Laboratorium 11

Artem Buhera 15.06.2021
GĆ01 135678

Na danym Laboratorium musieliśmy zaimplementować program do rozwiązywania równania różniczkowego cząstkowego $\frac{\partial U(x,t)}{\partial t} = D\frac{\partial^2 U(x,t)}{\partial x^2}$ dla współrzędnej przestrzennej $x \in [0,1]$ oraz czasu $t \in [0, t_{max}]$.

Warunki początkowe: $U(x,0) = 1 + \cos{(\pi x)}$

Warunki brzegowe: $\frac{\partial U(0,t)}{\partial x} = 0$ oraz $\frac{\partial U(1,t)}{\partial x} = 0$

Używane metody:

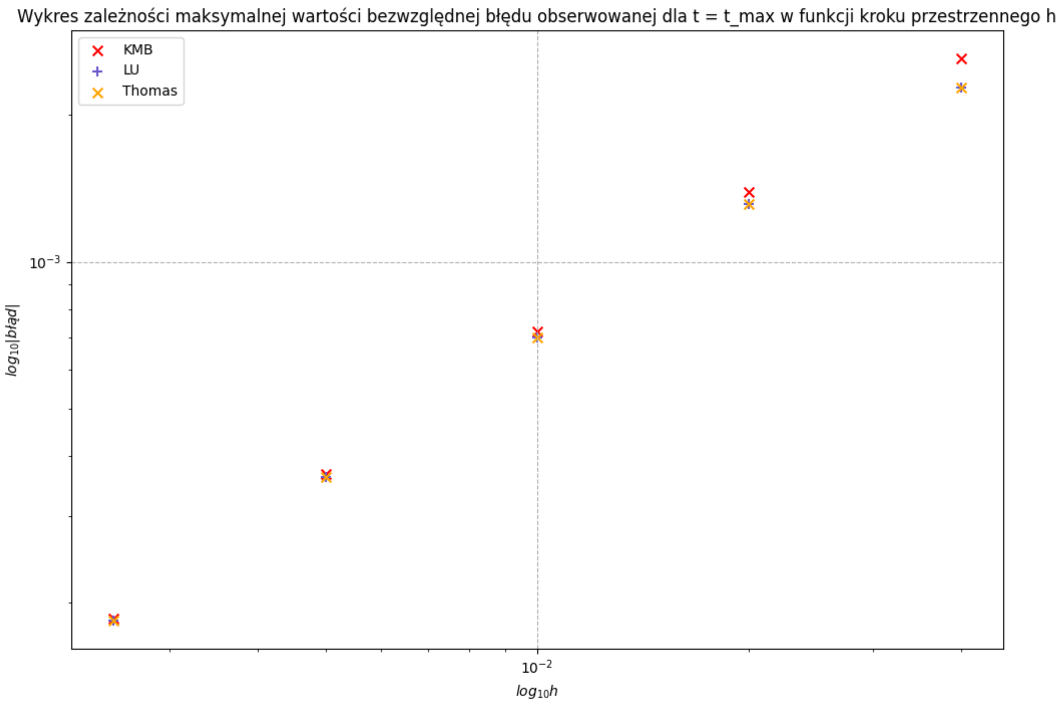
- Klasyczna metoda bezpośrednia
- Metoda Laasonen + dekompozycja LU
- Metoda Laasonen + algorytm Thomasa

Parametry:

- $t_{max} = 0$
- D = 1

Wyniki i wykresy

Zadanie 1

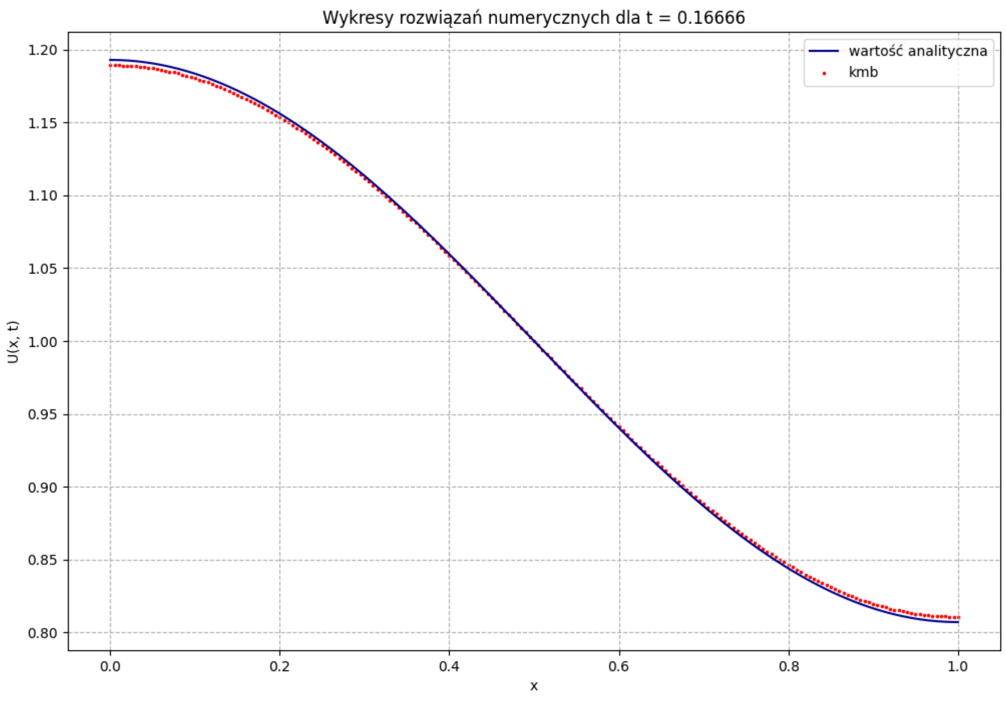


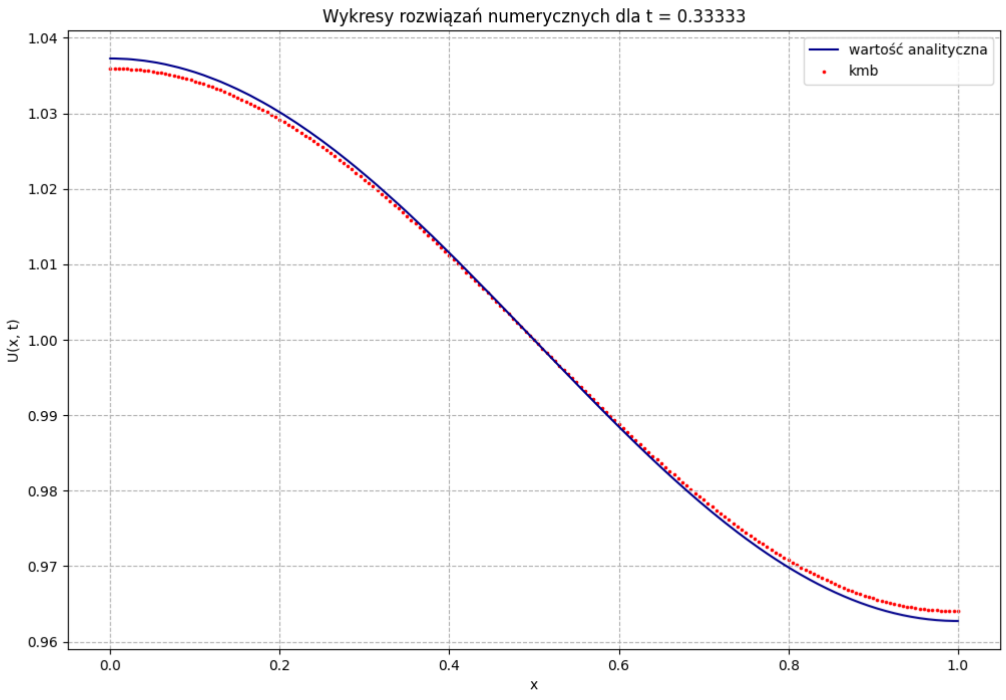
Na wykresach możemy zaobserować zgodność z rzędami teoretycznymi błędów na poziomie 2. Co więcej, dla wszystkich algorytmów otrzymujemy bardzo podobne błedy.

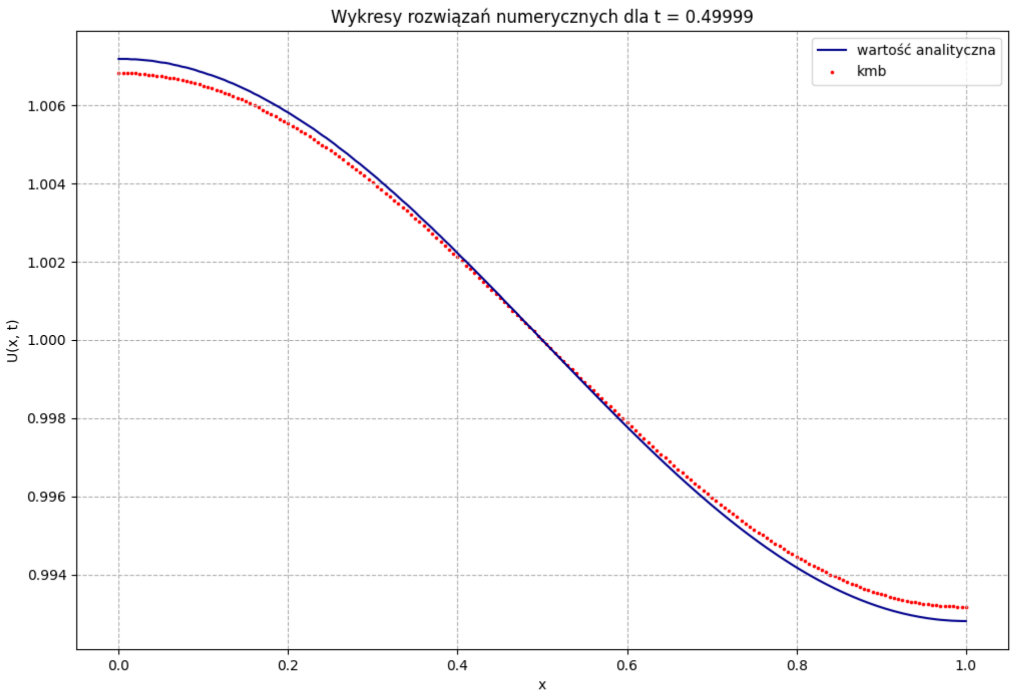
Zadanie 2

Poniżej są przedstawione wykresy wartości obliczonych poszczególnymi metodami w funkcji *x* dla *t* = $\frac{1}{3}t_{max}$, $\frac{2}{3}t_{max}$ oraz t_{max}.

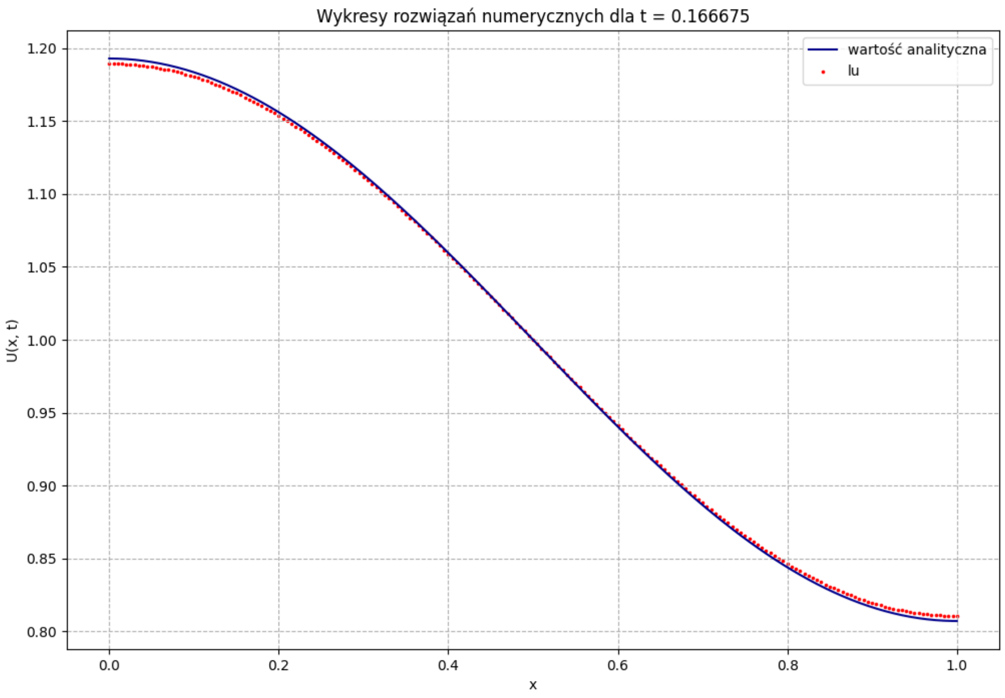
KMB

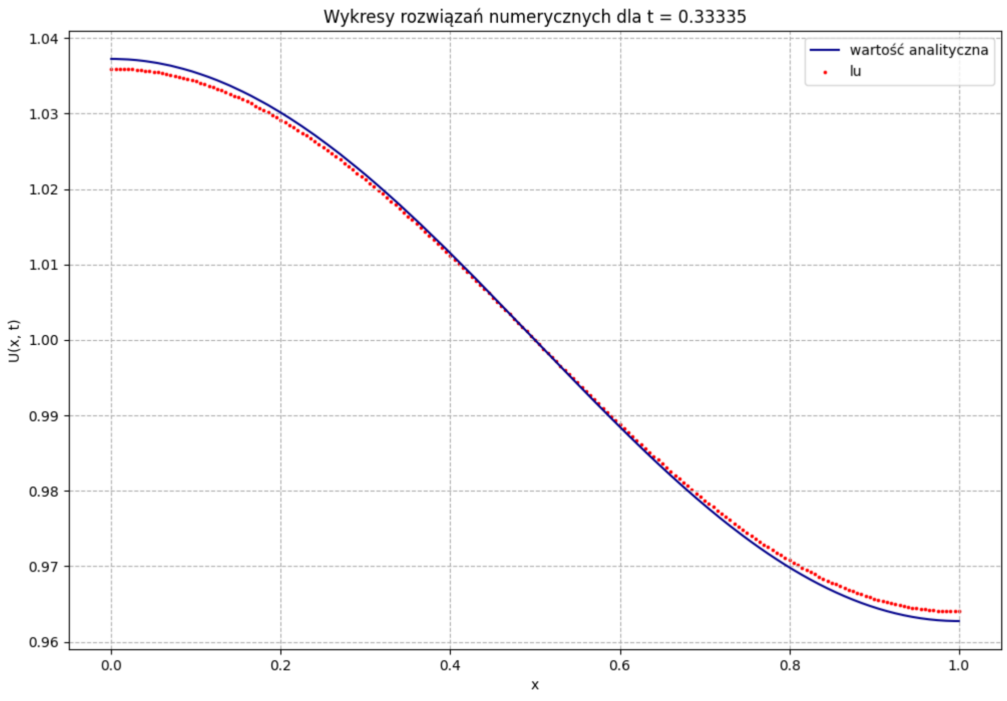


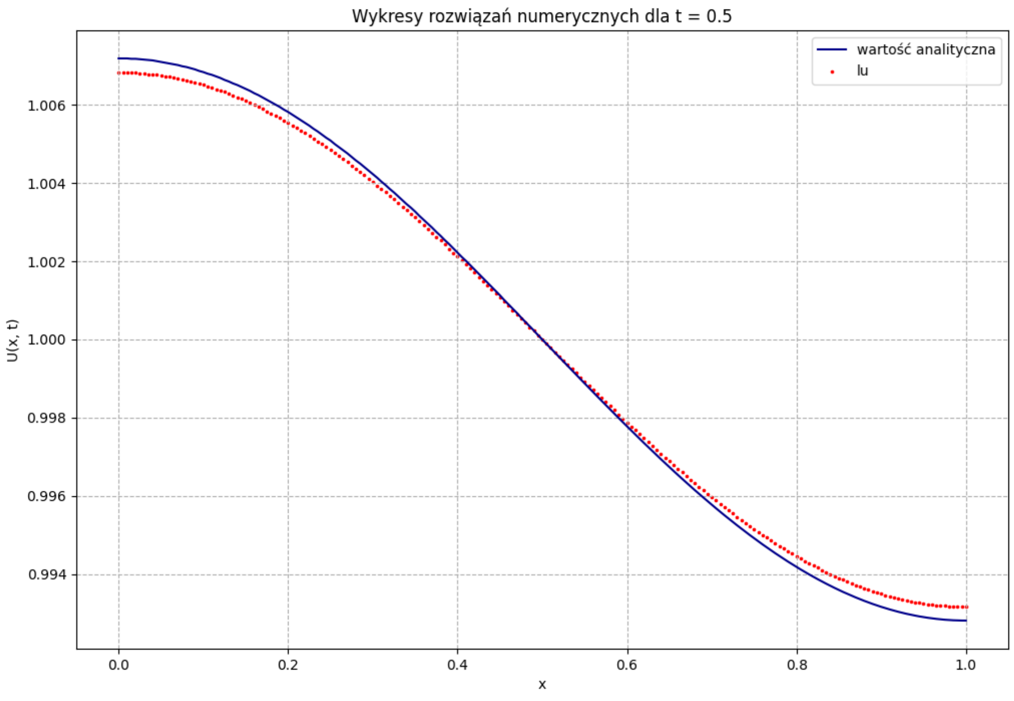



Wykresy rozwiązań numerycznych dla t = 0.49999
wartość analityczna
kmb
U(x, t)
x
1.006
1.004
1.002
1.000
0.998
0.996
0.994
0.0
0.2
0.4
0.6
0.8
1.0

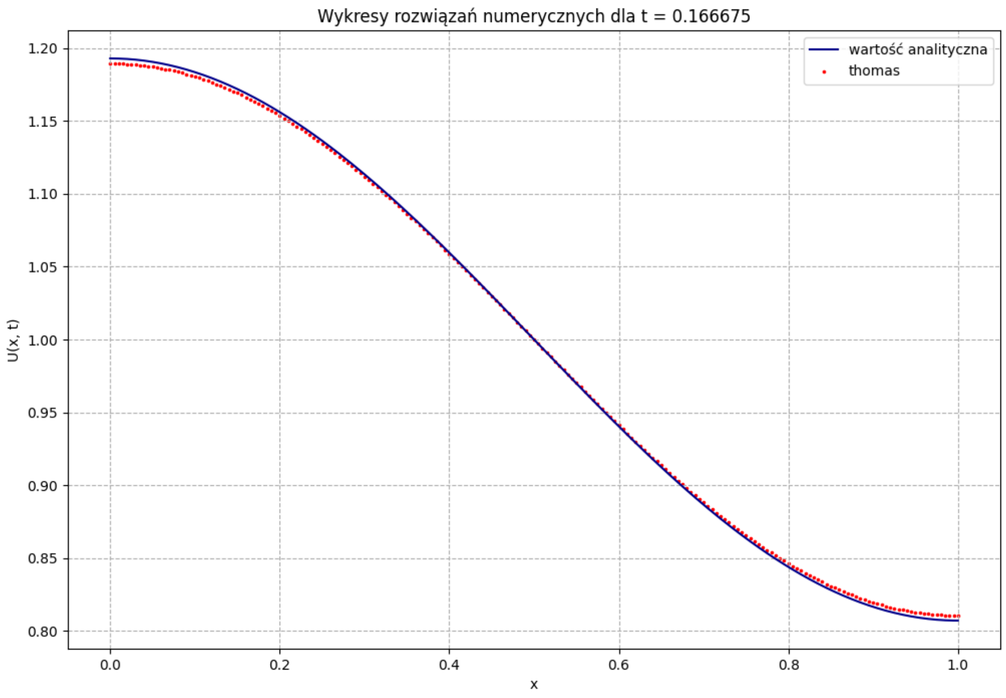
Laasonen LU

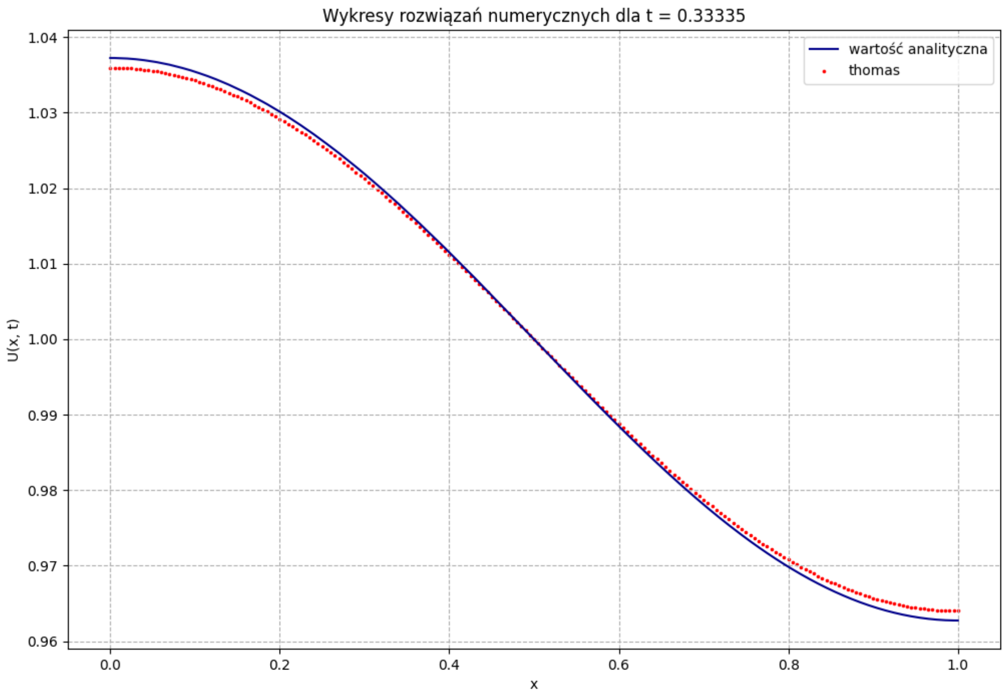

Wykresy rozwiązań numerycznych dla t = 0.166675
wartość analityczna
lu
U(x, t)
x
1.20
1.15
1.10
1.05
1.00
0.95
0.90
0.85
0.80
0.0
0.2
0.4
0.6
0.8
1.0

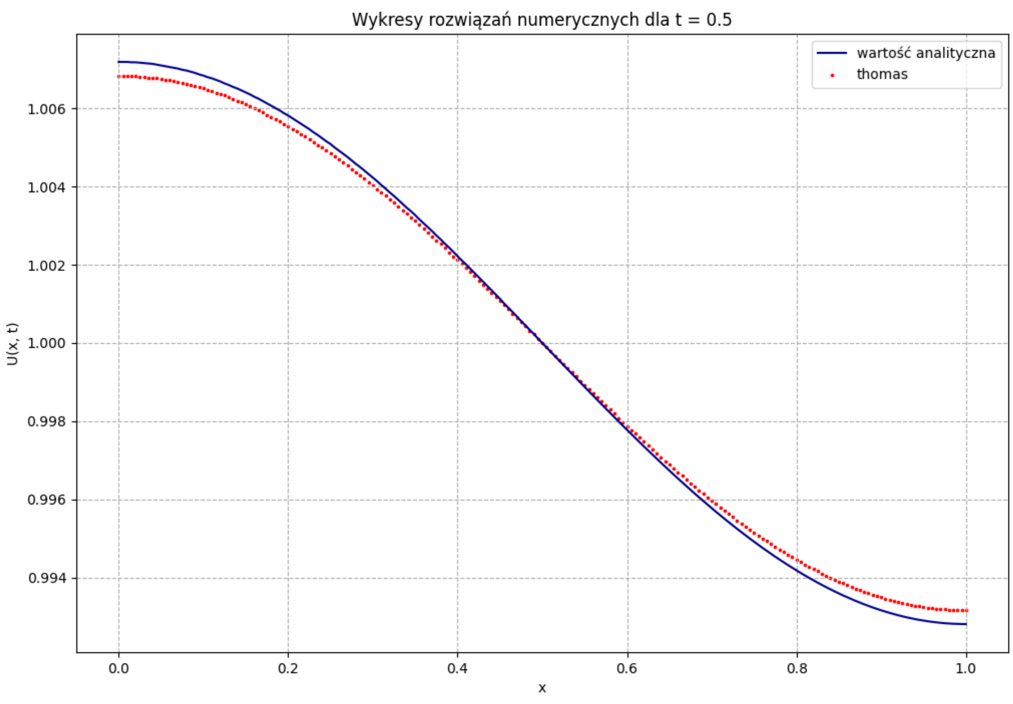

Wykresy rozwiązań numerycznych dla t = 0.33335
wartość analityczna
lu
U(x, t)
x
1.04
1.03
1.02
1.01
1.00
0.99
0.98
0.97
0.96
0.0
0.2
0.4
0.6
0.8
1.0


Wykresy rozwiązań numerycznych dla t = 0.5
wartość analityczna
lu
U(x, t)
x
1.006
1.004
1.002
1.000
0.998
0.996
0.994
0.0
0.2
0.4
0.6
0.8
1.0

Laasonen Thomas



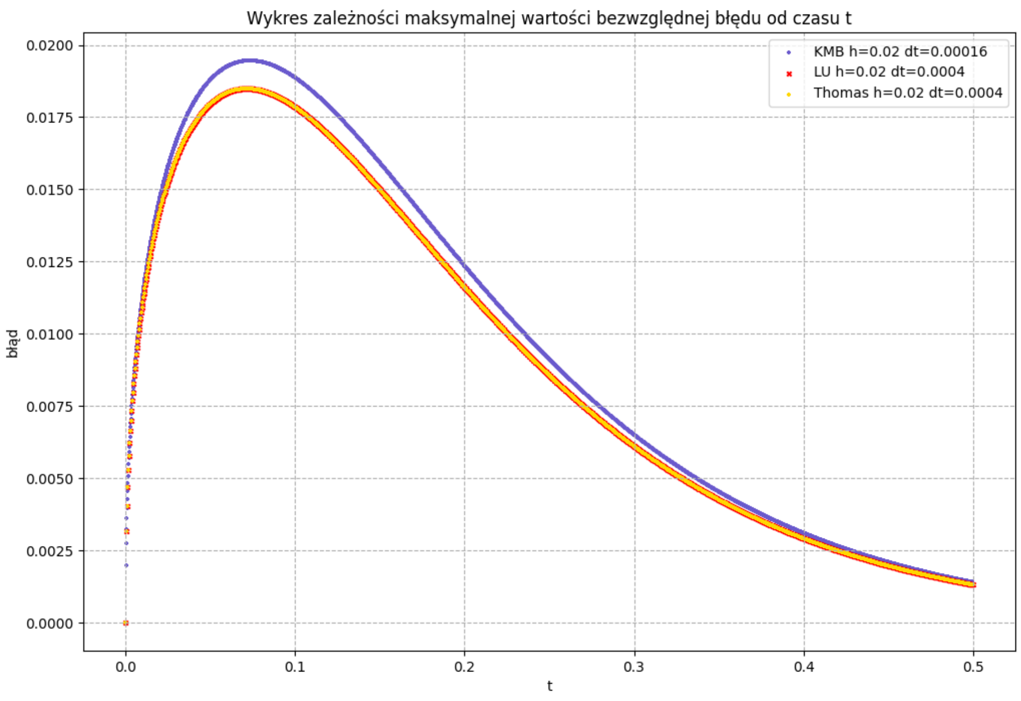


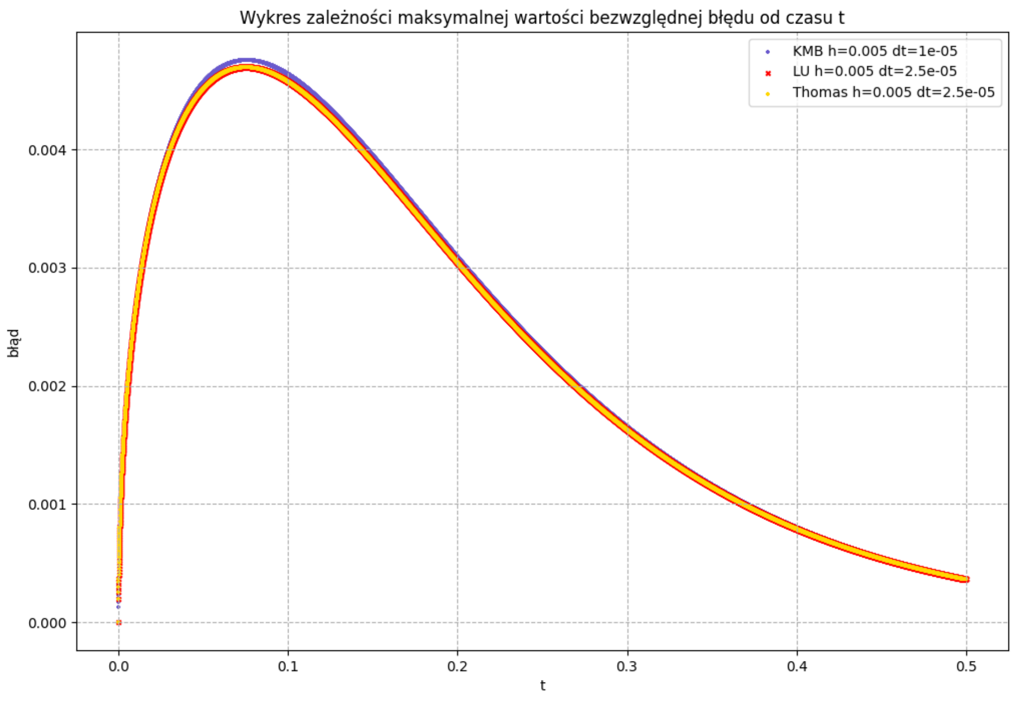


Wszystkie metody dają w miarę dobre wyniki, aczkolwiek możemy zaobserwować odchylenia na końcach i początkach wykresów – może to być spowodowane tym, że z warunków brzegowych nie dostajemy dokładne wartości, tylko zbliżone na podstawie przybliżeń dwupunktowych.

Zadanie 3

Zrobiliśmy wykresy dla róznych parametrów *h* i *dt.*





Zauważmy, że zmiejszenie kroków siatki powoduje duże zwiększenie dokładności obliczeń, aczkolwiek odpowiednio zwiększają się koszty obliczeniowe.

Metoda Laasonen daje bardzo podobne wyniki jak przy użyciu dekompozycji LU, tak i algorytmu Thomasa. Natomiast klasyczna metoda bezpośrednia daje trochę gorsze.

Zwiększenie poziomu płedów na początku wykresu i ich późniejszy spadek może być spowodowane osobliwością rozwiązania analitycznego, do wartości którego dążymy – wraz z wzrostem *t* wartości funkcji dążą do 1 i dlatego dla każdej kolejnej iteracji otrzymujemy coraz bliższe wyniki do poprzedniej.

Kod programu

```
#include <iostream>
#include <fstream>
#include <cmath>
#include <old/vector.h>
#include <old/matrix.h>

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 save_values(Matrix &U, double dt, double h, int i, string filename) {
    double t = i*dt;

    ofstream file;
    file.open("../lab11/" + filename);
    file << "x,calculated,analytic,t=" << t << endl;

    for (int j = 0; j < U.M; j++) {
        double x = j*h;
        file << x << "," << U[i][j] << "," << analytic(x, t) << endl;
    }

    file.close();
}


Matrix getU(int n, int m, double h) {
    Matrix laasonen(n, m);

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

    return laasonen;
}
```

```cpp
Matrix getA(int n, double h, double lambda) {
    Matrix A(n, 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;
}

Matrix getLDU(int n, double h, double lambda) {
    Matrix LDU(n, 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;
}


Matrix 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;

    Matrix 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(Matrix &LDU) {
    for (int i = 1; i < LDU.N; i++)
        LDU[i][1] -= LDU[i][0] * LDU[i-1][2] / LDU[i-1][1];
}

Vector thomas_solve(Matrix &LDU, Vector &b) {
    int N = b.N;
    for (int i = 1; i < N; i++)
        b[i] -= (LDU[i][0] * b[i-1]) / LDU[i-1][1];

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


Matrix 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;

    Matrix U = getU(N, M, h);

    Matrix LDU = getLDU(M, h, lambda);
    thomas_transform(LDU);

    for (int k = 1; k < N; k++) {
        Vector b(M);

        b[0] = -GAMMA;
        for (int i = 1; i < M-1; i++)
            b[i] = -U[k-1][i];
        b[M-1] = -THETA;

        Vector u_k_next = thomas_solve(LDU, b);
        for (int i = 0; i < M; i++)
            U[k][i] = u_k_next[i];
    }

    return U;
}

int getPartialPivot(const Matrix &A, int from) {
    int pivot = from;
    double max = A[from][from];

    for (int i = from; i < A.N; i++) {
        double value = abs(A[i][from]);
        if (value > max) {
            max = value;
            pivot = i;
        }
    }
```

```cpp
    return pivot;
}

void rearrangeIndexes(Vector &vect, const Vector &indexes) {
    Vector tmp = vect;
    for (int i = 0; i < vect.N; i++) {
        double val = tmp[i];
        int index = indexes[i];
        vect[index] = val;
    }
};

void LU_decomposition(const Matrix &A, Vector &indexes, Matrix &L, Matrix &U) {
    U = A;

    for (int k = 0; k < A.N-1; k++) {
        double divisor = U[k][k];
        if (divisor <= 1.0e-6 && divisor >= -1.0e-6) {
            int pivot = getPartialPivot(U, k);
            U.swapRows(k, pivot);
            L.swapRows(k, pivot);
            indexes.swapElements(k, pivot);
            divisor = U[k][k];

            cout << "Zamiana kolumn " << k << " oraz " << pivot << ": " << endl;
            U.print();
        }

        L[k][k] = 1.0;

        Vector firstRow = U[k];
        for (int row = k+1; row < A.N; row++) {
            double firstInCol = U[row][k];
            double multiplier = firstInCol / divisor;

            L[row][k] = multiplier;

            for (int col = k; col < A.N; col++)
                U[row][col] -= firstRow[col] * multiplier;
        }
    }

    L[A.N-1][A.N-1] = 1.0;
}

Vector LU_y(Matrix &L, Vector &b) {
    Vector y(b.N);

    for (int n = 0; n < b.N; n++) {
        double sum = 0.0;
        for (int m = 0; m <= n-1; m++)
            sum += L[n][m] * y[m];
        y[n] = b[n] - sum;
    }

    return y;
}
```

```
Vector LU_x(Matrix &U, Vector &y) {
    Vector x(y.N);

    int sumElementsCount = 0.0;
    for (int n = y.N-1; n >= 0.0; n--) {
        double sum = 0.0;
        int m = y.N - 1;
        for (int _ = 0; _ < sumElementsCount; _++) {
            sum += U[n][m] * x[m];
            m--;
        }
        sumElementsCount++;
        x[n] = (y[n] - sum) / U[n][n];
    }

    return x;
}

Matrix ML_LU(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;

    Matrix U = getU(N, M, h);
    Matrix A = getA(M, h, lambda);

    Vector indices(M);
    for (int index = 0; index < M; index++)
        indices[index] = index;

    Matrix A_L(M);
    Matrix A_U(M);
    LU_decomposition(A, indices, A_L, A_U);

    for (int k = 1; k < N; k++) {
        Vector b(M);

        b[0] = -GAMMA;
        for (int i = 1; i < M-1; i++)
            b[i] = -U[k-1][i];
        b[M-1] = -THETA;

        Vector y = LU_y(A_L, b);
        rearrangeIndexes(b, indices);

        Vector u_k_next = LU_x(A_U, y);

        for (int i = 0; i < M; i++)
            U[k][i] = u_k_next[i];
    }

    return U;
}
```

```cpp
void errors_t(double dt, double h, Matrix &U, string filename) {
    ofstream thomas_error_t;
    thomas_error_t.open(filename);
    thomas_error_t << "t,max_error,h=" << h << ",dt=" << dt << endl;

    for (int i = 0; i < U.N; i++) {
        double t = dt*i;

        double max_error = 0.0;
        for (int j = 0; j < U.M; j++) {
            double x = h*j;

            double error = abs(U[i][j] - analytic(x, t));
            if (max_error < error)
                max_error = error;
        }

        thomas_error_t << t << "," << max_error << endl;
    }
}

int main() {
    // zadanie 3
    double dt = 0.0004, dt_kmb = 0.00016;
    double h = 0.02, h_kmb = 0.02;

    Matrix kmb_smaller_step = KMB(dt_kmb, h_kmb);
    Matrix thomas_smaller_step = ML_Thomas(dt, h);
    Matrix lu_smaller_step = ML_LU(dt, h);

    errors_t(dt_kmb, h_kmb, kmb_smaller_step, "../lab11/kmb_error_t_small.csv");
    errors_t(dt, h, thomas_smaller_step, "../lab11/thomas_error_t_small.csv");
    errors_t(dt, h, lu_smaller_step, "../lab11/LU_error_t_small.csv");


    dt = 0.000025, dt_kmb = 0.00001;
    h = 0.005, h_kmb = 0.005;

    Matrix kmb = KMB(dt_kmb, h_kmb);
    Matrix thomas = ML_Thomas(dt, h);
    Matrix lu = ML_LU(dt, h);

    errors_t(dt_kmb, h_kmb, kmb, "../lab11/kmb_error_t.csv");
    errors_t(dt, h, thomas, "../lab11/thomas_error_t.csv");
    errors_t(dt, h, lu, "../lab11/LU_error_t.csv");


    // zadanie 2
    save_values(kmb, dt_kmb, h_kmb, kmb.N / 3, "kmb_1_t.csv");
    save_values(kmb, dt_kmb, h_kmb, 2 * kmb.N / 3, "kmb_2_t.csv");
    save_values(kmb, dt_kmb, h_kmb, kmb.N-1, "kmb_3_t.csv");

    save_values(thomas, dt, h, thomas.N / 3, "thomas_1_t.csv");
    save_values(thomas, dt, h, 2 * thomas.N / 3, "thomas_2_t.csv");
    save_values(thomas, dt, h, thomas.N-1, "thomas_3_t.csv");

    save_values(lu, dt, h, lu.N / 3, "lu_1_t.csv");
    save_values(lu, dt, h, 2 * lu.N / 3, "lu_2_t.csv");
    save_values(lu, dt, h, lu.N-1, "lu_3_t.csv");



    // zadanie 1
    ofstream errors_h;
    errors_h.open("../lab11/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);
        lu = ML_LU(dt, h);
        thomas = ML_Thomas(dt, h);

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

        double max_error_lu = 0.0;
        for (int i = 0; i < lu.M; i++) {
            double error = abs(lu[lu.N - 1][i] - analytic(i * h, T_MAX));
            if (max_error_lu < error)
                max_error_lu = error;
        }

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

        cout << h << "," << max_error_kmb << "," << max_error_lu << "," <<
max_error_thomas << endl;
        errors_h << h << "," << max_error_kmb << "," << max_error_lu << "," <<
max_error_thomas << endl;

        h /= 2.0;
    }
    errors_h.close();
}
```

Vector.h

```
#include <vector>
#include <iostream>

#ifndef LAB11_VECTOR_H
#define LAB11_VECTOR_H

class Vector {
    std::vector<double> data;

public:
    int N;

    Vector(const Vector &vect) : N(vect.N), data(vect.data) {};
    Vector(std::initializer_list<double> data) : N(data.size()),
data(std::vector<double>(data)) {};
    Vector(const std::vector<double> &data) : N(data.size()), data(data) {};
    Vector(int size) : N(size), data(std::vector<double>(size, 0)) {};
    Vector(int size, double value) : N(size), data(std::vector<double>(size,
value)) {};

    Vector& operator=(const Vector &vect);
    double& operator[](int i);
    double operator[](int i) const;

    void swapElements(int a, int b) { std::swap(data[a], data[b]); };
    void print(int width = 10);
};


#endif //LAB11_VECTOR_H
```

Vector.cpp

```cpp
#include "vector.h"
#include <iomanip>


Vector& Vector::operator=(const Vector &vect) {
    N = vect.N;
    data = vect.data;
    return *this;
};

double &Vector::operator[](int i) {
    return data[i];
}

double Vector::operator[](int i) const {
    return data[i];
}

void Vector::print(int width) {
    for (auto value : data) {
        std::cout << std::setw(width) << value << " ";
    }
    std::cout << std::endl;
}
```

Matrix.h

```
#include <vector>
#include "vector.h"

#ifndef LAB11_MATRIX_H
#define LAB11_MATRIX_H

class Matrix {
    std::vector<Vector> data;

public:
    int N, M;
    Matrix(const Matrix &matrix) : N(matrix.N), M(matrix.M), data(matrix.data) {};
    Matrix(int N, int M) : N(N), M(M), data(std::vector<Vector>(N, Vector(M))) {};
    Matrix(int N) : N(N), M(N), data(std::vector<Vector>(N, Vector(N))) {};
    Matrix(std::initializer_list<Vector> matrix) : N(matrix.size()),
M((*matrix.begin()).getN()),

data(std::vector<Vector>(matrix)) {};

    Matrix(int N, int M, double value) : N(N), M(M), data(std::vector<Vector>(N,
Vector(M, value))) {};

    Vector& operator[](int i) { return data[i]; };
    Vector operator[](int i) const { return data[i]; };
    std::vector<Vector>::iterator begin() { return data.begin(); }
    std::vector<Vector>::iterator end() { return data.end(); }

    void swapRows(int a, int b) { std::swap(data[a], data[b]); };

    void print(int width = 8) const;
};

#endif //LAB11_MATRIX_H
```

Matrix.cpp

```
#include "matrix.h"

void Matrix::print(int width) const {
    for (Vector row : data)
        row.print(width);
    std::cout << std::endl;
}
```