Metody obliczeniowe

Ćwiczenie 11-7

Władisław Czebotarew

1) ***Wstęp:***

a. **Temat**: Rozwiązywanie równania różniczkowego cząstkowego metodami numerycznymi.

b. **Równanie różniczkowe cząstkowe:**

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

Określone dla współrzędnej przestrzennej $x \in (-\infty, +\infty)$ oraz czasu $t \in [0, t_{max})$.

- Warunek początkowy: $U(x,0) = \begin{cases} 1 \; dla \; |x| < b \\ 0 \; dla \; |x| \geq b \end{cases}$
- Warunki brzegowe: $U(-\infty, t) = 0, \quad U(+\infty, t) = 0$

Zagadnienie to może opisywać transport ciepła, w ośrodku nieskończonym o współczynniku transportu ciepła D, po raptownym zetknięciu trzech części ośrodka o różnej temperaturze (ogrzanej warstwy o grubości 2b, oraz zimnych zewnętrznych pół-nieskończonych obszarów), w chwili t = 0.

- Rozwiązanie analityczne:

$$U(x,t) = \frac{1}{2} erf\left(\frac{x+b}{2\sqrt{Dt}}\right) - \frac{1}{2} erf\left(\frac{x-b}{2\sqrt{Dt}}\right)$$

Gdzie: $erf(z)$ jest tzw. funkcją błędu: $erf(z) = \frac{2}{\sqrt{\pi}} \int_0^z exp(-w^2) dw$

c. **Założenia:**

Do obliczeń numerycznych:

- Przedział nieskończony x należy zastąpić przedziałem skończonym

$$x \in [-a, a], \quad \text{gdzie } a \geq b + 6\sqrt{D t_{max}}$$

Do obliczenia funkcji erfc(z) z dokładnością bliską dokładności maszynowej dla zmiennych typu **double** należy zastosować pakiet **CALERF** udostępniony przez prowadzącego zajęcia.

d. **Parametry:**

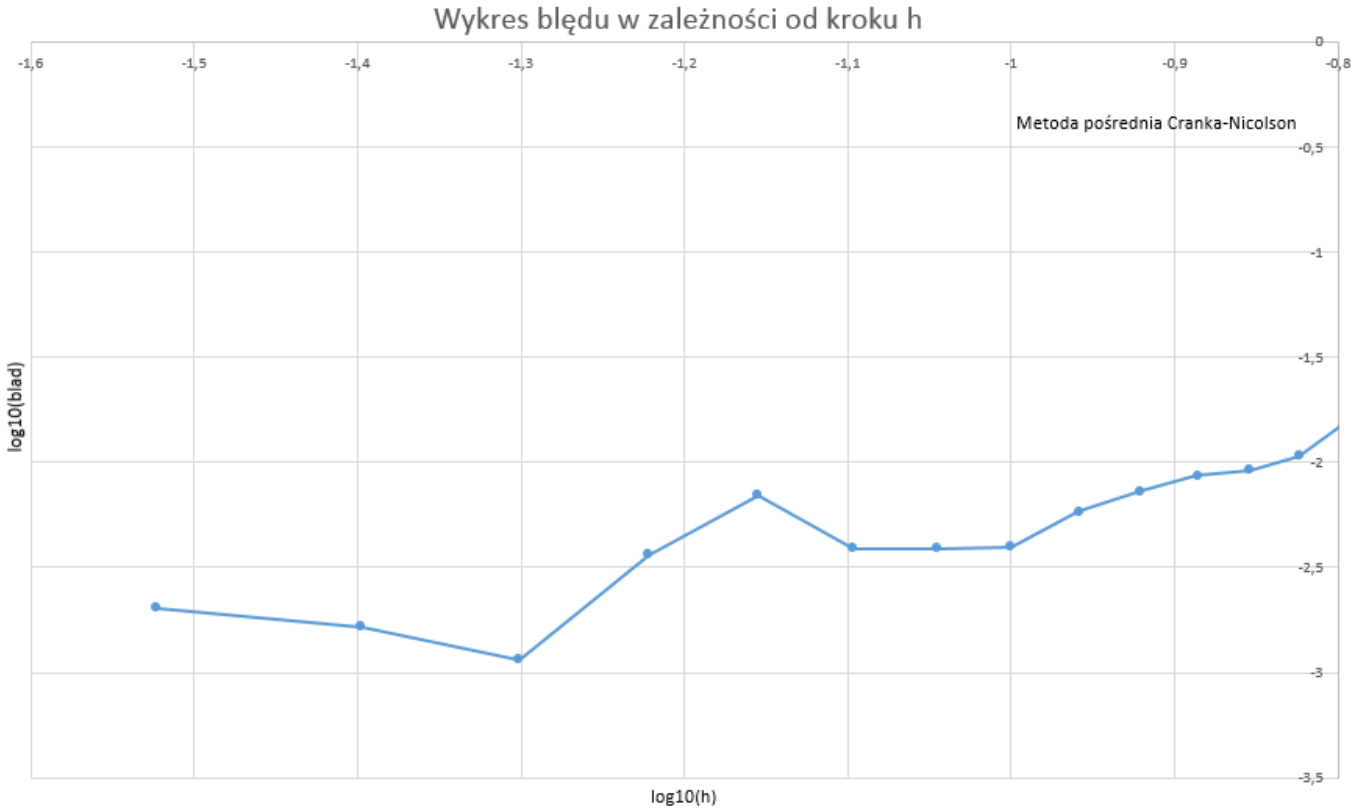
- $t_{max} = 2$
- $b = 1$
- $D = 1$

e. **Algorytmy:**

- Dyskretyzacja:
  - Metoda pośrednia Cranka-Nicolson
- Rozwiązania algebraiczne układów równań licznikowych:
  - Algorytm Thomasa
  - Metoda iteracyjna Jacobiego

2) ***Wykresy***:

a. *Wykres maksymalnej wartości bezwzględnej błędu obserwowanej dla* t_{max}, *w funkcji kroku przestrzennego h:*



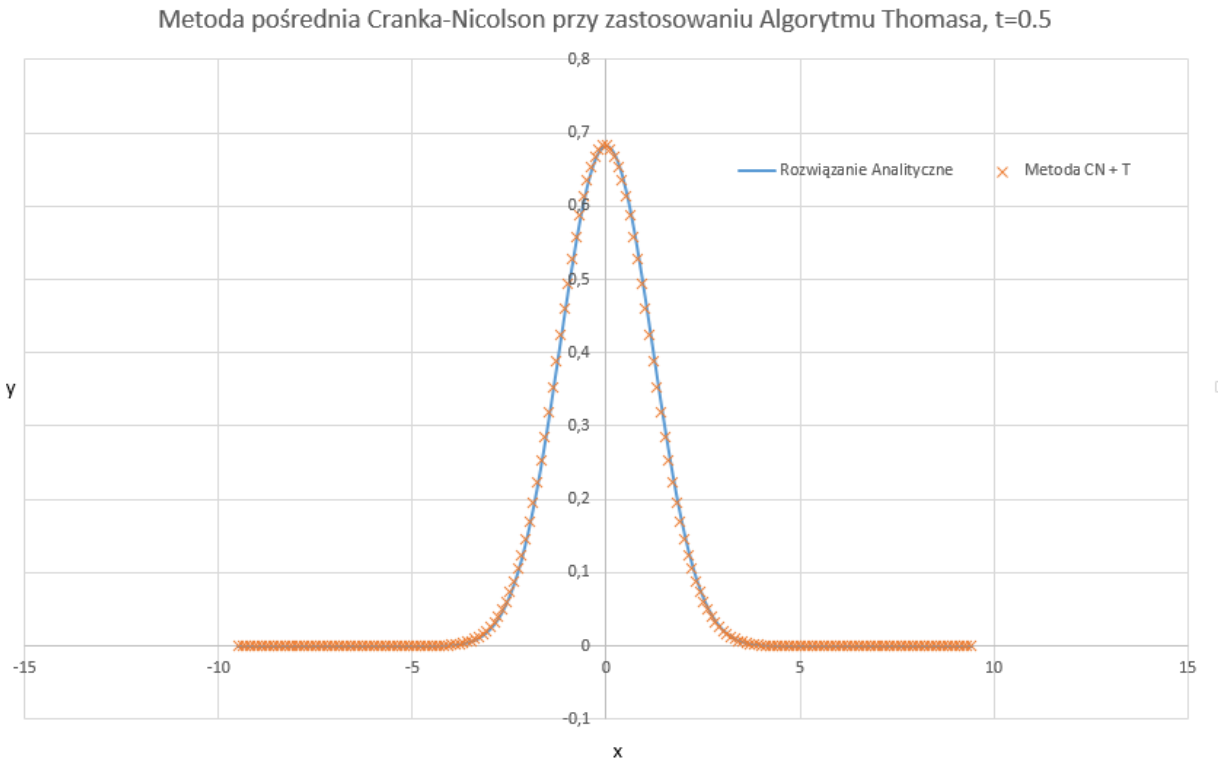
Wykres przedstawia zależność błędu od kroku (zakres od 0.5 do 0.01). Na jego podstawie można zauważyć, że rzędy dokładności potwierdzają przypuszczenia teoretyczne.

	Dane (dla h = 0.1)	**Teoretyczny rząd dokładności**
Metoda pośrednia Cranka-Nicolson	$\frac{(-2,137-(-1,805))}{(-0,95-(-0,785))} = 2,474$	**2**

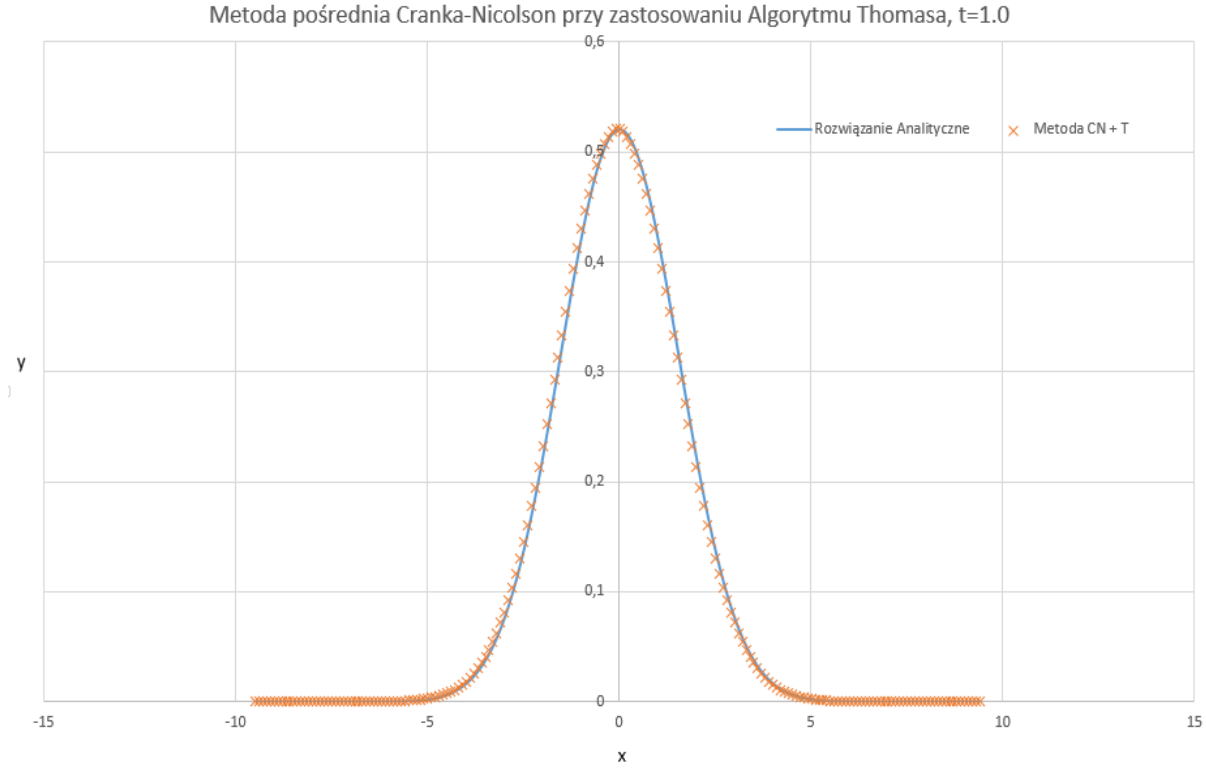
b. *Wykres rozwiązań numerycznych i analitycznych dla kilku wybranych wartości czasu t:*

- Metoda pośrednia Cranka-Nicolson przy zastosowaniu Algorytmu Thomasa :

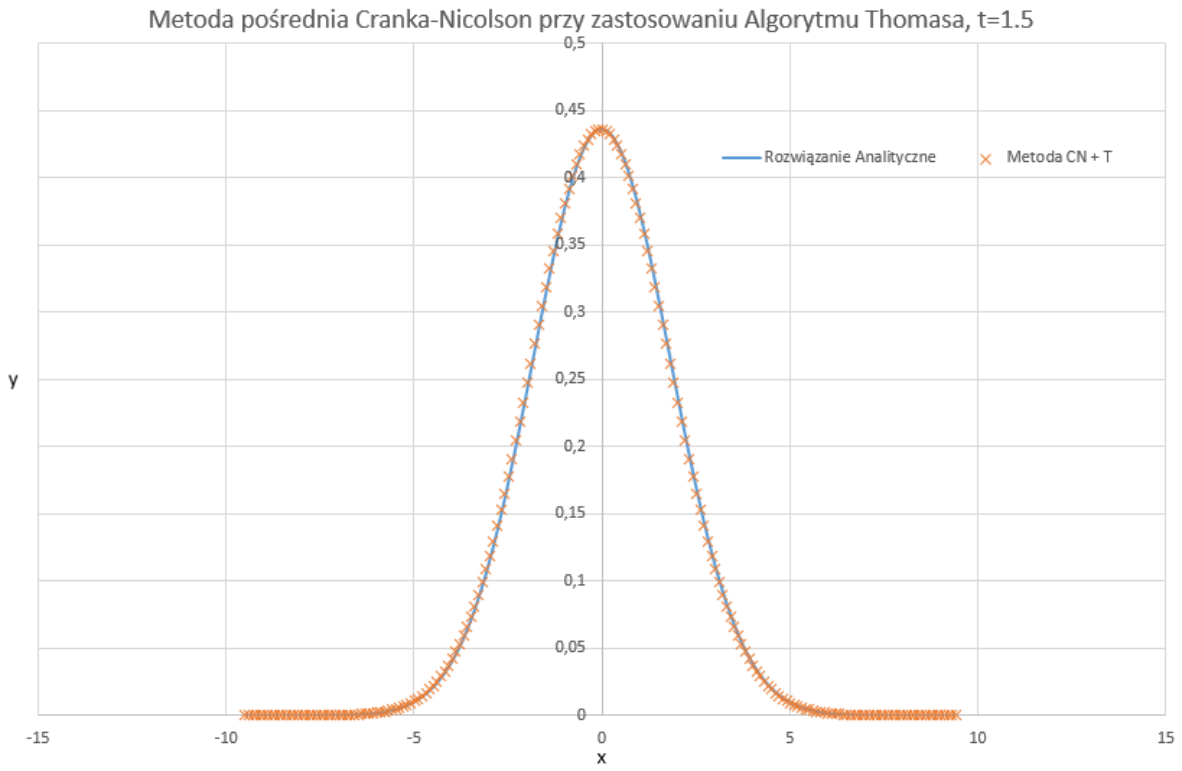
  - $t = 0.5$



  - $t = 1.0$

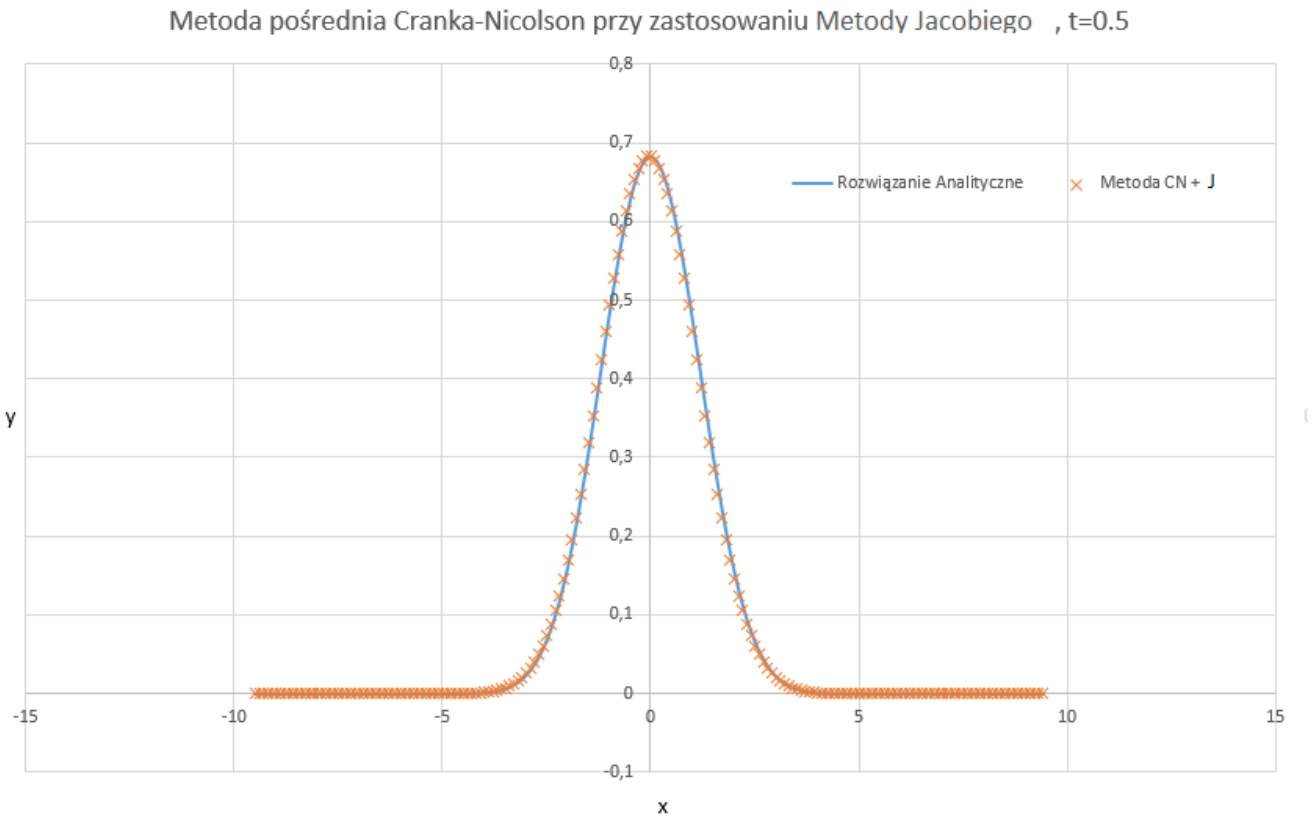


- $t = 1.5$

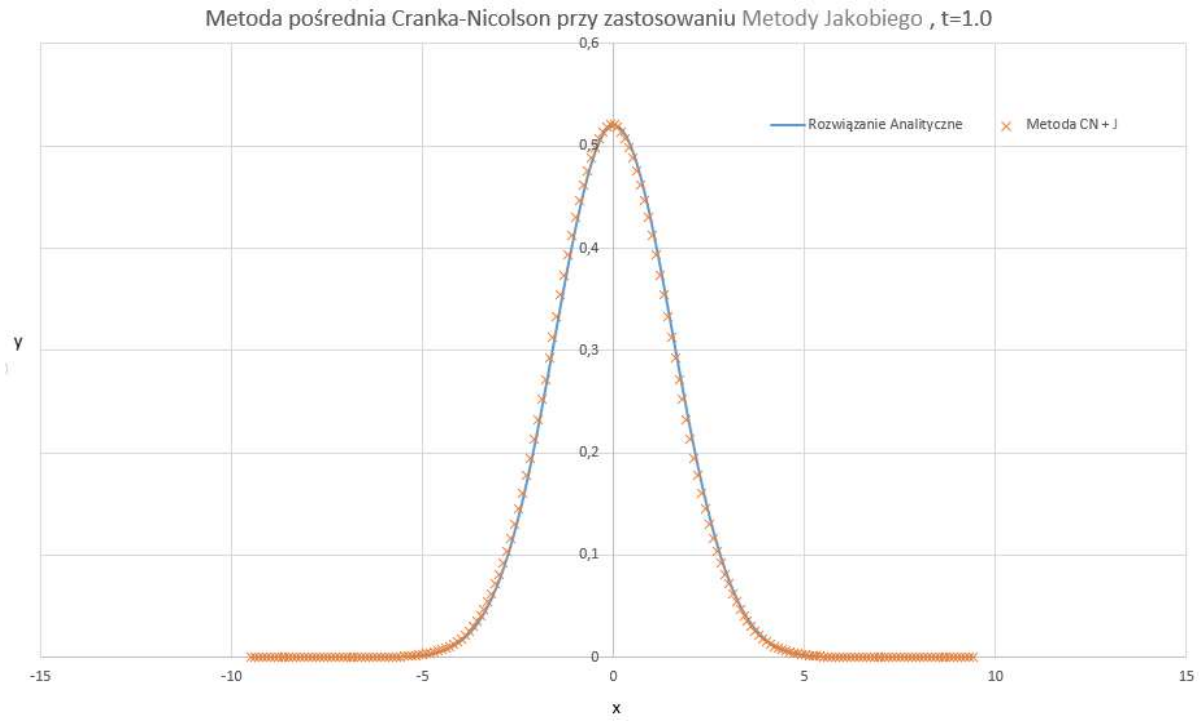


- <u>Metoda pośrednia Cranka-Nicolson przy zastosowaniu Metody Jacobiego:</u>

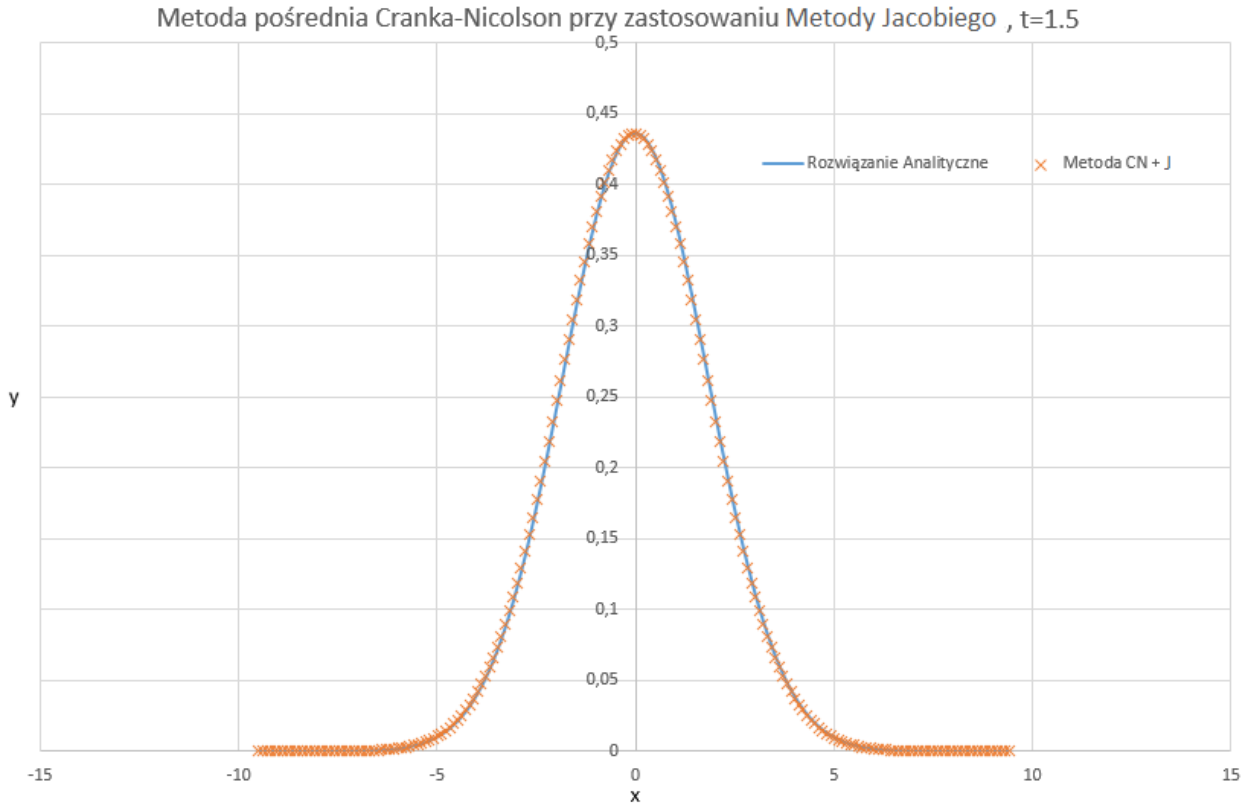
- $t = 0.5$



- $t = 1.0$



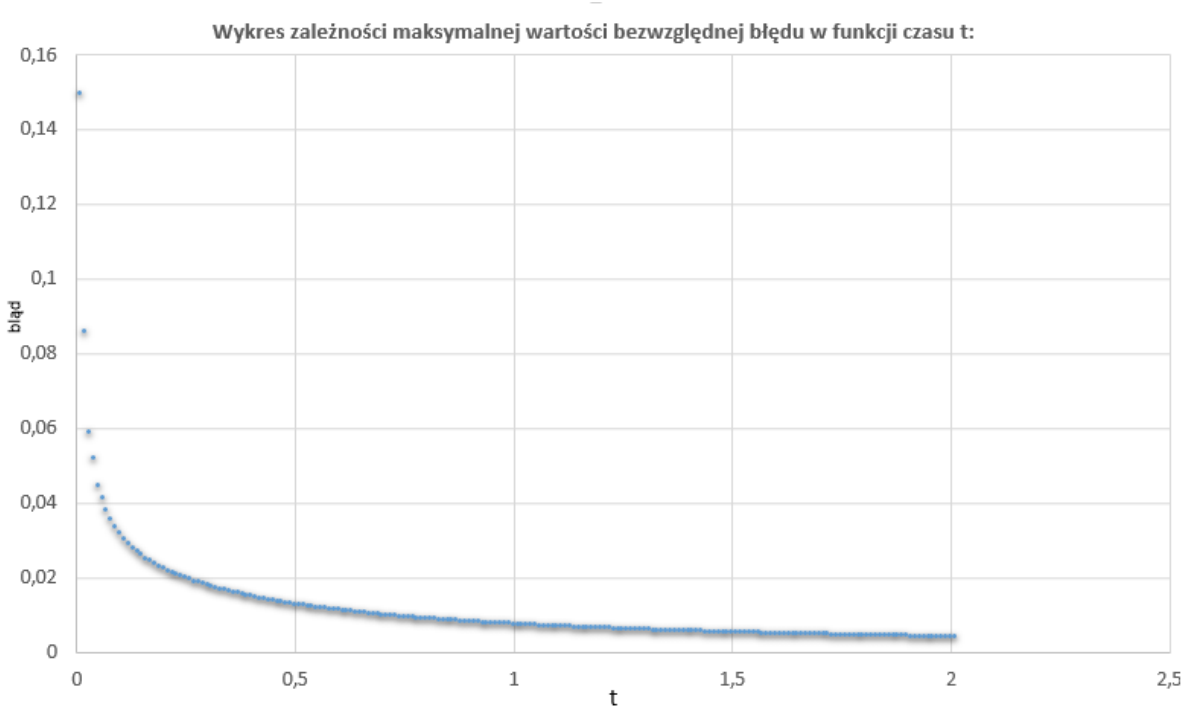
- $t = 1.5$



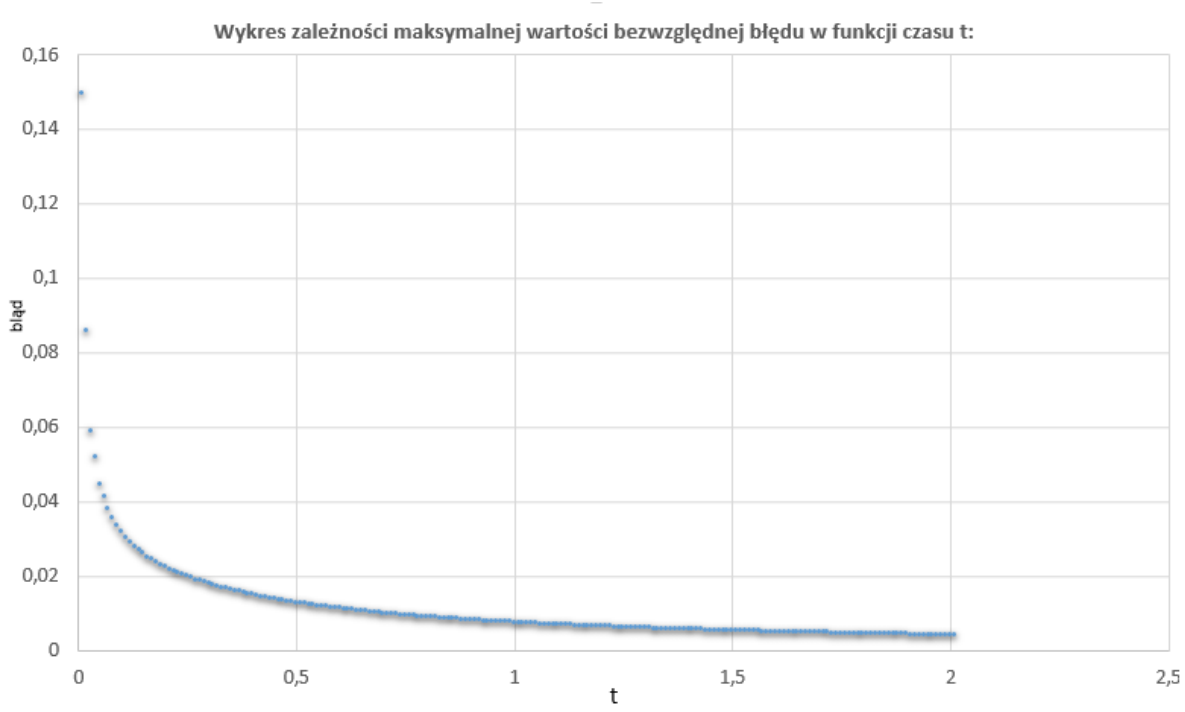
Dla wszystkich wykresów, rozwiązania analityczne pokrywają się z rozwiązaniami numerycznymi.

c. *Wykres zależności maksymalnej wartości bezwzględnej błędu w funkcji czasu t:*

- Metoda pośrednia Cranka-Nicolson przy zastosowaniu Algorytmu Thomasa:



- Metoda pośrednia Cranka-Nicolson przy zastosowaniu Metody Jacobiego:



Wartość błędu wyraźnie spada raz z obliczeniami kolejnych poziomów t. Najbardziej zauważalny jest ona dla małych wartości t, następnie błąd maleje liniowo. W przypadku gdy błąd rósłby wraz ze wzrostem t, świadczyło by o błędnym wyprowadzaniu i wykorzystaniu wzorów. Metoda pośrednia Cranka-Nicolson wykazuje nieco większa zbieżności dla początkowych poziomów czasowych.

3) ***Wnioski Końcowe:***

Obliczania numeryczne wartości równania różniczkowego wykorzystujące dyskretyzację metodą pośrednią Cranka-Nicolson oraz Algorytm Thomasa i metodę iteracyjną Jacobiego do rozwiązania układu równań są dokładne. Nie oznacza to jednak, ze nie są one obarczone błędami. Najważniejsza częścią błędu jest błąd maszynowy oraz błąd przybliżeń. Nie jesteśmy wstanie całkowicie ich wyeliminować. Dla powyższych metod (dla kroku h = 0.1 i t > 1) błąd jest rzędu $\mathbf{10^{-2}}$ co jest zadowalającą dokładnością – przy niewielkim czasie obliczeń.

4) ***Program:***

```
#include "stdafx.h"
#include <iostream>
#include <cmath>
#include <fstream>
#include "calerf.h"

using namespace std;

// STALE
const double D = 1.0;
const double b = 1.0;
const double tmax = 2.0;
const double a = b + 6.0 * sqrt(D * tmax);
const double lambda = 1.0;
const double eps = 1e-7;

//pliki do zapisania danych do wykresu
fstream file_analytical, file_error, file_tmax_error;

// FUNKCJE
double *newVector(int n)   //funkcja tworzaca wektor
{
	return new double[n];
}


double **newMatrix(int n, int m)	//funkcja tworzaca macierz
{
	double** matrix = new double*[n];

	for (int i = 0; i < n; i++)
	{
		matrix[i] = new double[m];
	}

	return matrix;
}

void deleteVector(double* x)    //funkcja usuwajaca wektor
{
	delete[] x;
}

void deleteMatrix(double** x, int n)    //funkcja usuwajaca macierz
{
	for (int i = 0; i < n; i++)
	{
		delete[] x[i];
```

```
	}
	delete[] x;
}

void fill(double* dtlvl, double* hlvl, double deltaDT, int hN, double deltaH, int hM )
//wypelnienie wektorow x i t
{
	dtlvl[0] = 0.0;
	hlvl[0] = -a;

	for (int i = 1; i<hN; i++)
		dtlvl[i] = dtlvl[i - 1] + deltaDT;

	for (int i = 1; i < hM; i++)
		hlvl[i] = hlvl[i - 1] + deltaH;

}


void analytical(double** analyt, int N, int M, double* dtLevels, double* hLevels, double dt, double
h)  //rozwiązanie analityczne
{

	for (int i = 0; i < M; i++){
		if (fabs(hLevels[i])<b)
			analyt[0][i] = 1.0; //warunek poczatkowy U(X, 0) = 1  dla  |x|<b
		else
			analyt[0][i] = 0.0;  //warunek poczatkowy U(X, 0) = 0  dla  |x|>=b
	}

	for (int i = 0; i < N; i++)  //warunek brzegowy U(-INF, t) = 0
		analyt[i][0] = 0.0;

	for (int i = 0; i < N; i++) //warunek brzegowy U(INF, t) = 0
		analyt[i][M - 1] = 0.0;



	for (int i = 1; i < N; i++)
	{
		for (int j = 1; j < M - 1; j++)
		{

			analyt[i][j] = 0.5*erf((hLevels[j] + b) / (2 * sqrt(D * dtLevels[i]))) -
0.5*erf((hLevels[j] - b) / (2 * sqrt(D * dtLevels[i])));
		}
	}

}



void thomasAlgorithm(double *l, double *d, double *u, int n) {
	for (int i = 1; i<n; i++) {
		d[i] = d[i] - l[i - 1] * (u[i - 1] / d[i - 1]);
	}
}

void solveEquation(double *l, double *d, double *u, double *b, int n) {
	for (int i = 1; i<n; i++) {
		b[i] = b[i] - l[i - 1] * b[i - 1] / d[i - 1];
	}

	b[n - 1] = b[n - 1] / d[n - 1];

	for (int i = n - 2; i >= 0; i--) {
		b[i] = (b[i] - u[i] * b[i + 1]) / d[i];
	}
}
```

```
void CrankNicolsonT(double** Matrix, int N, int M, double* dtLevels, double* hLevels, double dt,
double h, double lambda)
{
	double* U = newVector(M);
	double* D = newVector(M);
	double* L = newVector(M);
	double* bb = newVector(M);
	double* x = newVector(M);

	for (int i = 0; i < M; i++){
		if (fabs(hLevels[i]) < b)
			Matrix[0][i] = 1.0; //warunek poczatkowy U(X, 0) = 1  dla  |x|<b
		else
			Matrix[0][i] = 0.0;  //warunek poczatkowy U(X, 0) = 0  dla  |x|>=b
	}

	for (int i = 0; i < N; i++)  //warunek brzegowy U(-INF, t) = 0
		Matrix[i][0] = 0.0;

	for (int i = 0; i < N; i++) //warunek brzegowy U(INF, t) = 0
		Matrix[i][M - 1] = 0.0;

	for (int k = 1; k < N; k++)
	{
		L[0] = 0.0;
		D[0] = 1.0;
		U[0] = 0.0;
		bb[0] = 0.0;

		for (int i = 1; i < M - 1; i++)
		{
			L[i] = -(lambda / 2.0);
			D[i] = 1 + lambda;
			U[i] = -(lambda / 2.0);
			bb[i] = (lambda / 2.0 * Matrix[k - 1][i - 1] + (1.0 - lambda) * Matrix[k -
1][i] + (lambda / 2.0) * Matrix[k - 1][i + 1]);
		}

		L[M - 1] = 0.0;
		D[M - 1] = 1.0;
		U[M - 2] = 0.0;
		bb[M - 1] = 0.0;


		thomasAlgorithm(L, D, U, M);

		solveEquation(L, D, U, bb, M);


		for (int i = 1; i < M - 1; i++)
		{
			Matrix[k][i] = bb[i];
		}
	}

	deleteVector(U);
	deleteVector(D);
	deleteVector(L);
	deleteVector(bb);
	deleteVector(x);

}


double est(double *x, double *x_nowe, int n)
{
	double max = x[0];
```

```
	for (int i = 1; i < n; i++)
	{
		x[i] = fabs(x[i] - x_nowe[i]);
		if (x[i] > max) max = x[i];
	}

	return max;
}

double residuum(double **A, double *b, double *x_nowe, const int m)
{
	double *Ax = new double[m];
	double tmp;

	double max = 0.0;
	for (int i = 0; i < m; i++){
		double suma = 0.0;
		for (int j = 0; j < m; j++){
			tmp = A[i][j] * x_nowe[j];
			suma = suma + tmp;
		}
		Ax[i] = fabs(suma - b[i]);
		if (Ax[i] > max) max = Ax[i];
	}

	return max;
}


void metoda_Jacobiego(double **A, double *b, double *x, int n)
{
	double *x_nowe = new double[n]; //nowe przyblizenia
	double suma = 0.0;
	double EST = 0.0, RESIDUUM = 0.0;


	for (int iter = 0; iter < 100; iter++)
	{
		for (int i = 0; i < n; i++)
		{
			suma = 0.0;
			for (int j = 0; j < n; j++)
			if (j != i)
				suma += A[i][j] * x[j];

			x_nowe[i] = (1.0 / A[i][i]) * (b[i] - suma);

		}

		EST = est(x, x_nowe, n);
		RESIDUUM = residuum(A, b, x_nowe, n);

		for (int i = 0; i < n; i++)
			x[i] = x_nowe[i];


		if (EST < eps && RESIDUUM < eps)
			break;


	}

}


void CrankNicolsonJ(double** Matrix, int N, int M, double* dtLevels, double* hLevels, double dt,
double h, double lambda)
{
```

```
	double* U = newVector(M);
	double* D = newVector(M);
	double* L = newVector(M);
	double* bb = newVector(M);
	double* x = newVector(M);

	int n = M;
	double** A = newMatrix(n, n);

	for (int i = 0; i < n; i++)     //wyzerowanie macirzy A
	{
		for (int j = 0; j < n; j++)
		{
			A[i][j] = 0.0;
		}
	}

	for (int i = 0; i < M; i++){
		if (fabs(hLevels[i]) < b)
			Matrix[0][i] = 1.0; //warunek poczatkowy U(X, 0) = 1  dla  |x|<b
		else
			Matrix[0][i] = 0.0;  //warunek poczatkowy U(X, 0) = 0  dla  |x|>=b
	}

	for (int i = 0; i < N; i++)  //warunek brzegowy U(-INF, t) = 0
		Matrix[i][0] = 0.0;

	for (int i = 0; i < N; i++) //warunek brzegowy U(INF, t) = 0
		Matrix[i][M - 1] = 0.0;

	for (int k = 1; k < N; k++)
	{
		L[0] = 0.0;
		D[0] = 1.0;
		U[0] = 0.0;
		bb[0] = 0.0;

		for (int i = 1; i < M - 1; i++)
		{
			L[i] = -(lambda / 2.0);
			D[i] = 1 + lambda;
			U[i] = -(lambda / 2.0);
			bb[i] = (lambda / 2.0 * Matrix[k - 1][i - 1] + (1.0 - lambda) * Matrix[k -
1][i] + (lambda / 2.0) * Matrix[k - 1][i + 1]);
			x[i] = Matrix[0][i];
		}

		L[M - 2] = 0.0;
		D[M - 1] = 1.0;
		U[M - 2] = 0.0;
		bb[M - 1] = 0.0;
		x[M - 1] = 0.0;



		for (int i = 0; i < n - 1; i++)
		{
			A[i][i] = D[i];
			A[i + 1][i] = L[i];
			A[i][i + 1] = U[i];
		}

		A[n - 1][n - 1] = D[n - 1];

		metoda_Jacobiego(A,bb,x,M);

		for (int i = 1; i < M - 1; i++)
		{
			Matrix[k][i] = x[i];
```

```
			}
		}
		deleteMatrix(A, n);
		deleteVector(U);
		deleteVector(D);
		deleteVector(L);
		deleteVector(bb);
		deleteVector(x);
}


void error(double** ERROR, double** analyt, double** U, int N, int M)
{
		for (int i = 0; i < N; i++)
		{
			for (int j = 0; j < M; j++)
			{
				ERROR[i][j] = fabs(U[i][j] - analyt[i][j]);
			}
		}
}


void save_file(const char* fileName, double** matrix, int r, int c)
{
		ofstream file(fileName);

		file.setf(ios::scientific, ios::floatfield);
		file.precision(4);

		for (int i = 0; i < r; i++)
		{
			for (int j = 0; j < c; j++)
			{
				file << matrix[i][j] << "\t";
			}
			file << endl;
		}

		file.close();
}

void save_file_analytical(double** matrixA, double** matrixCT, double** matrixCJ, double* hlvl, int
M, int index)
{
		file_analytical.open("file_analytical.csv", fstream::out);

		for (int j = 0; j < M; j++)
		{
			file_analytical << hlvl[j] << ";" << matrixA[index][j] << ";" << matrixCT[index][j] <<
";" << matrixCJ[index][j] << ";" << endl;

		}
		file_analytical.close();
}


void save_file_error(const char* fileName, double** analyt, double** matrix, double* tlevel, int N,
int M){

		double error, max_error;

		file_error.open(fileName, fstream::out);
		for (int i = 1; i < N; i++)
		{
			max_error = 0.0;
			for (int j = 0; j < M; j++)
			{
				error = fabs(analyt[i][j] - matrix[i][j]);
```

```
                        if (error > max_error)
                                max_error = error;
                }
                file_error << " " <<tlevel[i] << ";" << max_error << endl;
        }
        file_error.close();
}

//==========


int _tmain(int argc, _TCHAR* argv[])
{
        file_tmax_error.open("file_tmax_error.csv", fstream::out);
        double h = 0.5;

        double errorCT, errorCJ, max_errprCT, max_errorCJ;

        for (h; h > 0.02; h -= 0.01){

                double dt = (lambda * (h*h) / D);
                int N = (int)((tmax / dt) + 2);
                int M = (int)(((2 * a) / h) + 1);

                double* dtLevel = newVector(N);     //wektor wartosci t
                double* hLevel = newVector(M);          //wektor wartosci x

                fill(dtLevel, hLevel, dt, N, h, M);                             //wypelnienie wektora t i
x



                double** AnalytMatrix = newMatrix(N, M);                        //macierz rozwiazan
analitycznych
                analytical(AnalytMatrix, N, M, dtLevel, hLevel, dt, h);                 //rozwiazanie
analityczne


                double** CrankThomasMatrix = newMatrix(N, M);
                CrankNicolsonT(CrankThomasMatrix, N, M, dtLevel, hLevel, dt, h, lambda);

                double** CrankJacobMatrix = newMatrix(N, M);
                CrankNicolsonJ(CrankJacobMatrix, N, M, dtLevel, hLevel, dt, h, lambda);

                //cout << N << endl;


                if (h < 0.101 && h > 0.099){

                        int index = 50;

                        save_file_analytical(AnalytMatrix, CrankThomasMatrix, CrankJacobMatrix, hLevel,
M, index);

                        for (int j = 0; j < M; j++)
                        {
                                file_analytical << hLevel[j] << " " << AnalytMatrix[index][j] << " " <<
CrankThomasMatrix[index][j] << " " << CrankJacobMatrix[index][j] << " " << endl;

                                cout.width(10);
                                cout << hLevel[j] << "|";
                                cout.width(15);
                                cout << AnalytMatrix[index][j] << "|";
                                cout.width(15);
                                cout << CrankThomasMatrix[index][j] << "|";
                                cout.width(15);
                                cout << CrankJacobMatrix[index][j] << "|" << endl;

                        }
```

```
                save_file_error("file_error_CT.csv", AnalytMatrix, CrankThomasMatrix, dtLevel,
N, M);

                save_file_error("file_error_CJ.csv", AnalytMatrix, CrankJacobMatrix, dtLevel,
N, M);

                cout << "end" << endl;

            }


            max_errprCT = 0.0;
            max_errorCJ = 0.0;
            //Maksymalna wartosc bezwzglednego bledu obserwacji dla t_max w funkcji kroku
przestrzennnego h
            for (int i = 0; i < M; i++)
            {
                errorCT = fabs(AnalytMatrix[N - 1][i] - CrankThomasMatrix[N - 1][i]);
                if (errorCT > max_errprCT)
                    max_errprCT = errorCT;

                errorCJ = fabs(AnalytMatrix[N - 1][i] - CrankJacobMatrix[N - 1][i]);
                if (errorCJ > max_errorCJ)
                    max_errorCJ = errorCJ;

            }
            file_tmax_error << log10(h) << ";" << log10(max_errprCT) << ";" << log10(max_errorCJ)
<< endl;

            cout << h << endl;

            deleteMatrix(CrankJacobMatrix, N);
            deleteMatrix(CrankThomasMatrix, N);
            deleteMatrix(AnalytMatrix, N);
            deleteVector(hLevel);
            deleteVector(dtLevel);
        }
        file_tmax_error.close();
        cout << "End" << endl;
        system("pause");
        return 0;
    }
```