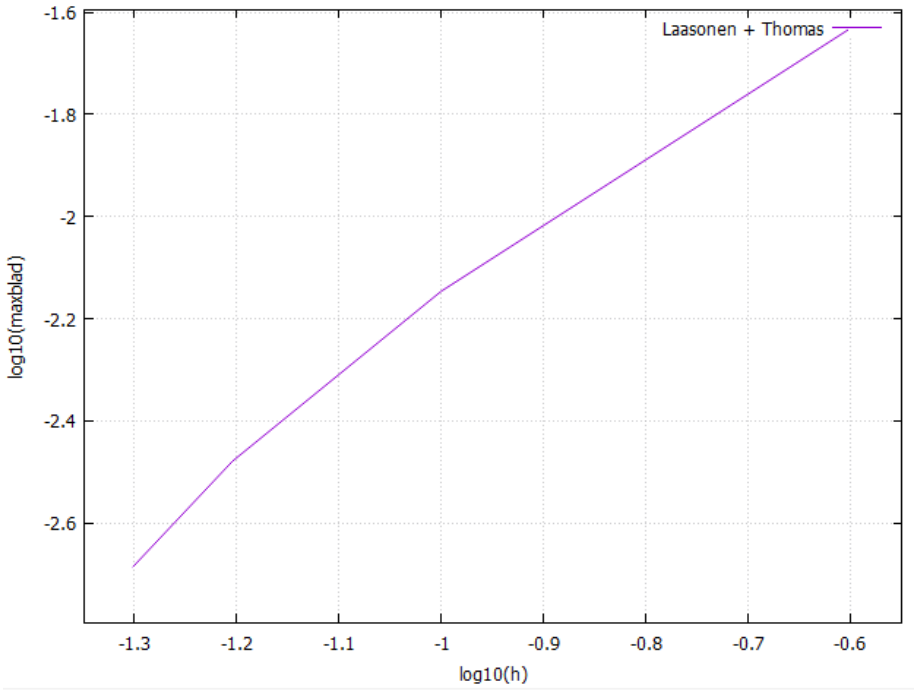
Metody Obliczeniowe - Laboratorium

Ćwiczenie 11-1

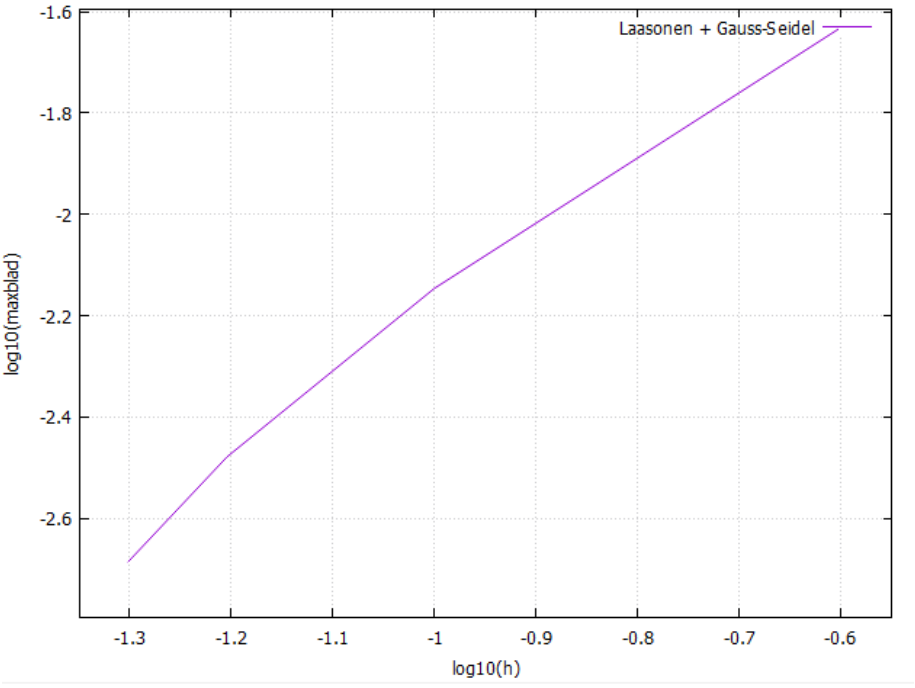
Szymon Czajkowski - grupa laboratoryjna 01

1. Wykresy zależności maksymalnej wartości bezwzględnej błędu dla wartości t_{max} w funkcji kroku przestrzennego h.

1.1. Metoda Laasonen + algorytm Thomasa:

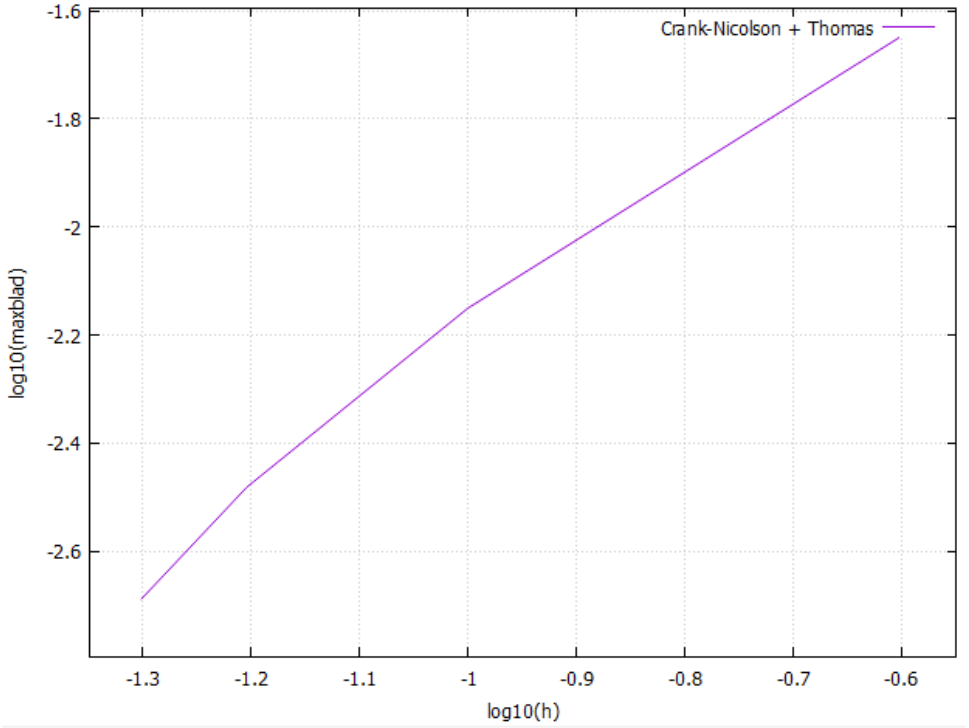


1.2. Metoda Laasonen + metoda iteracyjna Gaussa-Seidela:

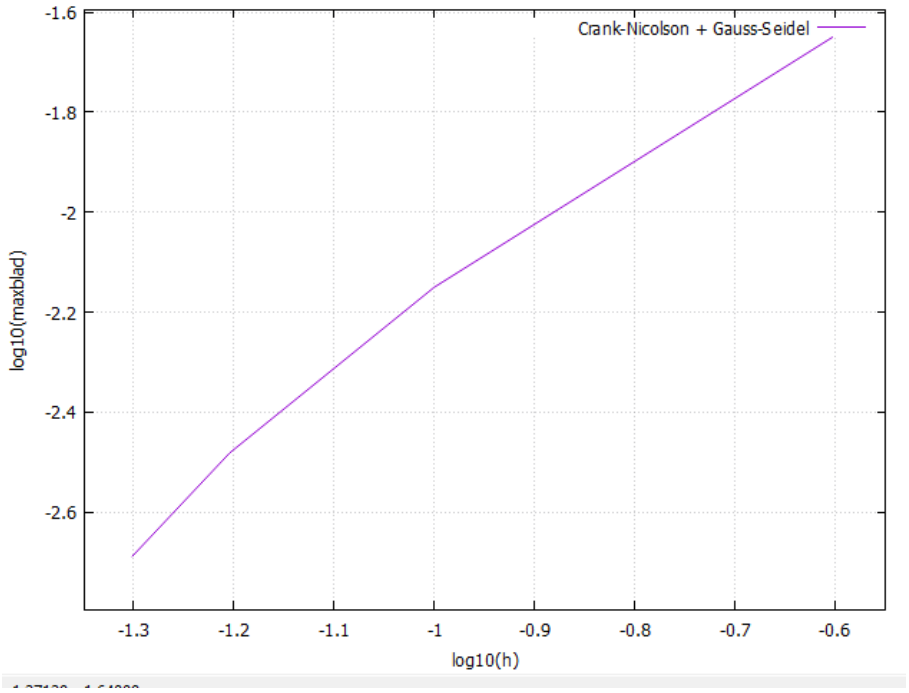


Rzędy odczytane z wykresów dla metody Laasonen są bliskie teoretycznym założeniom.

1.3. Metoda Cranka-Nicolson + algorytm Thomasa:



1.4. Metoda Cranka-Nicolson + metoda iteracyjna Gaussa-Seidela:

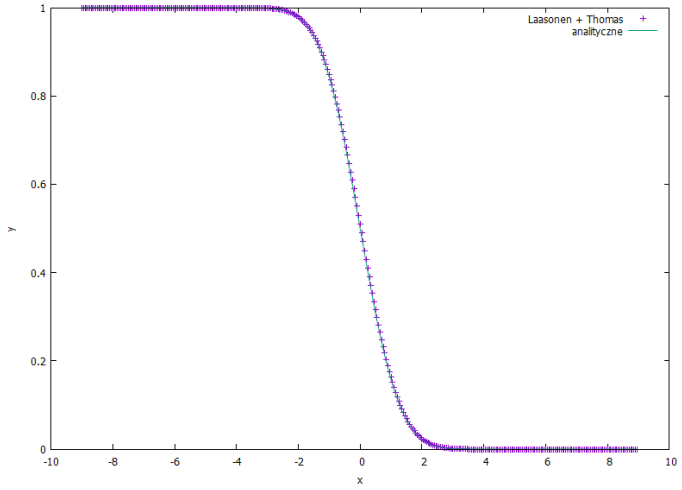


Rzędy odczytane z wykresów dla metody Cranka-Nicolson są bliskie teoretycznym założeniom.

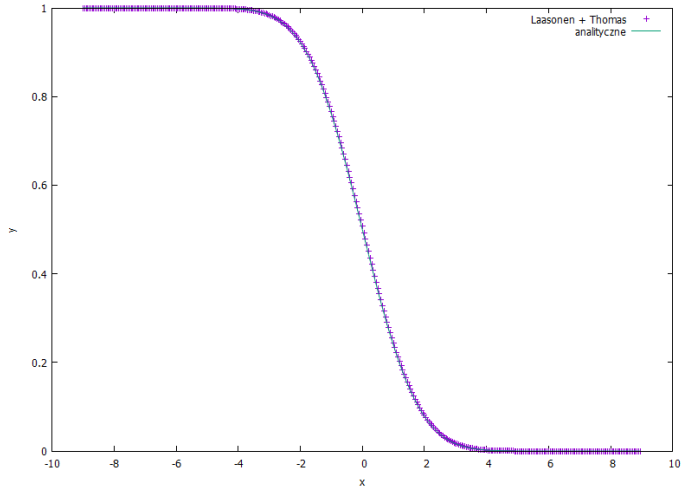
2. Wykresy rozwiązań numerycznych i analitycznych dla kilku wybranych wartości czasu.

2.1. Metoda Laasonen + algorytm Thomasa:

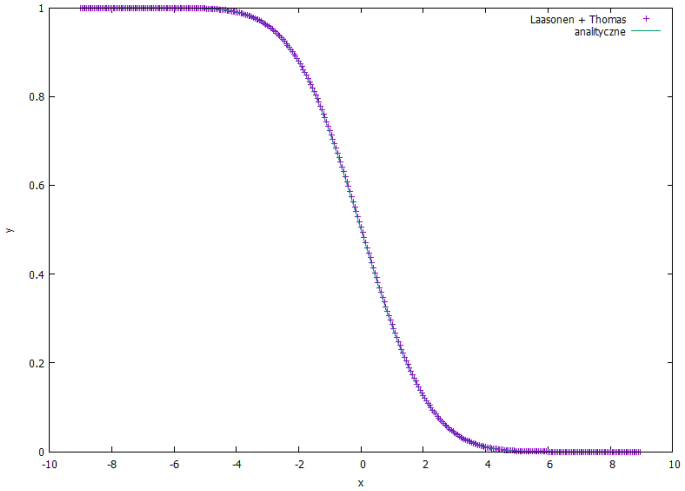
t = 0,5



t = 1

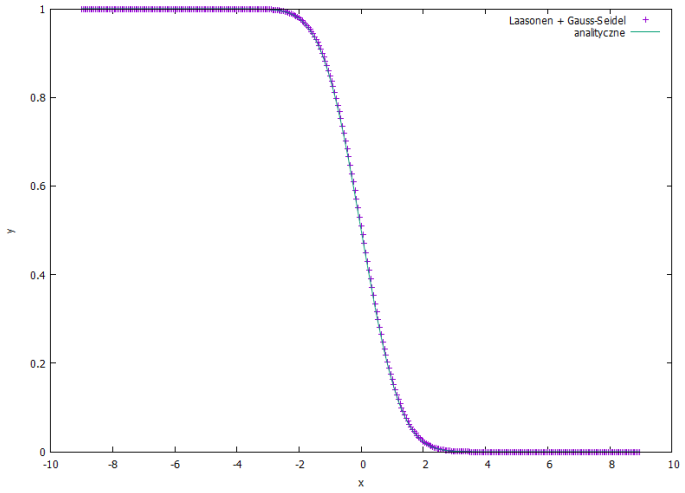


t = 1,5

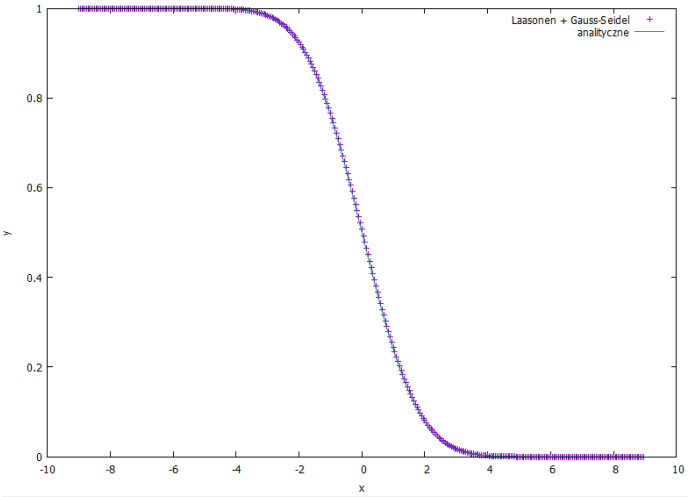


2.2. Metoda Laasonen + metoda iteracyjna Gaussa-Seidela:

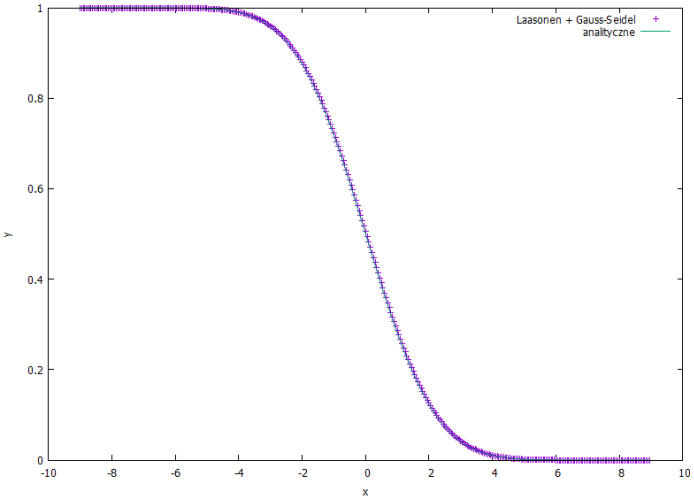
t = 0,5



t = 1

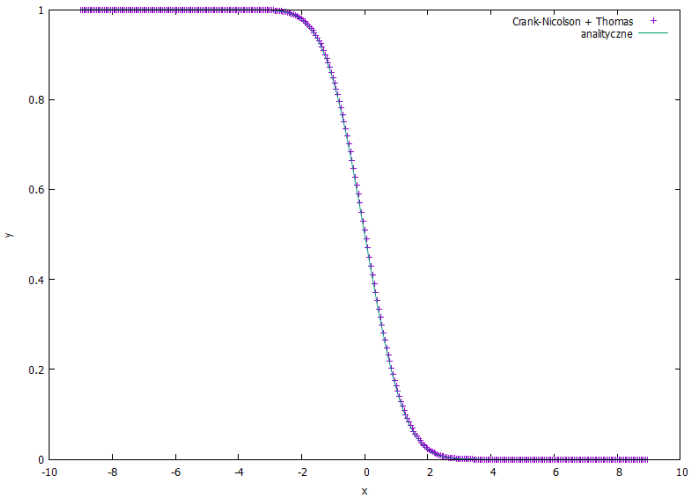


t = 1,5

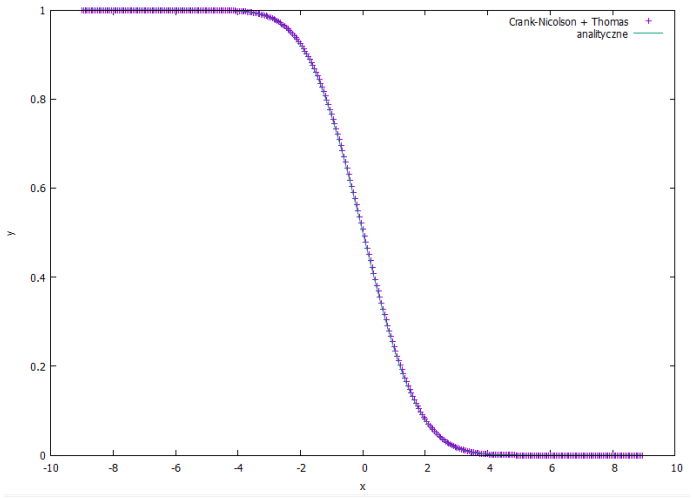


2.3. Metoda Cranka-Nicolson + algorytm Thomasa

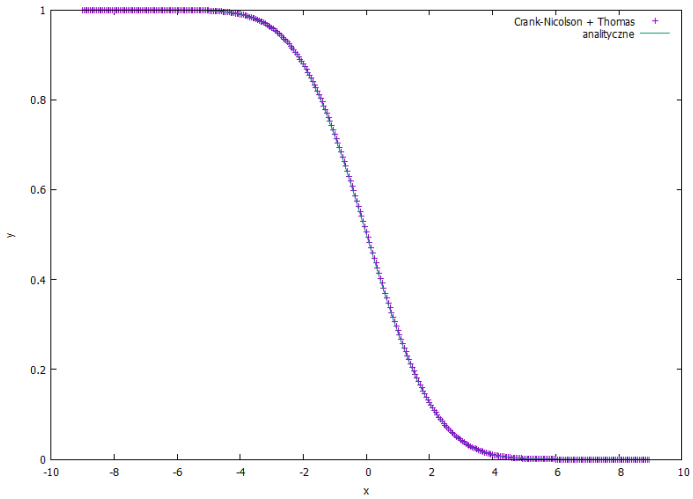
t = 0,5



t = 1

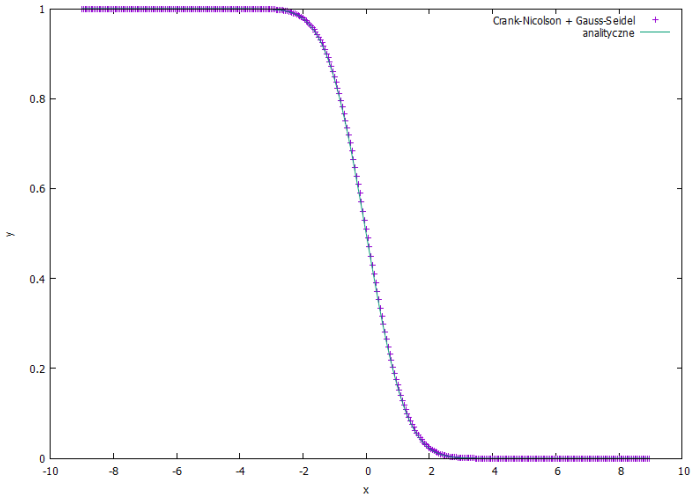


t = 1,5

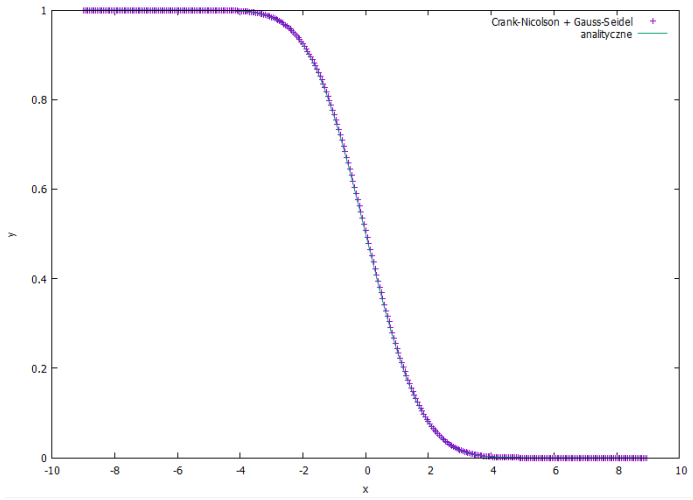


2.4. Metoda Cranka-Nicolson + metoda iteracyjna Gaussa-Seidela:

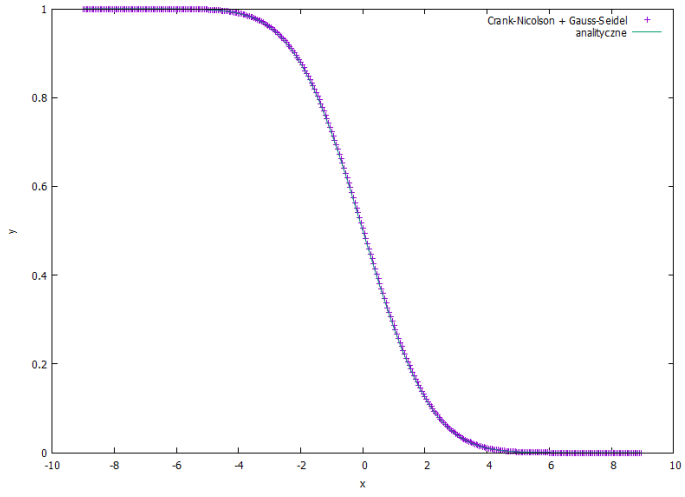
t = 0,5



t = 1

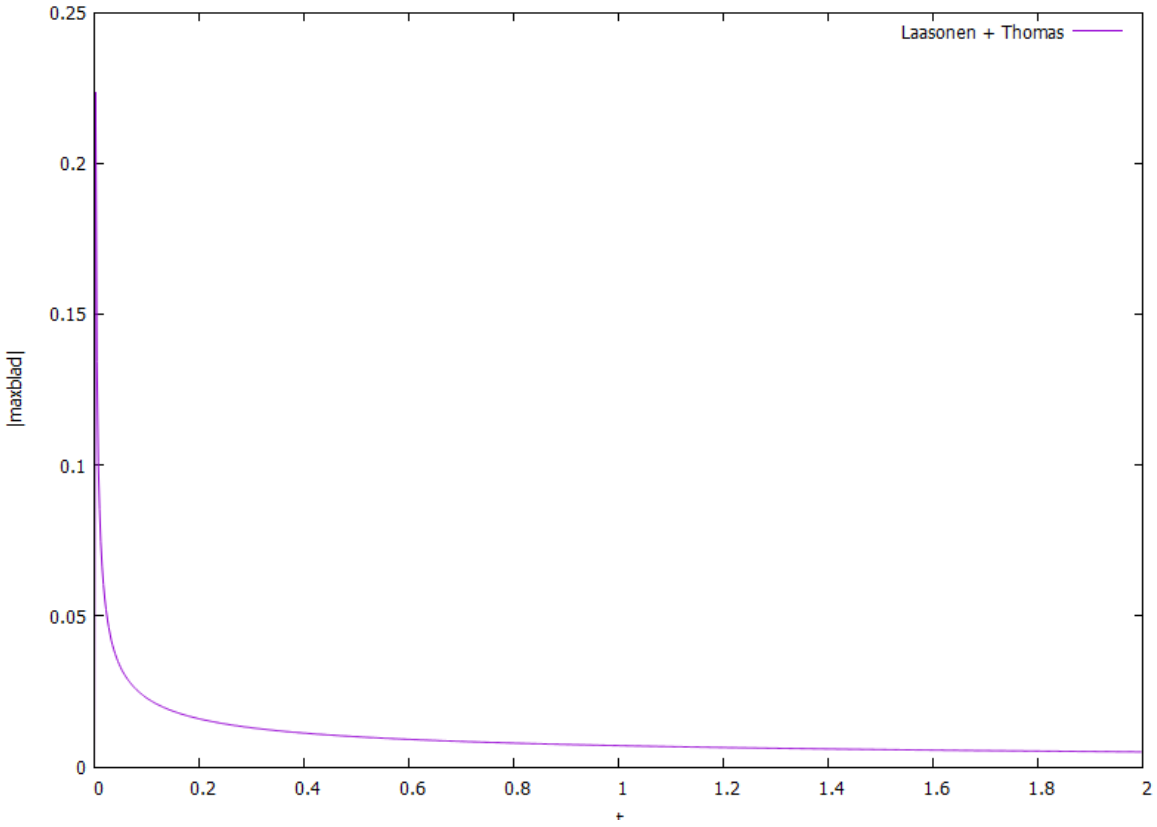


t = 1,5

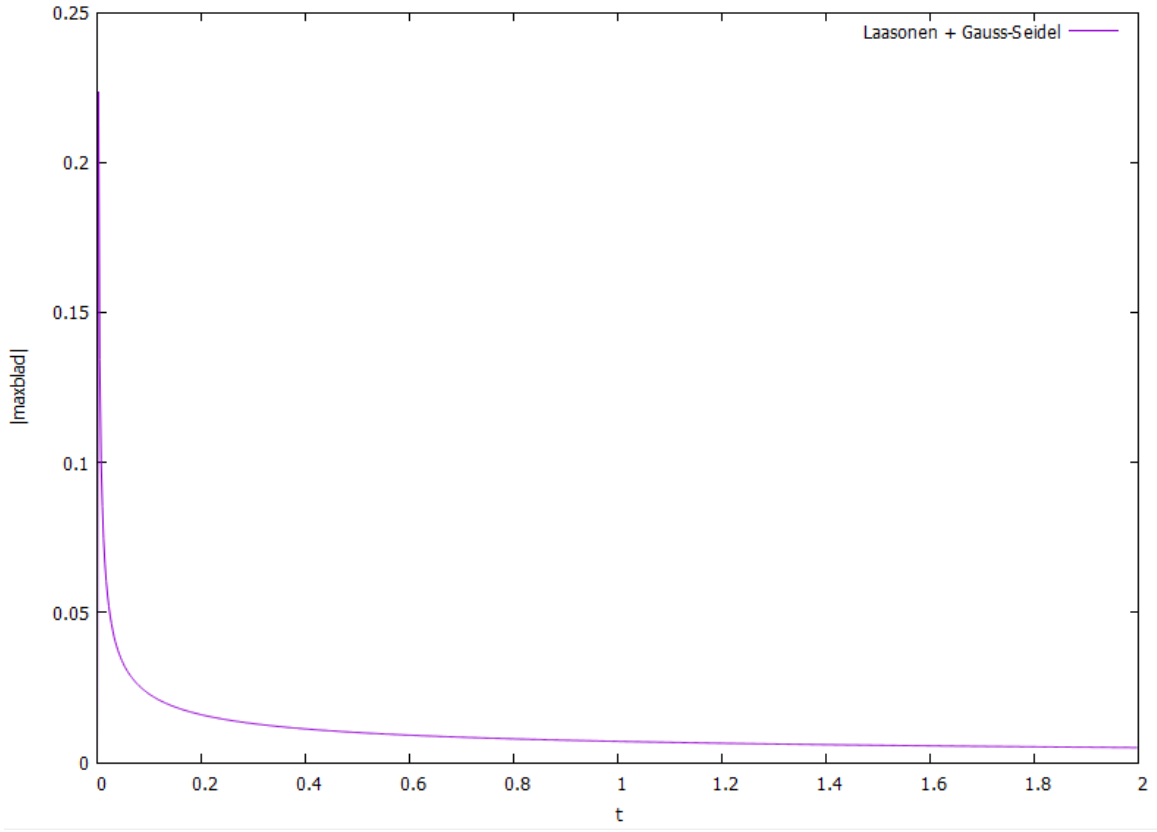


3. Wykresy zależności maksymalnej wartości bezwzględnej błędu w funkcji czasu t.

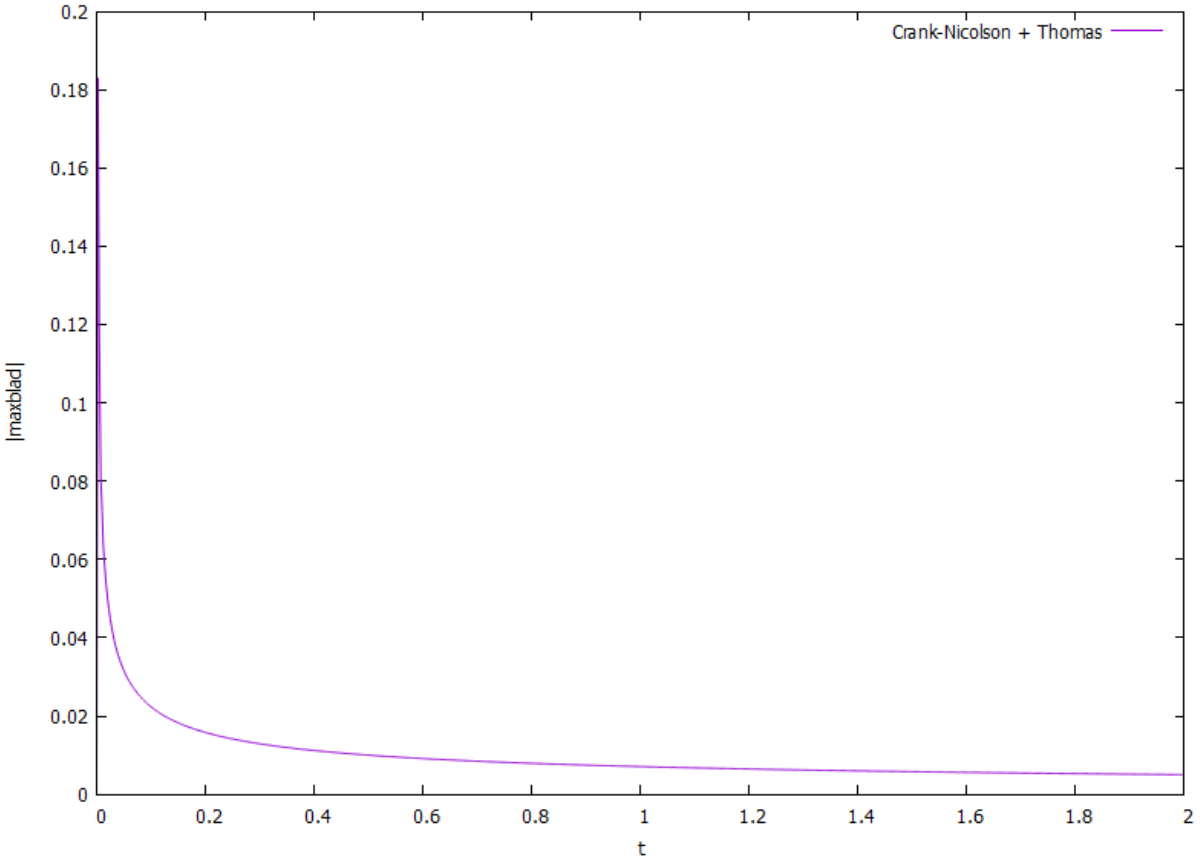
3.1. Metoda Laasonen + algorytm Thomasa:



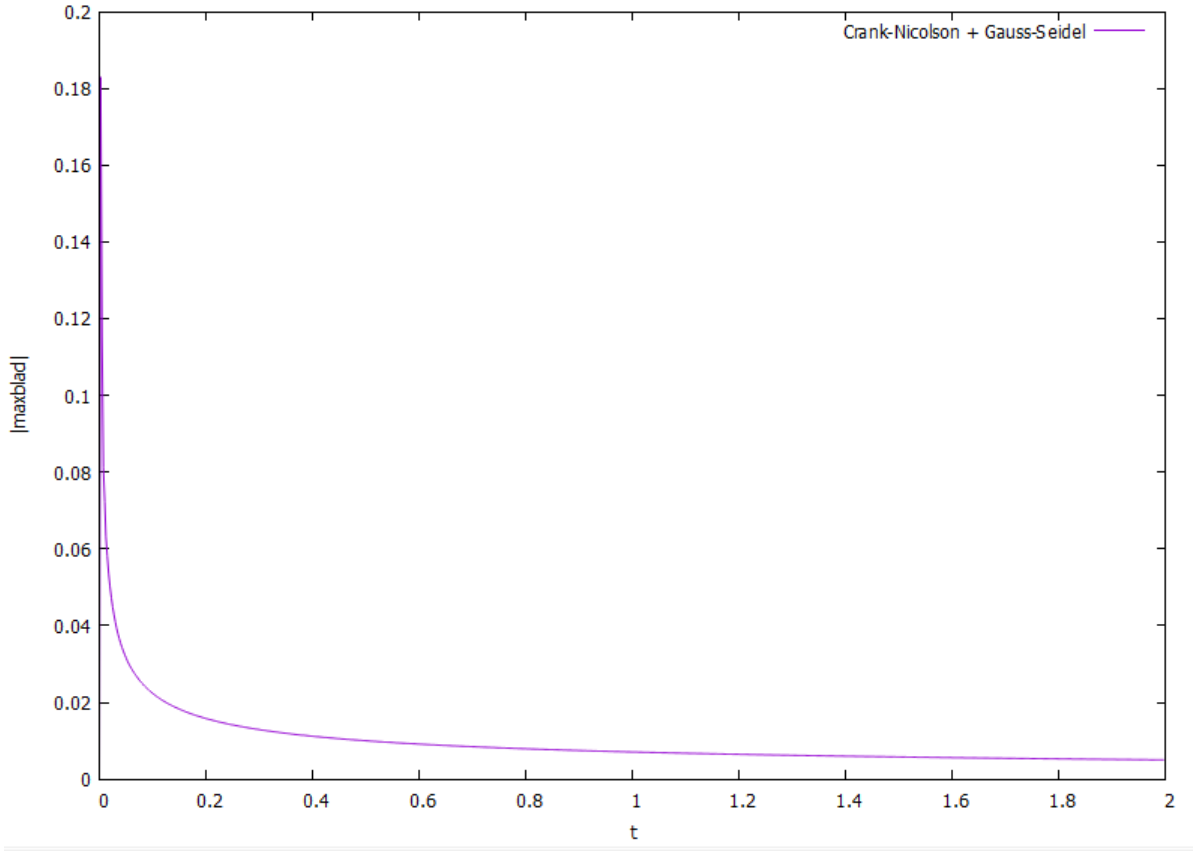
3.2. Metoda Laasonen + metoda iteracyjna Gaussa-Seidela:



3.3 Metoda Cranka-Nicolson + algorytm Thomasa:



3.4 Metoda Cranka-Nicolson + metoda iteracyjna Gaussa-Seidela:



Wnioski:

- Do obliczeń wybrałem krok o wartości 0.05, daje on dokładne przybliżenie rozwiązania, a czas obliczeń jest akceptowalny.

- Wykresy zależności maksymalnego błędu od kroku h pokazują że wszystkie metody mają porównywalną dokładność obliczeń.

- Obliczenia wykonywane z użyciem algorytmu Thomasa są szybsze niż metoda iteracyjna Gaussa-Seidela, spowodowane jest to tym, że w algorytmie Thomasa nie działamy na całej macierzy tak jak w metodzie Gaussa-Seidela.

- Metoda dyskretyzacji Cranka-Nicolsona daje większą dokładność obliczeń w stosunku do metody Laasonen, widać to na wykresach maksymalnej wartości bezwzględnej błędu w funkcji czasu, Metoda Cranka-Nicolson największy błąd o wartości około 0.18 osiąga przy czasie bliskim zeru, natomiast błąd w podobnym czasie w Metodzie Laasonen wynosi ponad 0.2.

- Niski błąd przy czasie równym 0 został uzyskany dzięki warunkom brzegowym.

- Malejący maksymalny błąd w funkcji czasu, jest wynikiem numerycznej stabilnośći zastosowanych metod

Kod programu:

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>
#include "calerf.h"
using namespace std;
#define N_MAX 50
const double TOLX = 1e-8;
const double TOLF = 1e-8;
const double T_MIN = 0.0;
const double T_MAX = 2.0;
const double D = 1.0;
const double POCZATEK_PRZEDZIALU = -(6.0 * sqrt(D * T_MAX));
const double KONIEC_PRZEDZIALU = 6.0 * sqrt(D * T_MAX);;
const double H = 0.05;
const double LAMBDA = 1.0;
double **stworz_macierz(int N, int M) {
    double **macierz = new double *[N];
    for (int i = 0; i < N; i++) {
        macierz[i] = new double[M];
    }
    return macierz;
}
void usun_macierz(double **macierz, int N) {
    for (int i = 0; i < N; i++) {
        delete[] macierz[i];
    }
    delete[] macierz;
}
double *siatka_kroku(double h, int M) {
    double *wektor = new double[M];
    double x = POCZATEK_PRZEDZIALU;
    for (int i = 0; i < M; i++) {
        wektor[i] = x;
        x += h;
    }
    return wektor;
}
double *siatka_czasu(double dt, int N) {
    double *wektor = new double[N];
    double t = T_MIN;
    for (int i = 0; i < N; i++) {
        wektor[i] = t;
        t += dt;
    }
    return wektor;
}
void zapisz_do_pliku_w(double *wektor, int N, string nazwa) {
    ofstream plik;
    plik.open(nazwa);
    double *czas = siatka_czasu((LAMBDA * H * H) / D, N);
    for (int i = 0; i < N; i++) {
        plik << czas[i] << "\t" << wektor[i] << endl;
    }
}
void zapisz_do_pliku_m(double **macierz, int N, int M, string nazwa) {
    ofstream plik;
    plik.open(nazwa);
    double *kroki = siatka_kroku(H, N);
    for (int i = 0; i < N; i++) {
        plik << kroki[i] << "\t";
        for (int j = 0; j < M; j++) {
            plik << macierz[i][j] << "\t";
        }
        plik << endl;
    }
    plik.close();
}

double rozwiazanie_analityczne(double x, double t) {
    return calerfpack::erfc_1((x) / (2.0 * sqrt(D * t))) / 2.0;
}
```

```
double **oblicz_rozwiazanie_analityczne(double h, double dt, int N, int M) {
    double **macierz_rozwiazan = stworz_macierz(N, M);
    double x = POCZATEK_PRZEDZIALU;
    double t = T_MIN;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            macierz_rozwiazan[i][j] = rozwiazanie_analityczne(x, t);
            x += h;
        }
        x = POCZATEK_PRZEDZIALU;
        t += dt;
    }
    return macierz_rozwiazan;
}
void warunek_poczatkowy(double **macierz, double h, int M) {
    double x = POCZATEK_PRZEDZIALU;
    for (int i = 0; i < M; i++) {
        if (x < 0.0) {
            macierz[0][i] = 1.0;
        } else {
            macierz[0][i] = 0.0;
        }
        x += h;
    }
}
void warunek_brzegowy(double **macierz, int N, int M) {
    for (int i = 0; i < N; i++) {
        macierz[i][0] = 1.0;
        macierz[i][M - 1] = 0.0;
    }
}
double norma_maksimum(double *wektor, int N) {
    double max = fabs(wektor[0]);
    for (int i = 1; i < N; i++) {
        if (max < fabs(wektor[i])) {
            max = fabs(wektor[i]);
        }
    }
    return max;
}
double* policz_bledy(double **analityczne, double **numeryczne, int n, int m) {
    double **bledy = stworz_macierz(n, m);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            bledy[i][j] = fabs(numeryczne[i][j] - analityczne[i][j]);
        }
    }
    double* max_blad = new double[n];
    for (int i = 0; i < n; i++) {
        max_blad[i] = norma_maksimum(bledy[i], m);
    }
    usun_macierz(bledy, n);
    return max_blad;
}
double **transponuj(double **macierz, int N, int M) {
    double **result = stworz_macierz(M, N);
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < M; ++j) {
            result[j][i] = macierz[i][j];
        }
    }
    return result;
}
void thomas(double **a, double *b, double *x, int m) {
    double *l = new double[m];
    double *d = new double[m];
    double *u = new double[m];
    d[0] = a[0][0];
    for (int i = 1; i < m - 1; i++) {
        u[i] = a[i][i + 1];
        d[i] = a[i][i];
        l[i - 1] = a[i][i - 1];
    }
    d[m - 1] = a[m - 1][m - 1];
```

```
    for (int i = 1; i < m; i++) {
        d[i] = d[i] - ((l[i - 1] / d[i - 1]) * u[i - 1]);
        b[i] = b[i] - ((l[i - 1] / d[i - 1]) * b[i - 1]);
    }
    x[m - 1] = b[m - 1] / d[m - 1];
    for (int i = m - 2; i >= 0; i--) {
        x[i] = (b[i] - u[i] * x[i + 1]) / d[i];
    }
}
bool koniec(double* residuum, double* estymator_bledu, int m) {
    for (int i = 0; i < m; i++) {
        if (residuum[i] > TOLX || estymator_bledu[i] > TOLF) {
            return false;
        }
    }
    return true;
}
void gauss_seidel(double** macierz, double* b, double* przybl_x, int N) {
    double suma = 0.0;
    double estymator_bledu[N];
    double residuum[N];
    double tmp_x[N];
    double prawa_strona[N];
    for (int k = 0; k < N_MAX; k++) {
        for (int i = 0; i < N; i++) {
            suma = 0.0;
            for (int j = 0; j < N; j++) {
                if (i < j) {
                    suma += macierz[i][j] * przybl_x[j];
                }
            }
            prawa_strona[i] = b[i] - suma;
        }
        for (int i = 0; i < N; i++) {
            suma = 0.0;
            for (int j = 0; j < i; j++) {
                suma += macierz[i][j] * tmp_x[j];
            }
            tmp_x[i] = (prawa_strona[i] - suma) / macierz[i][i];
        }
        for (int i = 0; i < N; i++) {
            residuum[i] = 0.0;
            estymator_bledu[i] = fabs(tmp_x[i] - przybl_x[i]);
            for (int j = 0; j < N; j++) {
                residuum[i] += macierz[i][j] * tmp_x[j];
            }
            residuum[i] = fabs(residuum[i] - b[i]);
            przybl_x[i] = tmp_x[i];
        }
        if (koniec(residuum, estymator_bledu, N)) {
            break;
        }
    }
}
double **laasonen_gs_pom(double h, int n, int m) {
    double **U = stworz_macierz(n, m);
    warunek_poczatkowy(U, h, m);
    warunek_brzegowy(U, n, m);
    double *b = new double[m];
    double *x = new double[m];
    for (int i = 0; i < m; i++) {
        b[i] = 0.0;
        x[i] = 0.0;
    }
    double **a = stworz_macierz(m, m);
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < m; j++) {
            a[i][j] = 0.0;
        }
    }
    for (int k = 1; k < n; k++) {
        a[0][0] = 1.0;
        b[0] = U[k - 1][0];
        for (int i = 1; i < m - 1; i++) {
```

```
            a[i][i + 1] = LAMBDA;
            a[i][i] = -(1.0 + 2.0 * LAMBDA);
            a[i][i - 1] = LAMBDA;
            b[i] = -U[k - 1][i];
        }
        b[m - 1] = 0.0;
        a[m - 1][m - 1] = 1.0;
        gauss_seidel(a, b, x, m);
        for (int i = 1; i < m - 1; i++) {
            U[k][i] = x[i];
        }
    }
    usun_macierz(a, m);
    delete[] b;
    delete[] x;
    return U;
}
double **laasonen_thomas_pom(double h, int n, int m) {
    double **U = stworz_macierz(n, m);
    warunek_poczatkowy(U, h, m);
    warunek_brzegowy(U, n, m);
    double *b = new double[m];
    double *x = new double[m];
    for (int i = 0; i < m; i++) {
        b[i] = 0.0;
        x[i] = 0.0;
    }
    double **a = stworz_macierz(m, m);
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < m; j++) {
            a[i][j] = 0.0;
        }
    }
    for (int k = 1; k < n; k++) {
        a[0][0] = 1.0;
        b[0] = U[k - 1][0];
        for (int i = 1; i < m - 1; i++) {
            a[i][i + 1] = LAMBDA;
            a[i][i] = -(1.0 + 2.0 * LAMBDA);
            a[i][i - 1] = LAMBDA;
            b[i] = -U[k - 1][i];
        }
        b[m - 1] = 0.0;
        a[m - 1][m - 1] = 1.0;
        thomas(a, b, x, m);
        for (int i = 1; i < m - 1; i++) {
            U[k][i] = x[i];
        }
    }
    usun_macierz(a, m);
    delete[] b;
    delete[] x;
    return U;
}
double** crank_nicolson_thomas_pom(double h, int n, int m) {
    double **U = stworz_macierz(n, m);
    warunek_poczatkowy(U, h, m);
    warunek_brzegowy(U, n, m);
    double *b = new double[m];
    double *x = new double[m];
    for (int i = 0; i < m; i++) {
        b[i] = 0.0;
        x[i] = 0.0;
    }
    double **a = stworz_macierz(m, m);
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < m; j++) {
            a[i][j] = 0.0;
        }
    }
    for (int k = 1; k < n; k++) {
        a[0][0] = 1.0;
        a[0][1] = 0.0;
        b[0] = 1.0;
```

```
        for (int i = 1; i < m - 1; i++) {
            a[i][i + 1] = LAMBDA / 2.0;
            a[i][i] = -(1.0 + LAMBDA);
            a[i][i - 1] = LAMBDA / 2.0;
            b[i] = -((LAMBDA / 2.0) * U[k - 1][i - 1]  + (1.0 - LAMBDA) * U[k - 1][i] + (LAMBDA / 2.0)
* U[k - 1][i + 1]);
        }
        b[m - 1] = 0.0;
        a[m - 1][m - 1] = 1.0;
        a[m - 1][m - 2] = 0.0;
        thomas(a, b, x, m);
        for (int i = 1; i < m - 1; i++) {
            U[k][i] = x[i];
        }
    }
    usun_macierz(a, m);
    delete[] b;
    delete[] x;
    return U;
}
double** crank_nicolson_gs_pom(double h, int n, int m) {
    double **U = stworz_macierz(n, m);
    warunek_poczatkowy(U, h, m);
    warunek_brzegowy(U, n, m);
    double *b = new double[m];
    double *x = new double[m];
    for (int i = 0; i < m; i++) {
        b[i] = 0.0;
        x[i] = 0.0;
    }
    double **a = stworz_macierz(m, m);
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < m; j++)
        {
            a[i][j] = 0.0;
        }
    }
    for (int k = 1; k < n; k++) {
        a[0][0] = 1.0;
        a[0][1] = 0.0;
        b[0] = 1.0;
        for (int i = 1; i < m - 1; i++) {
            a[i][i + 1] = LAMBDA / 2.0;
            a[i][i] = -(1.0 + LAMBDA);
            a[i][i - 1] = LAMBDA / 2.0;
            b[i] = -((LAMBDA / 2.0) * U[k - 1][i - 1]  + (1.0 - LAMBDA) * U[k - 1][i] + (LAMBDA / 2.0)
* U[k - 1][i + 1]);
        }
        b[m - 1] = 0.0;
        a[m - 1][m - 1] = 1.0;
        a[m - 1][m - 2] = 0.0;
        gauss_seidel(a, b, x, m);
        for (int i = 1; i < m - 1; i++) {
            U[k][i] = x[i];
        }
    }
    usun_macierz(a, m);
    delete[] b;
    delete[] x;
    return U;
}
void laasonen() {
    double h = H;
    double dt = (LAMBDA * h * h) / D;
    int n = ((T_MAX - T_MIN) / dt);
    int m = ((KONIEC_PRZEDZIALU - POCZATEK_PRZEDZIALU) / h);
    double** analityczne = oblicz_rozwiazanie_analityczne(h, dt, n, m);
    double** numeryczneT = laasonen_thomas_pom(h, n, m);
    double** numeryczneGS = laasonen_gs_pom(h, n, m);
    zapisz_do_pliku_m(transponuj(analityczne, n, m), m, n, "rozwiazanie_analityczne.txt");
    zapisz_do_pliku_m(transponuj(numeryczneT, n, m), m, n, "lt_numerycznie.txt");
    zapisz_do_pliku_m(transponuj(numeryczneGS, n, m), m, n, "lgs_numerycznie.txt");
    zapisz_do_pliku_w(policz_bledy(analityczne, numeryczneT, n, m), n, "lt_bledy_t.txt");
    zapisz_do_pliku_w(policz_bledy(analityczne, numeryczneGS, n, m), n, "lgs_bledy_t.txt");
```

```
    usun_macierz(analityczne, n);
    usun_macierz(numeryczneT, n);
    usun_macierz(numeryczneGS, n);
}
void crank_nicolson() {
    double h = H;
    double dt = (LAMBDA * h * h) / D;
    int n = (T_MAX - T_MIN) / dt;
    int m = (KONIEC_PRZEDZIALU - POCZATEK_PRZEDZIALU) / h;
    double** analityczne = oblicz_rozwiazanie_analityczne(h, dt, n, m);
    double** numeryczneT = crank_nicolson_thomas_pom(h, n, m);
    double** numeryczneGS = crank_nicolson_gs_pom(h, n, m);
    zapisz_do_pliku_m(transponuj(analityczne, n, m), m, n, "rozwiazanie_analityczne.txt");
    zapisz_do_pliku_m(transponuj(numeryczneT, n, m), m, n, "cnt_numerycznie.txt");
    zapisz_do_pliku_m(transponuj(numeryczneGS, n, m), m, n, "cngs_numerycznie.txt");
    zapisz_do_pliku_w(policz_bledy(analityczne, numeryczneT, n, m), n, "cnt_bledy_t.txt");
    zapisz_do_pliku_w(policz_bledy(analityczne, numeryczneGS, n, m), n,  "cngs_bledy_t.txt");
    usun_macierz(analityczne, n);
    usun_macierz(numeryczneT, n);
    usun_macierz(numeryczneGS, n);
}
void laasonen_thomas_bledy_h() {
    double kroki[4] = {0.25, 0.1, 0.0625, 0.05};
    double h;
    double dt;
    int n, m;
    ofstream plik;
    plik.open("lt_bledy_h.txt");
    cout << "lt" << endl;
    for (int i = 0; i < 4; i++) {
        h = kroki[i];
        dt = (LAMBDA * h * h) / D;
        n = (T_MAX - T_MIN) / dt;
        m = (KONIEC_PRZEDZIALU - POCZATEK_PRZEDZIALU) / h;
        double** analityczne = oblicz_rozwiazanie_analityczne(h, dt, n, m);
        double** numeryczne = laasonen_thomas_pom(h, n, m);
        double* bledy = policz_bledy(analityczne, numeryczne, n, m);
        plik << log10(h) << "\t" << log10(bledy[n-1]) << endl;
        delete[] bledy;
        usun_macierz(numeryczne, n);
        usun_macierz(analityczne, n);
    }
    plik.close();
}
void laasonen_gs_bledy_h() {
    double kroki[4] = {0.25, 0.1, 0.0625, 0.05};
    double h;
    double dt;
    int n, m;
    ofstream plik;
    plik.open("lgs_bledy_h.txt");
    cout << "lgs" << endl;
    for (int i = 0; i < 4; i++) {
        h = kroki[i];
        dt = (LAMBDA * h * h) / D;
        n = (T_MAX - T_MIN) / dt;
        m = (KONIEC_PRZEDZIALU - POCZATEK_PRZEDZIALU) / h;
        double** analityczne = oblicz_rozwiazanie_analityczne(h, dt, n, m);
        double** numeryczne = laasonen_gs_pom(h, n, m);
        double* bledy = policz_bledy(analityczne, numeryczne, n, m);
        plik << log10(h) << "\t" << log10(bledy[n - 1]) << endl;
        delete[] bledy;
        usun_macierz(numeryczne, n);
        usun_macierz(analityczne, n);
    }
    plik.close();
}
void crank_nicolson_thomas_bledy_h() {
    double kroki[4] = {0.25, 0.1, 0.0625, 0.05};
    double h;
    double dt;
    int n, m;
    ofstream plik;
    plik.open("cnt_bledy_h.txt");
```

```
    cout << "cnt" << endl;
    for (int i = 0; i < 4; i++) {
        h = kroki[i];
        dt = (LAMBDA * h * h) / D;
        n = (T_MAX - T_MIN) / dt;
        m = (KONIEC_PRZEDZIALU - POCZATEK_PRZEDZIALU) / h;
        double** analityczne = oblicz_rozwiazanie_analityczne(h, dt, n, m);
        double** numeryczne = crank_nicolson_thomas_pom(h, n, m);
        double* bledy = policz_bledy(analityczne, numeryczne, n, m);
        plik << log10(h) << "\t" << log10(bledy[n - 1]) << endl;
        delete[] bledy;
        usun_macierz(numeryczne, n);
        usun_macierz(analityczne, n);
    }
    plik.close();
}
void crank_nicolson_gs_bledy_h() {
    double kroki[4] = {0.25, 0.1, 0.0625, 0.05};
    double h;
    double dt;
    int n, m;
    ofstream plik;
    plik.open("cngs_bledy_h.txt");
    cout << "cngs" << endl;
    for (int i = 0; i < 4; i++) {
        h = kroki[i];
        dt = (LAMBDA * h * h) / D;
        n = (T_MAX - T_MIN) / dt;
        m = (KONIEC_PRZEDZIALU - POCZATEK_PRZEDZIALU) / h;
        double** analityczne = oblicz_rozwiazanie_analityczne(h, dt, n, m);
        double** numeryczne = crank_nicolson_gs_pom(h, n, m);
        double* bledy = policz_bledy(analityczne, numeryczne, n, m);
        plik << log10(h) << "\t" << log10(bledy[n - 1]) << endl;
        delete[] bledy;
        usun_macierz(numeryczne, n);
        usun_macierz(analityczne, n);
    }
    plik.close();
}
int main() {
    laasonen();
    crank_nicolson();
    laasonen_thomas_bledy_h();
    laasonen_gs_bledy_h();
    crank_nicolson_thomas_bledy_h();
    crank_nicolson_gs_bledy_h();
    return 0;
}
```