vectU[i-1]);

	vectB[0] = prevResult[0];
	for (int i = 1; i < n; ++i)
		vectB[i] = prevResult[i] - (L[i-1] * vectB[i - 1]) / vectU[i-1]; // nowe b

	result[n - 1] = vectB[n - 1]/vectU[n-1];
	for (int i = n - 2; i >= 0; --i)
		result[i] = (vectB[i] - U[i] * result[i + 1]) / vectU[i]; // propagacja
wsteczna - rozwiazania

	delete[] vectU;
	delete[] vectB;
}

void laasonenDiscretisation(double h, double dt)
{
	ofstream file("Dane.csv", std::ofstream::trunc);
	std::locale mylocale("");
	file.imbue(mylocale);

	double tMax = 2.0;
	double lambda = dt / h / h;//raczej zawsze 1, chyba ze beda baddzo male kroki, wtedy
moze byc blad
	int n = b / h; // n - ilosc węzłów zmiennej przestrzennej
	int k = tMax / dt; // k - ilosc wezlow zmiennej czasowej

	prevResult = new double[n];
	result = new double[n];
	analyticResult = new double[n];

	L = new double[n - 1];
	U = new double[n - 1];
	D = new double[n];

	double* xNodes = new double[n+1];
	double* tNodes = new double[k+1];

	//wyznaczam siatke przestrzenna, zeby nie powtarzac obliczen
	for (int i = 0; i < n+1; i++)
	{
		xNodes[i] = h*i;
	}

	//wyznaczam siatke czasowa, zeby nie powtarzac obliczen
	for (int j = 0; j < k+1; j++)
	{
		tNodes[j] = dt*j;
	}




	//warunek poczatkowy
	for (int i = 0; i < n; i++)
	{
		prevResult[i] = -1.0;
	}
```

```
	//ustawienie macierzy i wyrazu wolnego
	prevResult[0] = 1.0;
	prevResult[n - 1] = -1.0;

	D[0] = 1.0;
	U[0] = 0.0;
	//uzupelnianie macierzy A
	for (int i = 1; i < n-1; i++)
	{
		U[i] = lambda;
		L[i-1] = lambda;
		D[i] = -(1.0 + 2.0*lambda);
	}

	//i jeszcze ostatni wiersz
	L[n - 2] = 0.0;
	D[n - 1] = 1.0;

	// wrzucam pierwszy wezel czasowy do pliku
	if (file.is_open())
	{
		file << tNodes[0];
	}
	else
	{
		cout << "Blad otwarcia pliku" << endl;
		return;
	}

	//wrzucam siatke przestrzenna do pliku
	for (int i = 0; i < n; i++)
	{
		file << xNodes[i] << ";";
	}
	file << endl;

	//wrzucam rozwiazania w chwili t=0
	for (int i = 0; i < n; i++)
	{
		file << -prevResult[i] << ";";

	}

	file << endl;

	//inicjalizacja zmiennych zwiazanych z szukaniem maksymalnego bledu w funkcji czasu
(zad3)
	double maxError;
	double tmpError;
	ofstream fileErrorT("MaxBladWFunkcjiCzasu.csv", fstream::trunc);
	fileErrorT.imbue(mylocale);

	for (int j = 1; j<k; j++)
	{
		maxError = 0.0;
		tmpError = 0.0;

		prevResult[0] = 0.0;
		prevResult[n - 1] = 1.0; // zgodnie z warunkami brzegowymi

		solveSor(prevResult,result,n);
		//solveThomas(prevResult,result,n);
```

```
            file << tNodes[j];

            for (int i = 0; i < n; i++)
            {
                  //wygenerowanie arkusza danych rozwiazania analitycznego (zad2)
                  //result[i] = analyticSolution(xNodes[i], tNodes[j]);
                  file << result[i] << ";";

                  prevResult[i] = -result[i];

                  //wygenerowanie danych zwiazanych z szukaniem maksymalnego bledu w
funkcji czasu (zad3)
                  tmpError = fabs(result[i] - analyticSolution(xNodes[i], tNodes[j]));
                  if (tmpError > maxError)
                  {
                        maxError = tmpError;
                  }
            }

            fileErrorT << tNodes[j] << ";" << maxError << ";" << endl;

            file << endl;
      }


      //zad1
      //maksymalny blad bezwzgledny dla tmax (ostatnie co bylo liczone to wlasnie rownanie
dla tmax) w funkcji kroku przestrzennego h (musze uruchomic program kilka razy z roznym
krokiem h)
      /*
      double maxError = 0.0;
      double tmpError;
      for (int i = 0; i < n; i++)
      {
            tmpError = fabs(result[i] - analyticSolution(xNodes[i], tNodes[k - 1]));
            if (tmpError > maxError)
            {
                  maxError = tmpError;
            }
      }

      ofstream fileErrorTMax("BladTmax.csv", fstream::app);
      fileErrorTMax.imbue(mylocale);
      fileErrorTMax << h << ";" << log10(h) << ";" << maxError << ";" << log10(maxError) <<
";" << endl;
      fileErrorTMax.close();
      */

      //zad3
      fileErrorT.close();

      delete[] xNodes;
      delete[] tNodes;

      delete[] prevResult;
      delete[] result;
      delete[] analyticResult;

      delete[] L;
      delete[] U;
      delete[] D;

      file.close();
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
	double h = 0.02;
	double dt = h*h;

	laasonenDiscretisation(h, dt);

	return 0;
}
```