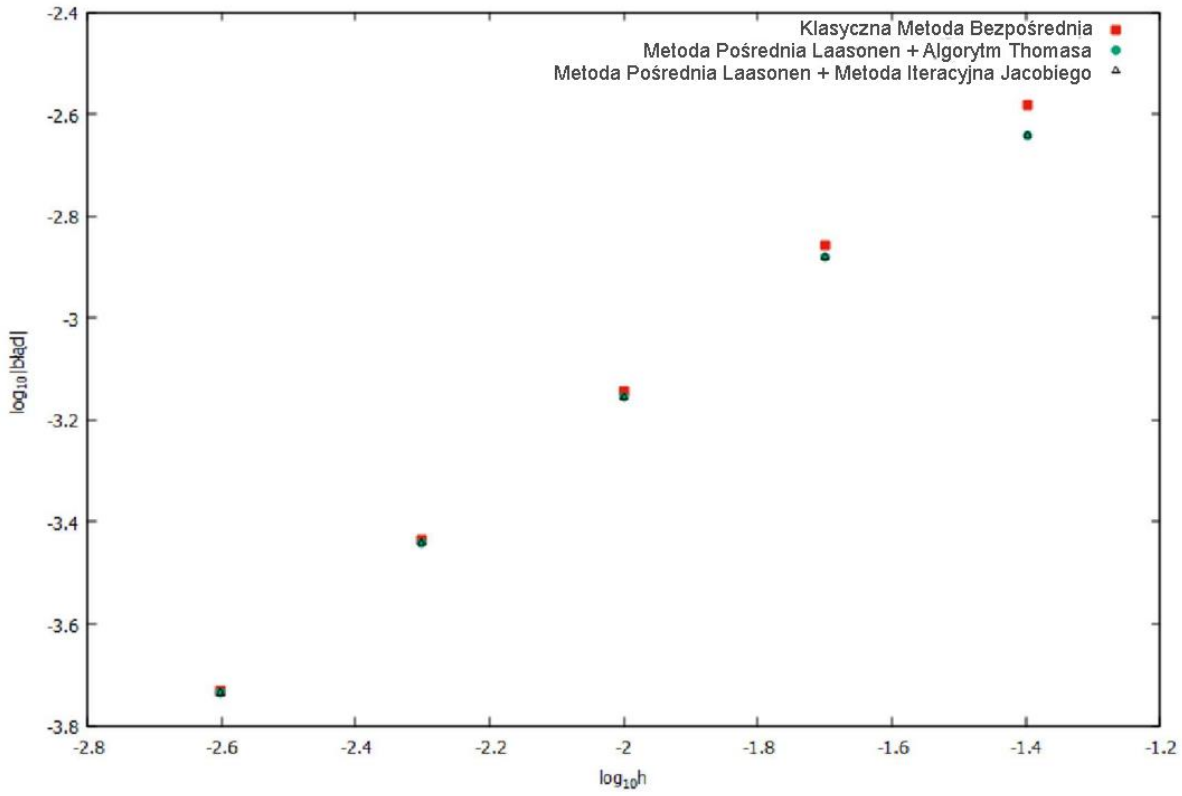
Wykres zależności maksymalnej wartości bezwzględnej błędu dla tmax w funkcji kroku przestrzennego h.



Na wykresie zostały przedstawione maksymalne wartości bezwzględne błędów bezwzględnych obliczonych rozwiązań przy pomocy wszystkich wykorzystanych metod w funkcji kroku przestrzennego. Wykres został wykonany w skali logarytmicznej, na osi x znajdują się logarytmy dziesiętne kroku siatki przestrzennej, natomiast na osi y logarytmy dziesiętne wartości bezwzględnej maksymalnego błędu bezwzględnego rozwiązań.

Na podstawie wykresu można zauważyć, że dla metody pośredniej Laasonen z wykorzystaniem obu metod rozwiązywania układów równań liniowych maksymalny błąd rozwiązań jest bardzo zbliżony (na wykresie wartości się pokrywają), natomiast w przypadku Klasycznej Metody Bezpośredniej wraz ze wzrostem kroku siatki przestrzennej maksymalny błąd rozwiązania jest coraz większy w odniesieniu do metody Laasonen.

Obliczony rząd dokładności jako współczynnik nachylenia prostej:

- Klasyczna Metoda

$$\frac{-3.43584-(-2.85644)}{-2.30103-(-1.69897)} \approx 0.96$$

- Metoda pośrednia Laasonen z wykorzystaniem algorytmu Thomasa

$$\frac{-3.44210-(-2.88088)}{-2.30103-(-1.69897)} \approx 0.93$$

- Metoda pośrednia Laasonen z wykorzystaniem metody iteracyjnej Jacobiego

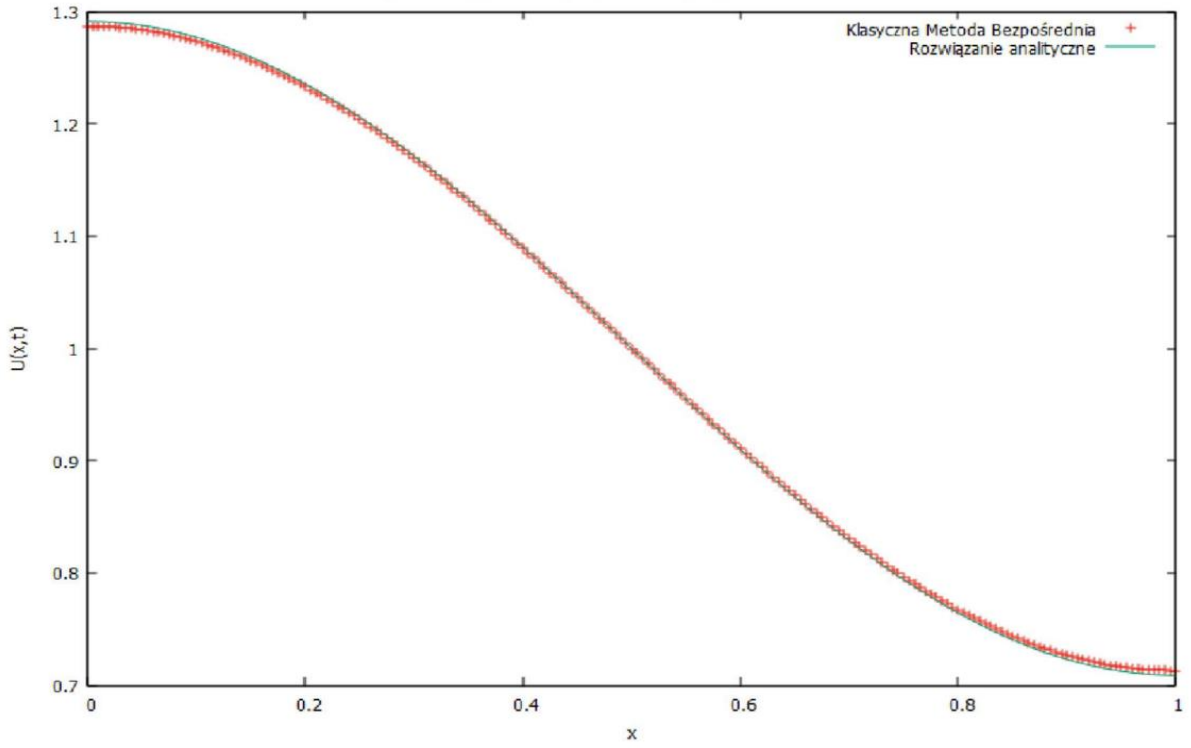
$$\frac{-3.44192-(-2.88087)}{-2.30103-(-1.69897)} \approx 0.93$$

Dokładność wszystkich rozpatrywanych metod jest w przybliżeniu pierwszego rzędu, natomiast teoretyczne rzędy dokładności dla tych metod względem kroku siatki przestrzennej wynoszą dwa. Ta różnica może wynikać z rzędu dokładności przybliżeń dwupunktowych zastosowanych do dyskretyzacji warunków brzegowych (zastosowana tam różnica progresywna i wsteczna są pierwszego rzędu dokładności) oraz ze źle dobranego kroku przestrzennego dla tego problemu.

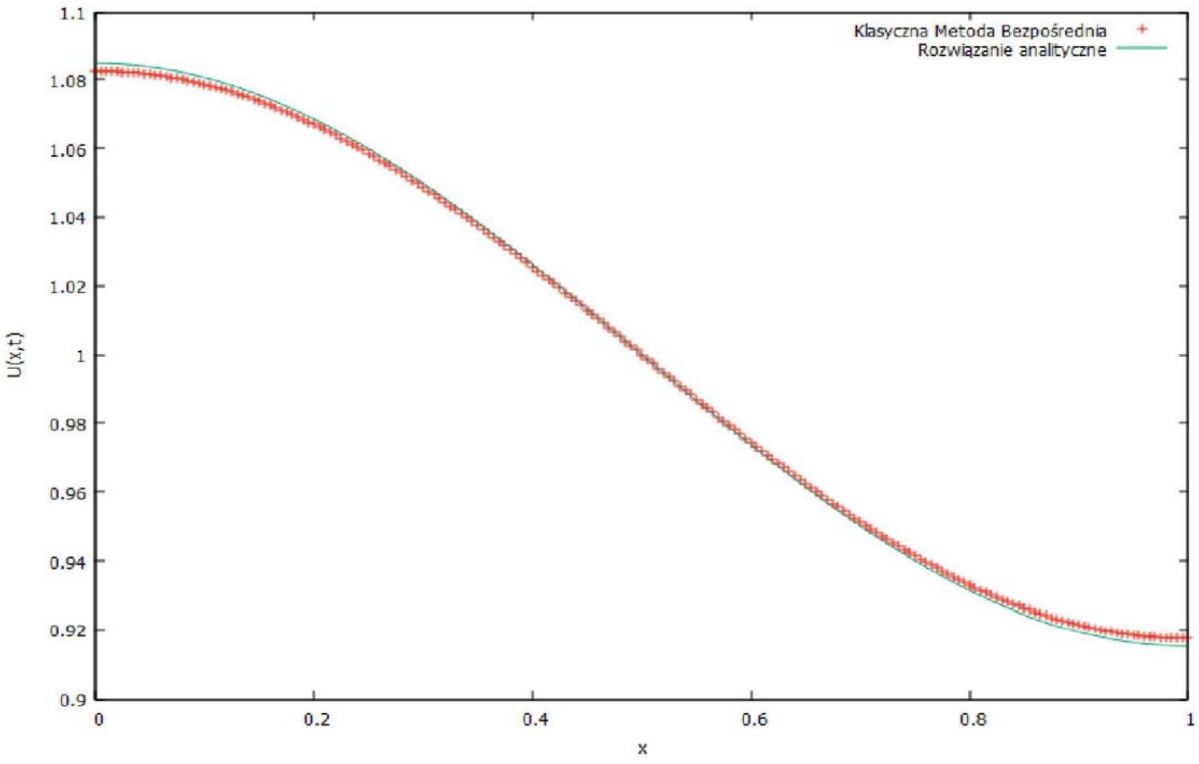
Wykresy rozwiązań numerycznych i analitycznych

Klasyczna metoda bezpośrednia, dla kroku siatki przestrzennej h = 0.005 i kroku czasowego δt = 0.00001

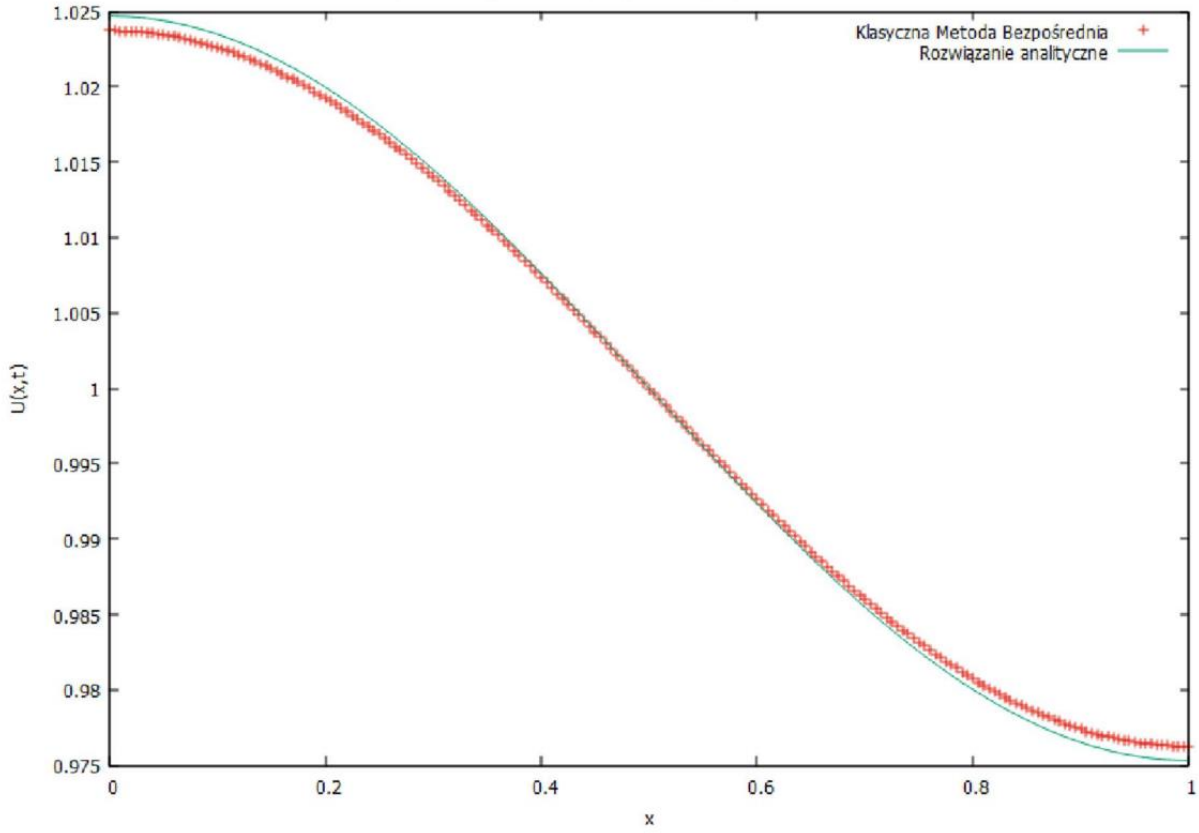
Wykres rozwiązań numerycznych i analitycznych dla t=0.125



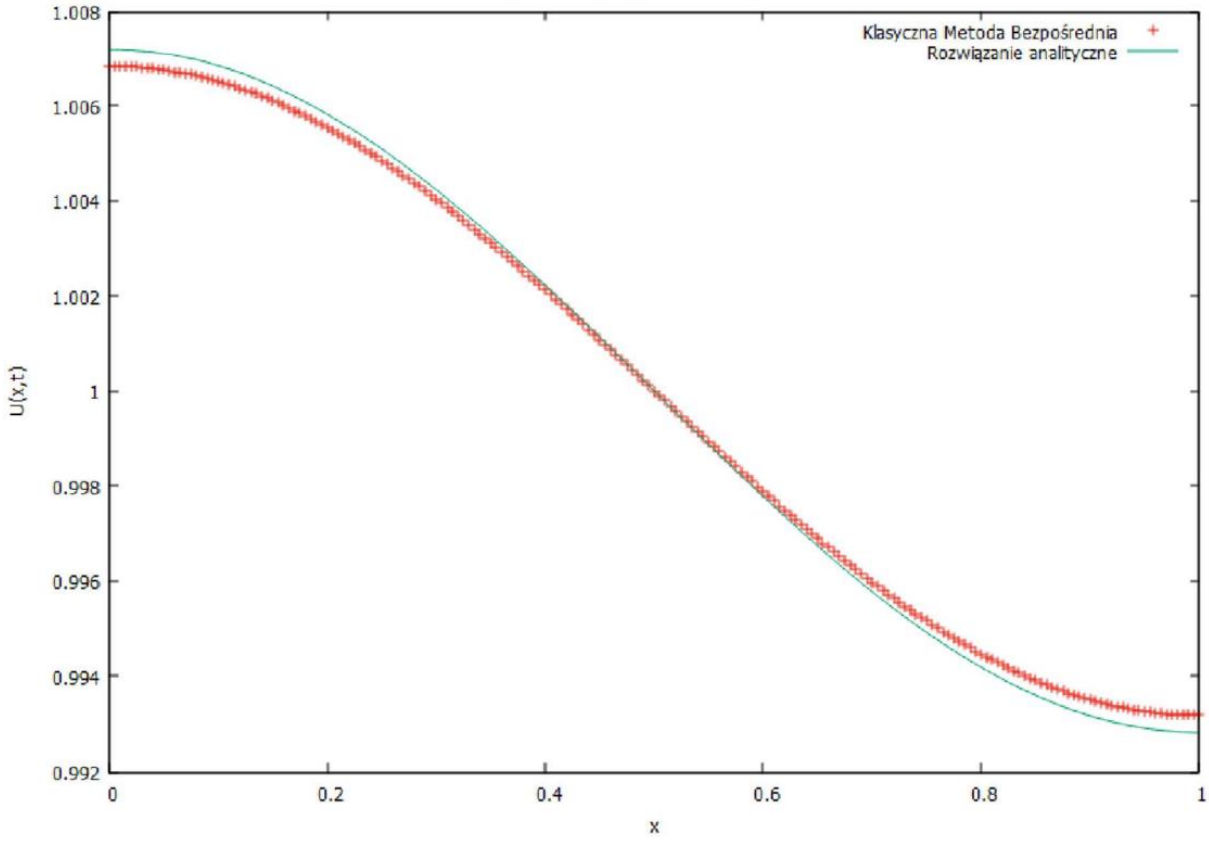
Wykres rozwiązań numerycznych i analitycznych dla t=0.25



Wykres rozwiązań numerycznych i analitycznych dla t=0.375

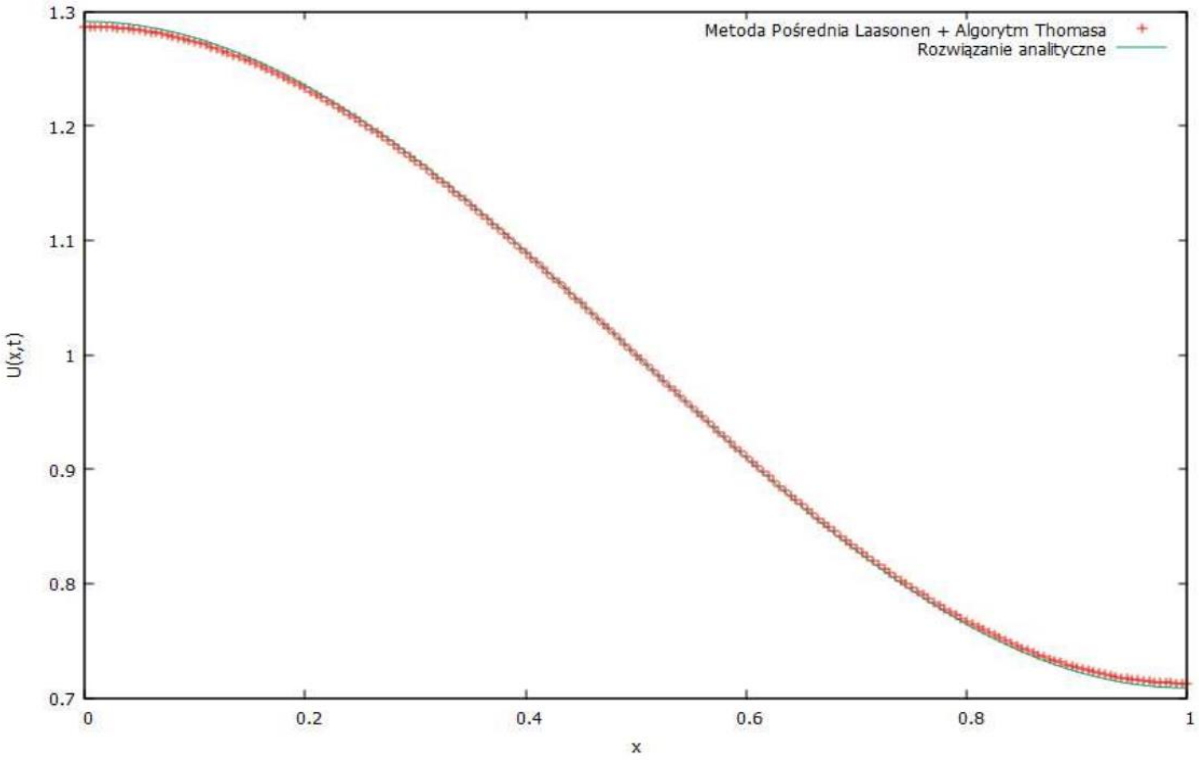


Wykres rozwiązań numerycznych i analitycznych dla t=0.5

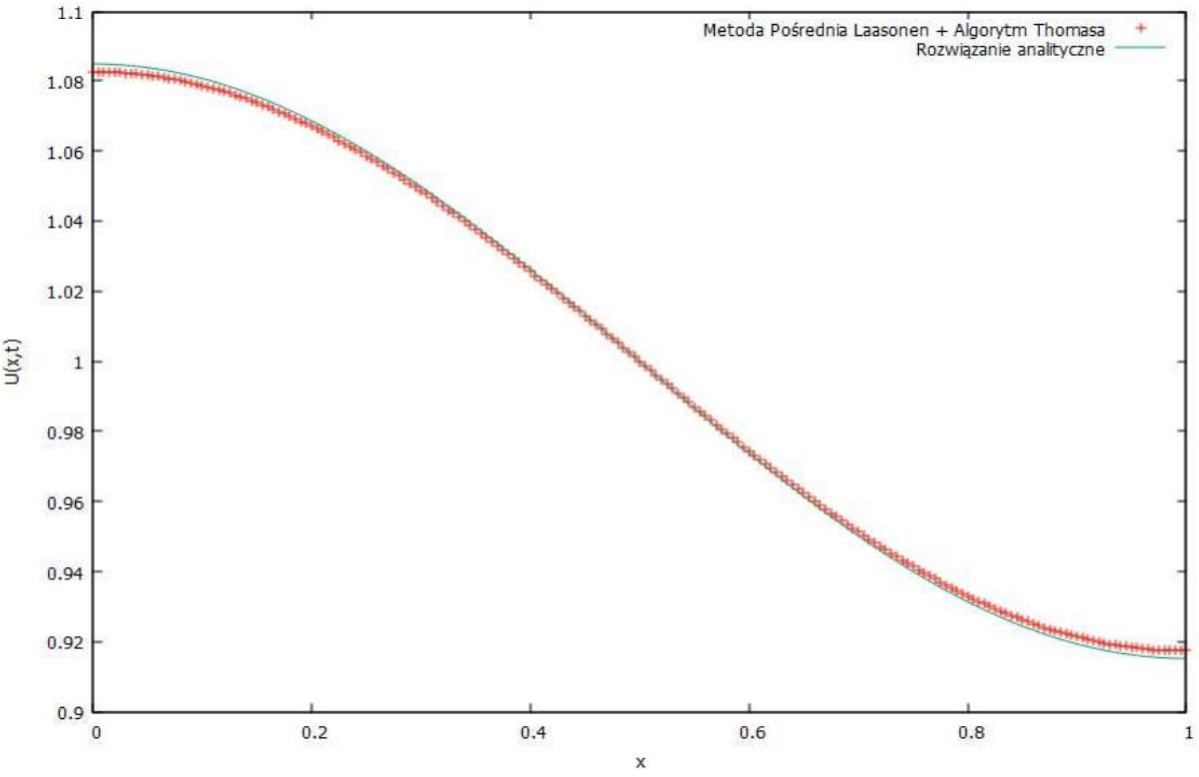


Metoda Laasonen z wykorzystaniem algorytmu Thomasa, dla kroku siatki przestrzennej h = 0.005 i kroku czasowego δt = 0.000025

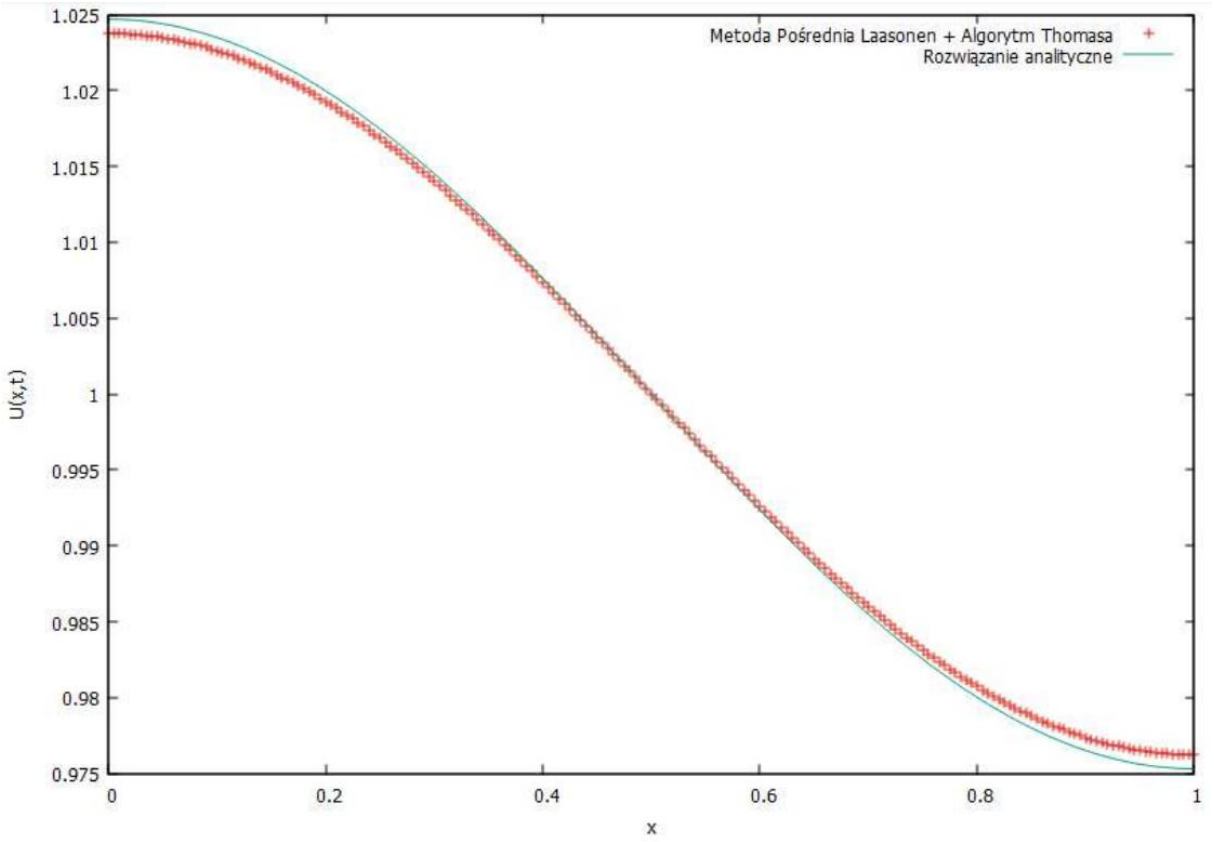
Wykres rozwiązań numerycznych i analitycznych dla t=0.125



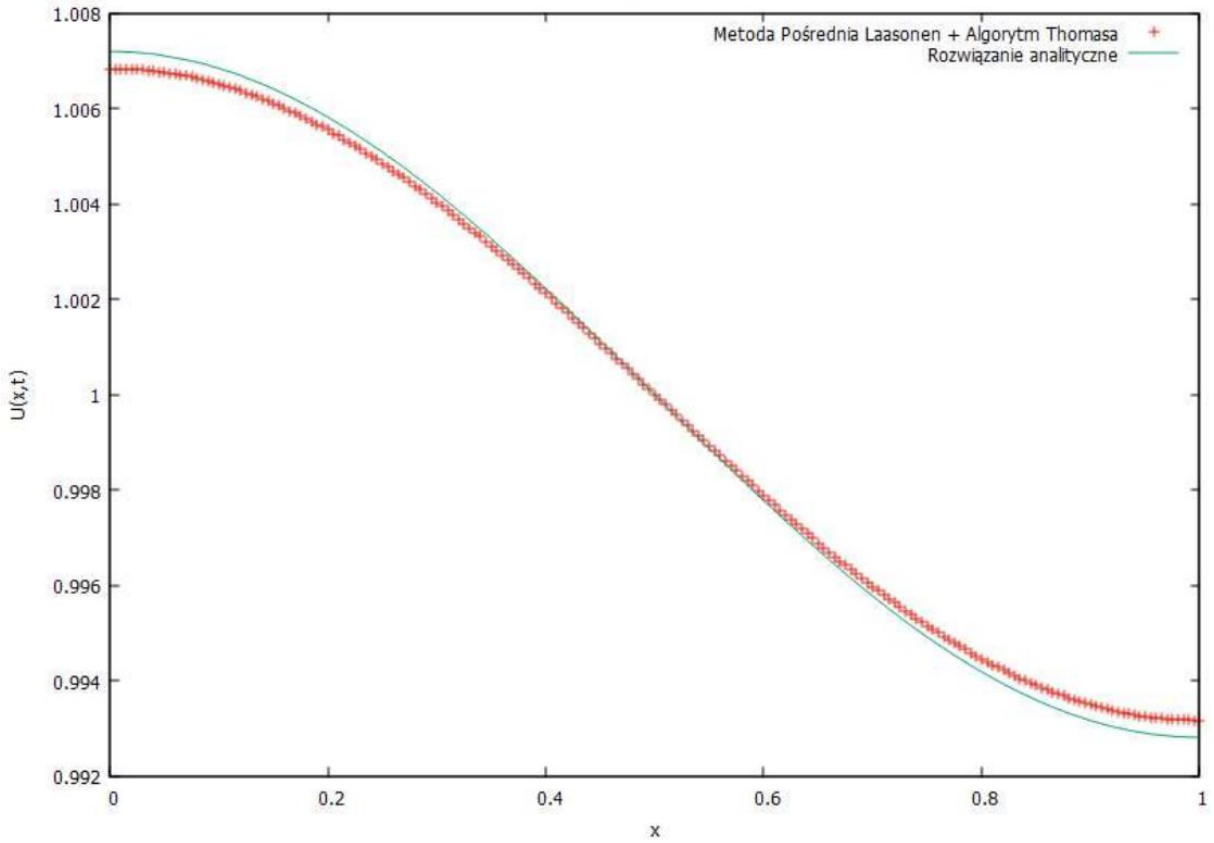
Wykres rozwiązań numerycznych i analitycznych dla t=0.25



Wykres rozwiązań numerycznych i analitycznych dla t=0.375

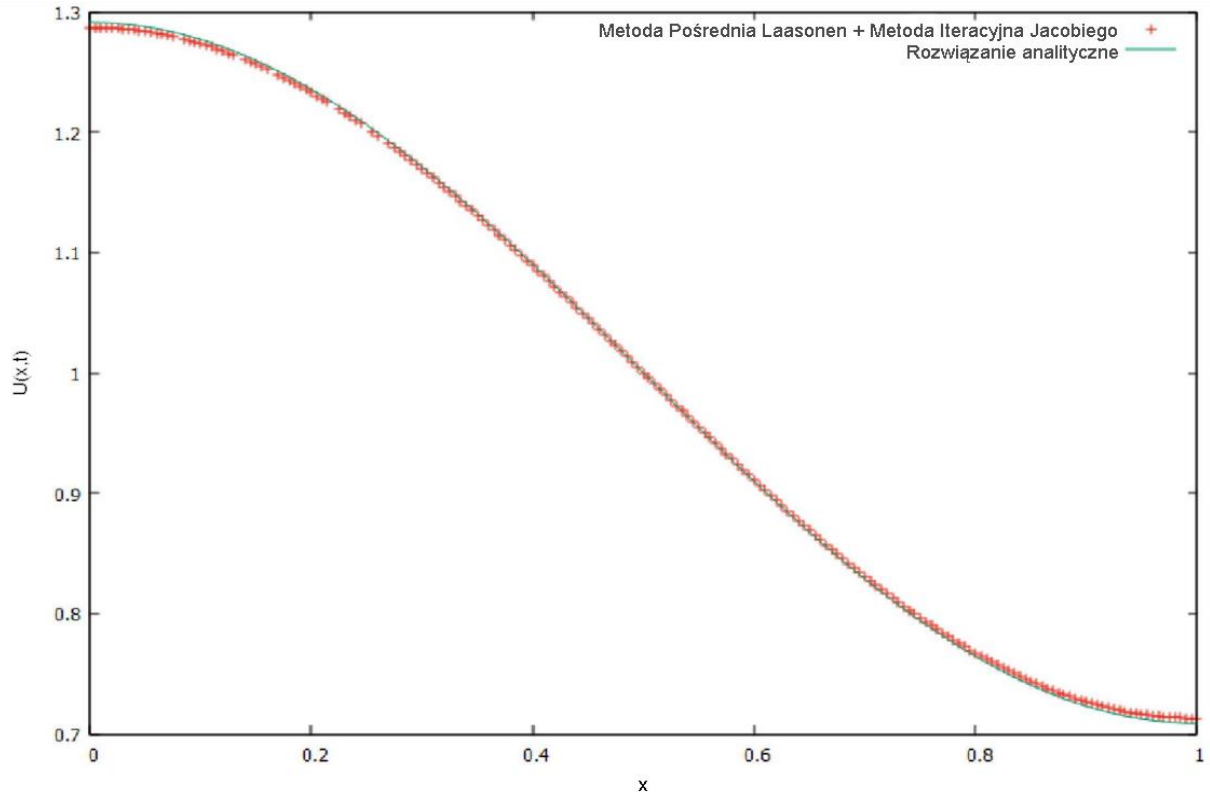


Wykres rozwiązań numerycznych i analitycznych dla t=0.5

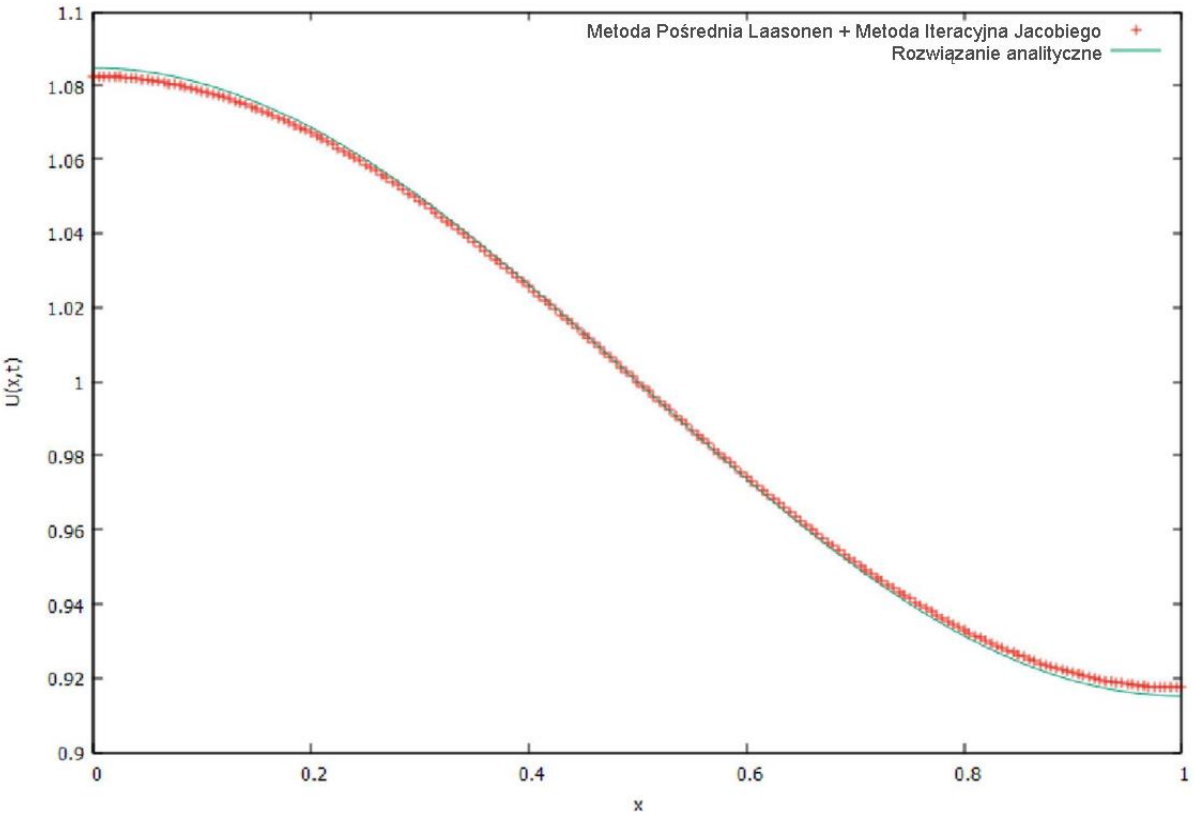


Metoda Laasonen z wykorzystaniem metody iteracyjnej Jacobiego, dla kroku siatki przestrzennej h = 0.005 i kroku czasowego δt = 0.000025

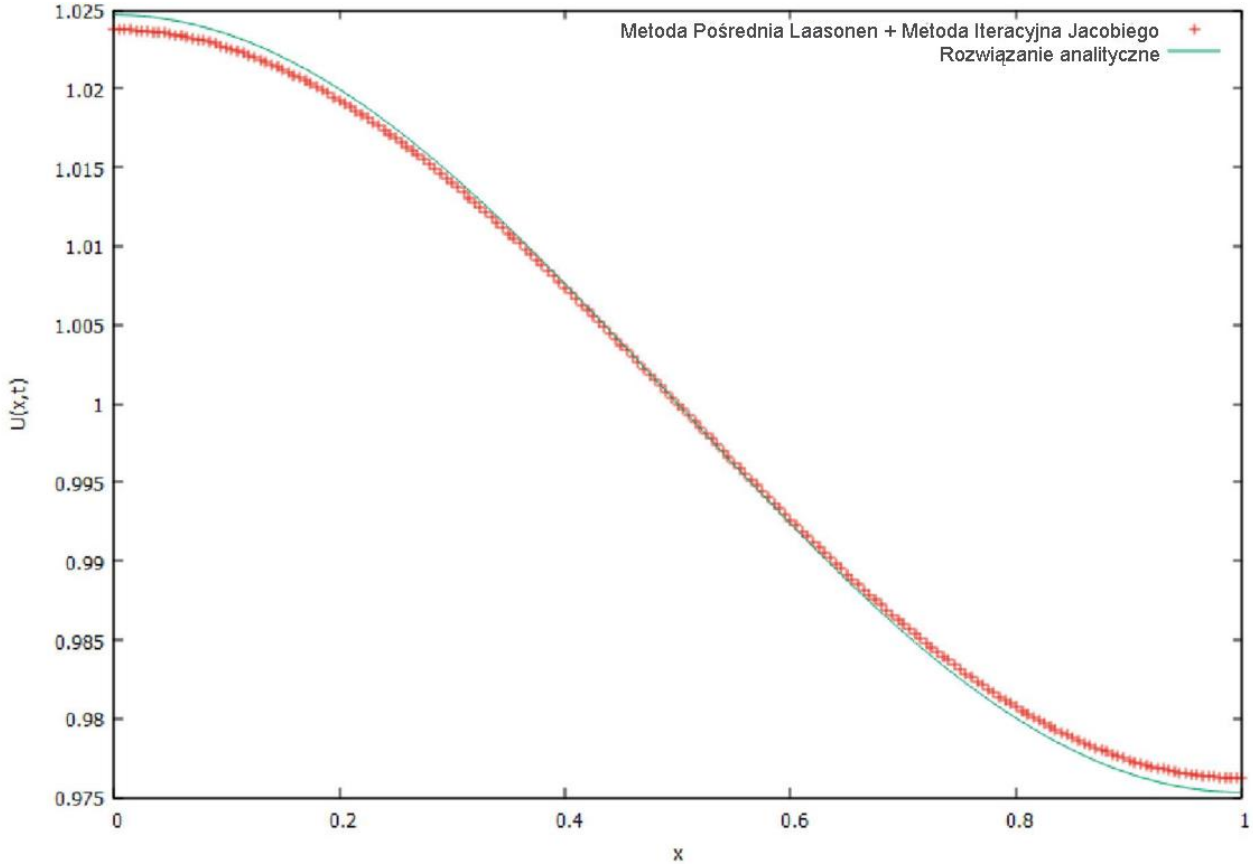
Wykres rozwiązań numerycznych i analitycznych dla t=0.125



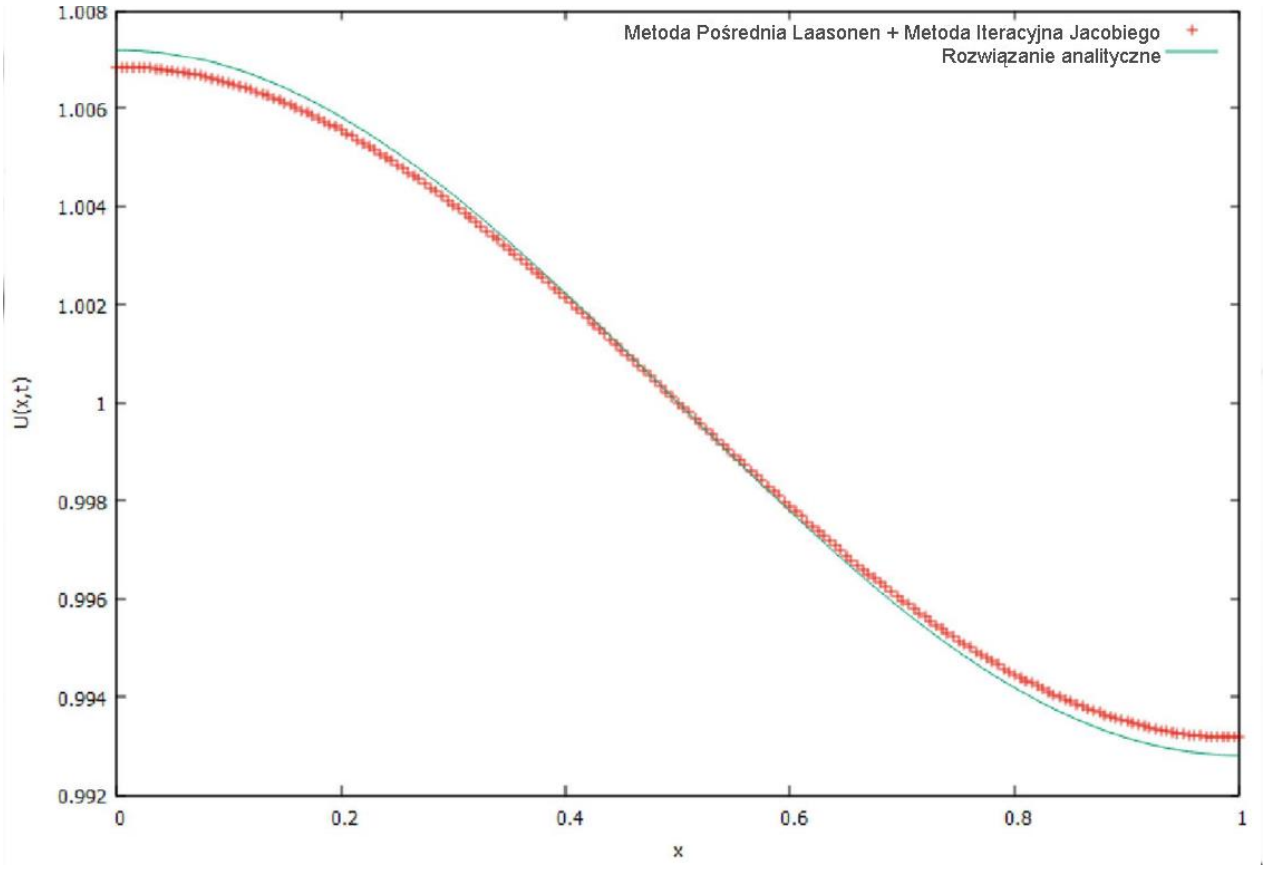
Wykres rozwiązań numerycznych i analitycznych dla t=0.25



Wykres rozwiązań numerycznych i analitycznych dla t=0.375

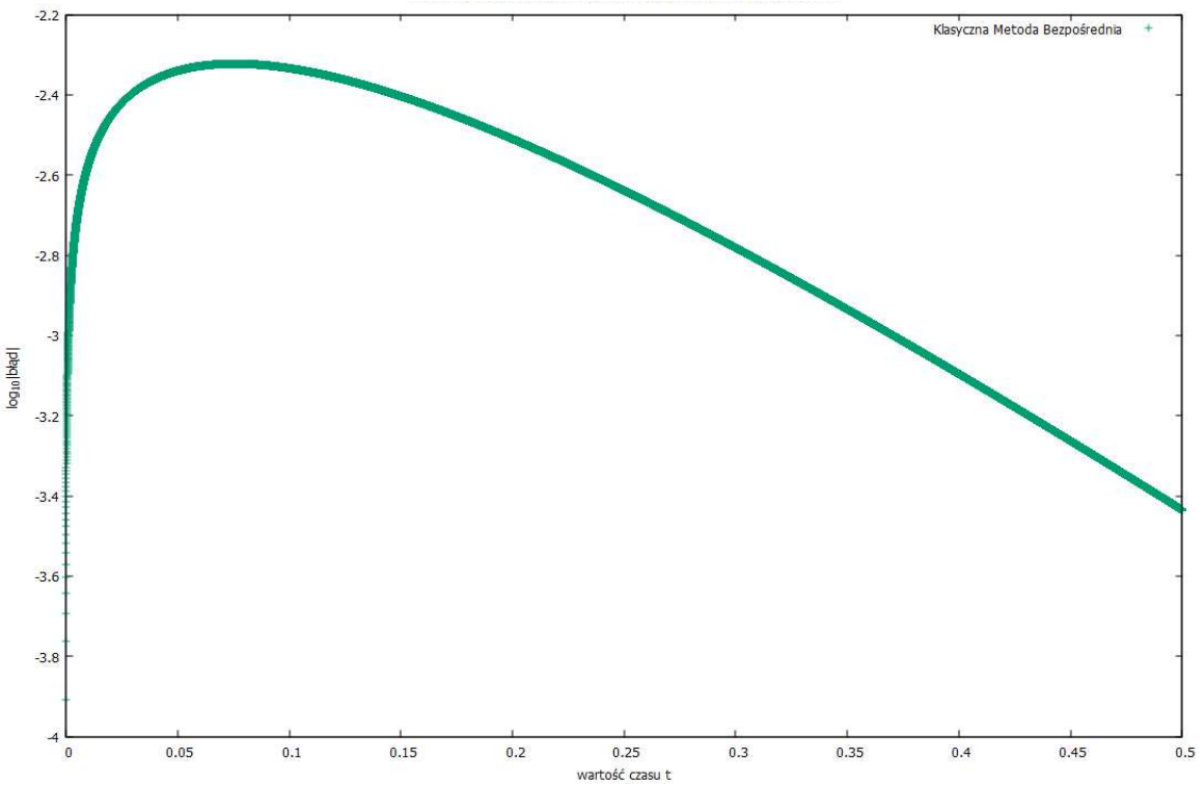


Wykres rozwiązań numerycznych i analitycznych dla t=0.5

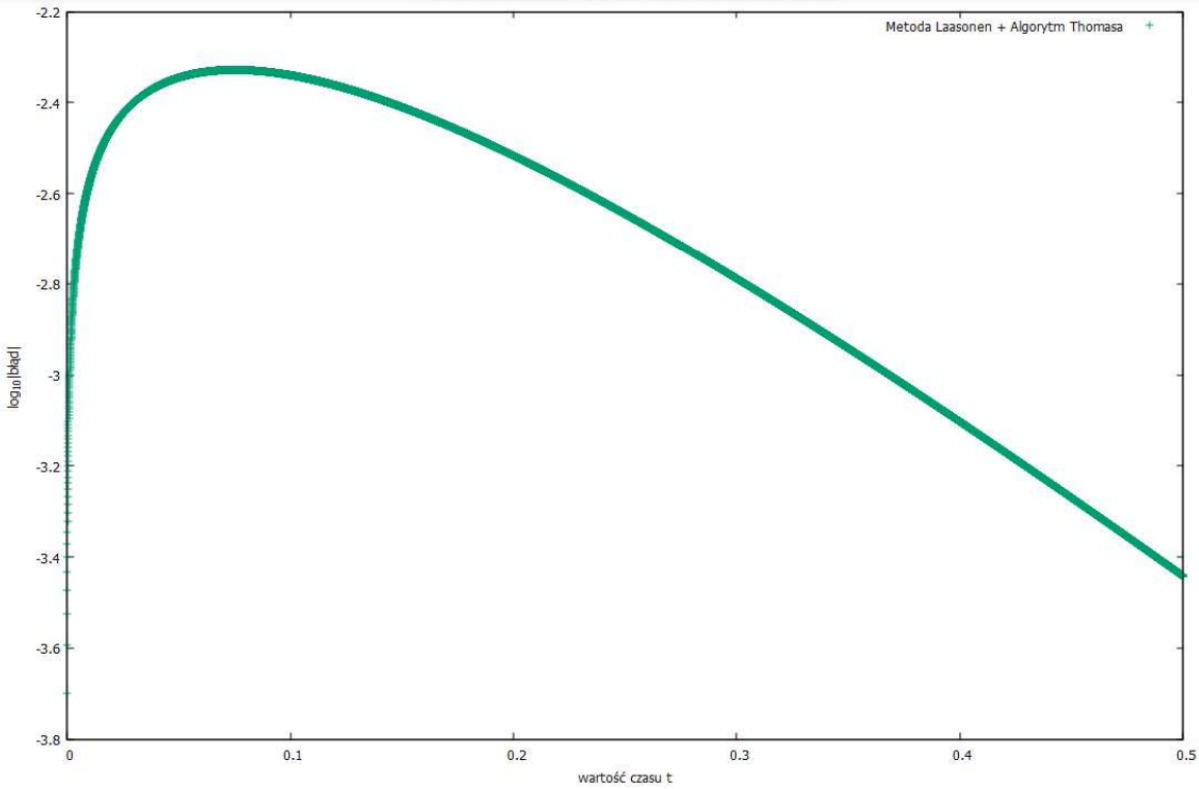


Wykresy maksymalnej wartości bezwzględnej błędu w funkcji czasu

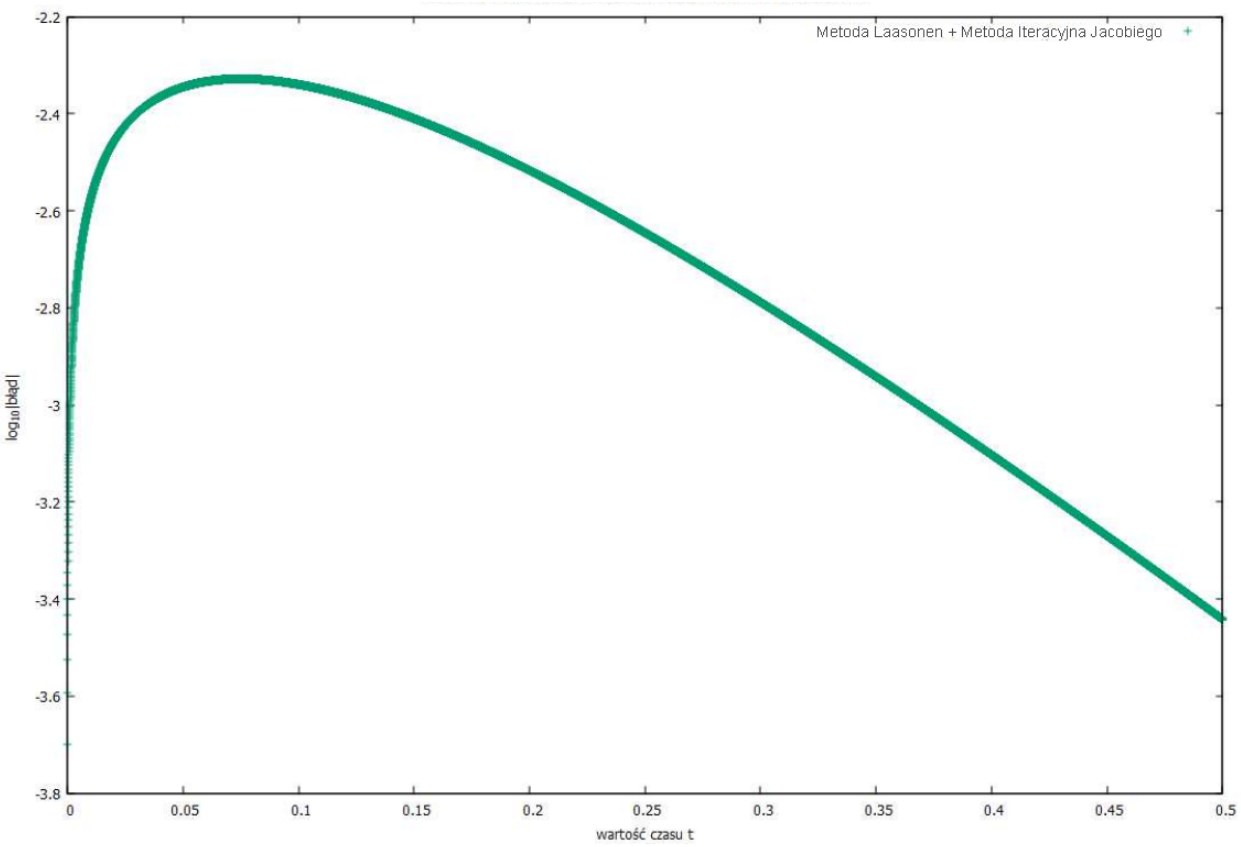
Zależność maksymalnej wartości bezwzględnej błędu w funkcji czasu t



Zależność maksymalnej wartości bezwzględnej błędu w funkcji czasu t



Zależność maksymalnej wartości bezwzględnej błędu w funkcji czasu t



Jak widać na powyższych wykresach, błąd maksymalny względem funkcji czasu jest bardzo zbliżony dla wszystkich rozpatrywanych metod. Jego wartość maleje, co może wynikać z faktu, że wraz z upływem czasu wartości rozwiązań równania zbiegają do jedynki.

Kod programu:

```
#include <iostream>
#include <fstream>
#include <cmath>
#include <vector>
#include <string>

#define M_PI 3.14159265358979323846

using namespace std;

const double T_MAX = 0.5;
const double D = 1.0;
const double ALPHA = 1.0, BETA = 0.0, GAMMA = 0.0, PHI = 1.0, PSI = 0.0, THETA =
0.0;
const double X_MIN = 0.0, X_MAX = 1.0;

double analytic(double x, double t)
{
    return 1.0 + exp(-pow(M_PI, 2) * D * t) * cos(M_PI * x);
```

```
}

double initial_condition(double x)
{
    return 1.0 + cos(M_PI * x);
}

void printMatrix(const vector<vector<double>>& matrix)
{
    std::cout << "printMatrix" << std::endl;

    for (vector<double> row : matrix)
    {
        for (double element : row)
            std::cout << element << " ";

        std::cout << std::endl;
    }
}

void printVector(const vector<double>& vector)
{
    std::cout << "printVector" << std::endl;

    for (double element : vector)
        std::cout << element << " ";

    std::cout << std::endl;
}

void save_values(const vector<vector<double>>& U, double dt, double h, int i,
const string& filename)
{
    double t = i * dt;
    ofstream file(filename);
    file << "x,calculated,analytic,t=" << t << endl;
    for (int j = 0; j < U[0].size(); j++)
    {
        double x = j * h;
        file << x << "," << U[i][j] << "," << analytic(x, t) << endl;
    }
    file.close();
}

vector<vector<double>> getU(int n, int m, double h)
{
    vector<vector<double>> matrix(n, vector<double>(m, 0.0));

    matrix[0][0] = initial_condition(X_MIN);
    for (int i = 1; i < m - 1; i++)
        matrix[0][i] = initial_condition(i * h);

    matrix[0][m - 1] = initial_condition(X_MAX);
    return matrix;
}

vector<vector<double>> getA(int n, double h, double lambda)
{
    vector<vector<double>> A(n, vector<double>(n, 0.0));
    A[0][0] = -ALPHA / h + BETA;
    A[0][1] = ALPHA / h;
    for (int i = 1; i < n - 1; i++)
    {
```

```
        A[i][i - 1] = lambda;
        A[i][i] = -(1.0 + 2.0 * lambda);
        A[i][i + 1] = lambda;
    }
    A[n - 1][n - 2] = -PHI / h;
    A[n - 1][n - 1] = PHI / h + PSI;
    return A;
}

std::vector<std::vector<double>> getLDU(int n, double h, double lambda)
{
    std::vector<std::vector<double>> LDU(n, std::vector<double>(3, 0.0));

    LDU[0][1] = -ALPHA / h + BETA;
    LDU[0][2] = ALPHA / h;

    for (int i = 1; i < n - 1; i++)
    {
        LDU[i][0] = lambda;
        LDU[i][1] = -(1.0 + 2.0 * lambda);
        LDU[i][2] = lambda;
    }

    LDU[n - 1][0] = -PHI / h;
    LDU[n - 1][1] = PHI / h + PSI;

    return LDU;
}

std::vector<std::vector<double>> KMB(double dt, double h)
{
    double lambda = D * dt / (h * h);
    int N = T_MAX / dt + 1.0;
    int M = (X_MAX - X_MIN) / h + 1.0;

    std::vector<std::vector<double>> U = getU(N, M, h);
    for (int i = 1; i < N; i++)
    {
        for (int j = 1; j < M - 1; j++)
            U[i][j] = lambda * U[i - 1][j - 1] + (1.0 - 2.0 * lambda) * U[i -
1][j] + lambda * U[i - 1][j + 1];

        U[i][0] = U[i][1];
        U[i][M - 1] = U[i][M - 2];
    }

    return U;
}

void thomas_transform(std::vector<std::vector<double>>& LDU)
{
    for (int i = 1; i < LDU.size(); i++)
        LDU[i][1] -= LDU[i][0] * LDU[i - 1][2] / LDU[i - 1][1];
}

std::vector<double> thomas_solve(const std::vector<std::vector<double>>& LDU,
const std::vector<double>& b)
{
    int N = b.size();

    std::vector<double> temp_b = b;

    for (int i = 1; i < N; i++)
```

```
        temp_b[i] -= (LDU[i][0] * temp_b[i - 1]) / LDU[i - 1][1];

    std::vector<double> x(N);

    x[N - 1] = temp_b[N - 1] / LDU[N - 1][1];
    for (int i = N - 2; i >= 0; i--)
        x[i] = (temp_b[i] - LDU[i][2] * x[i + 1]) / LDU[i][1];

    return x;
}

std::vector<std::vector<double>> ML_Thomas(double dt, double h)
{
    double lambda = D * dt / (h * h);

    int N = T_MAX / dt + 1.0;
    int M = (X_MAX - X_MIN) / h + 1.0;

    std::vector<std::vector<double>> U = getU(N, M, h);
    std::vector<std::vector<double>> LDU = getLDU(M, h, lambda);
    thomas_transform(LDU);

    for (int k = 1; k < N; k++)
    {
        std::vector<double> b(M);
        b[0] = -GAMMA;
        for (int i = 1; i < M - 1; i++)
            b[i] = -U[k - 1][i];
        b[M - 1] = -THETA;
        std::vector<double> u_k_next = thomas_solve(LDU, b);
        for (int i = 0; i < M; i++)
            U[k][i] = u_k_next[i];
    }
    return U;
}

std::vector<std::vector<double>> ML_Jacobi(double dt, double h)
{
    double lambda = D * dt / (h * h);
    int N = T_MAX / dt + 1.0;
    int M = (X_MAX - X_MIN) / h + 1.0;

    std::vector<std::vector<double>> U = getU(N, M, h);
    std::vector<std::vector<double>> U_new = U;

    double residual = 0.0;
    double error_estimator = 0.0;

    int max_iterations = 15;
    double tolerance = 1e-5;

    for (int iteration = 0; iteration < max_iterations; iteration++)
    {
        residual = 0.0;

        for (int k = 1; k < N; k++)
        {
            for (int i = 1; i < M - 1; i++)
            {
                U_new[k][i] = (lambda * U[k - 1][i - 1] + (1.0 - 2.0 * lambda) *
U[k - 1][i] + lambda * U[k - 1][i + 1]);
            }
```

```
            U_new[k][0] = U_new[k][1];
            U_new[k][M - 1] = U_new[k][M - 2];
        }

        for (int k = 0; k < N; k++)
        {
            for (int i = 0; i < M; i++)
            {
                residual += std::abs(U_new[k][i] - U[k][i]);
                error_estimator += std::abs(U_new[k][i]);
            }
        }

        if (residual < tolerance && error_estimator < tolerance)
        {
            U = U_new;
            break;
        }

        U = U_new;
    }

    return U;
}

void errors_t(double dt, double h, const std::vector<std::vector<double>>& U,
const std::string& filename)
{
    std::ofstream error_t_file;
    error_t_file.open(filename);
    error_t_file << "t,max_error,h=" << h << ",dt=" << dt << std::endl;

    for (int i = 0; i < U.size(); i++)
    {
        double t = dt * i;
        double max_error = 0.0;
        for (int j = 0; j < U[i].size(); j++)
        {
            double x = h * j;
            double error = std::abs(U[i][j] - analytic(x, t));
            if (max_error < error)
                max_error = error;
        }

        error_t_file << t << "," << max_error << std::endl;
    }

    error_t_file.close();
}

int main()
{
    cout << "zadanie 3" << endl;
    // zadanie 3
    double dt = 0.000025, dt_kmb = 0.00001;
    double h = 0.005;

    vector<vector<double>> kmb = KMB(dt_kmb, h);
    vector<vector<double>> thomas = ML_Thomas(dt, h);
    vector<vector<double>> jacobi = ML_Jacobi(dt, h);

    errors_t(dt_kmb, h, kmb, "/kmb_error_t.csv");
    errors_t(dt, h, thomas, "thomas_error_t.csv");
```

```
    errors_t(dt, h, jacobi, "jacobi_error_t.csv");

    cout << "zadanie 2" << endl;
    // zadanie 2
    save_values(kmb, dt_kmb, h, kmb.size() / 4.0, "kmb_1_t.csv");
    save_values(kmb, dt_kmb, h, 2 * kmb.size() / 4.0, "kmb_2_t.csv");
    save_values(kmb, dt_kmb, h, 3 * kmb.size() / 4.0, "kmb_3_t.csv");
    save_values(kmb, dt_kmb, h, kmb.size() - 1.0, "kmb_4_t.csv");

    save_values(thomas, dt, h, thomas.size() / 4.0, "thomas_1_t.csv");
    save_values(thomas, dt, h, 2 * thomas.size() / 4.0, "thomas_2_t.csv");
    save_values(thomas, dt, h, 3 * thomas.size() / 4.0, "thomas_3_t.csv");
    save_values(thomas, dt, h, thomas.size() - 1.0, "thomas_4_t.csv");

    save_values(jacobi, dt, h, jacobi.size() / 4.0, "jacobi_1_t.csv");
    save_values(jacobi, dt, h, 2 * jacobi.size() / 4.0, "jacobi_2_t.csv");
    save_values(jacobi, dt, h, 3 * jacobi.size() / 4.0, "jacobi_3_t.csv");
    save_values(jacobi, dt, h, jacobi.size() - 1.0, "jacobi_4_t.csv");

    cout << "zadanie 1" << endl;
    // zadanie 1
    ofstream errors_h;
    errors_h.open("errors_h.csv");
    errors_h << "h,kmb,lu,thomas" << endl;

    h = 0.04;
    for (int _ = 0; _ < 5; _++)
    {
        dt = h * h;
        dt_kmb = 0.4 * h * h;
        kmb = KMB(dt_kmb, h);
        jacobi = ML_Jacobi(dt, h);
        thomas = ML_Thomas(dt, h);

        double max_error_kmb = 0.0;
        for (int i = 0; i < kmb[0].size(); i++)
        {
            double error = abs(kmb[kmb.size() - 1][i] - analytic(i * h, T_MAX));
            if (max_error_kmb < error)
                max_error_kmb = error;
        }

        double max_error_jacobi = 0.0;
        for (int i = 0; i < jacobi[0].size(); i++)
        {
            double error = abs(jacobi[jacobi.size() - 1][i] - analytic(i * h,
T_MAX));
            if (max_error_jacobi < error)
                max_error_jacobi = error;
        }

        double max_error_thomas = 0.0;
        for (int i = 0; i < thomas[0].size(); i++)
        {
            double error = abs(thomas[thomas.size() - 1][i] - analytic(i * h,
T_MAX));
            if (max_error_thomas < error)
                max_error_thomas = error;
        }

        cout << h << "," << max_error_kmb << "," << max_error_jacobi << "," <<
max_error_thomas << endl;
```

```
        errors_h << h << "," << max_error_kmb << "," << max_error_jacobi << ","
<< max_error_thomas << endl;
        h /= 2.0;
    }
    errors_h.close();

    system("pause");
    return 0;
}
```