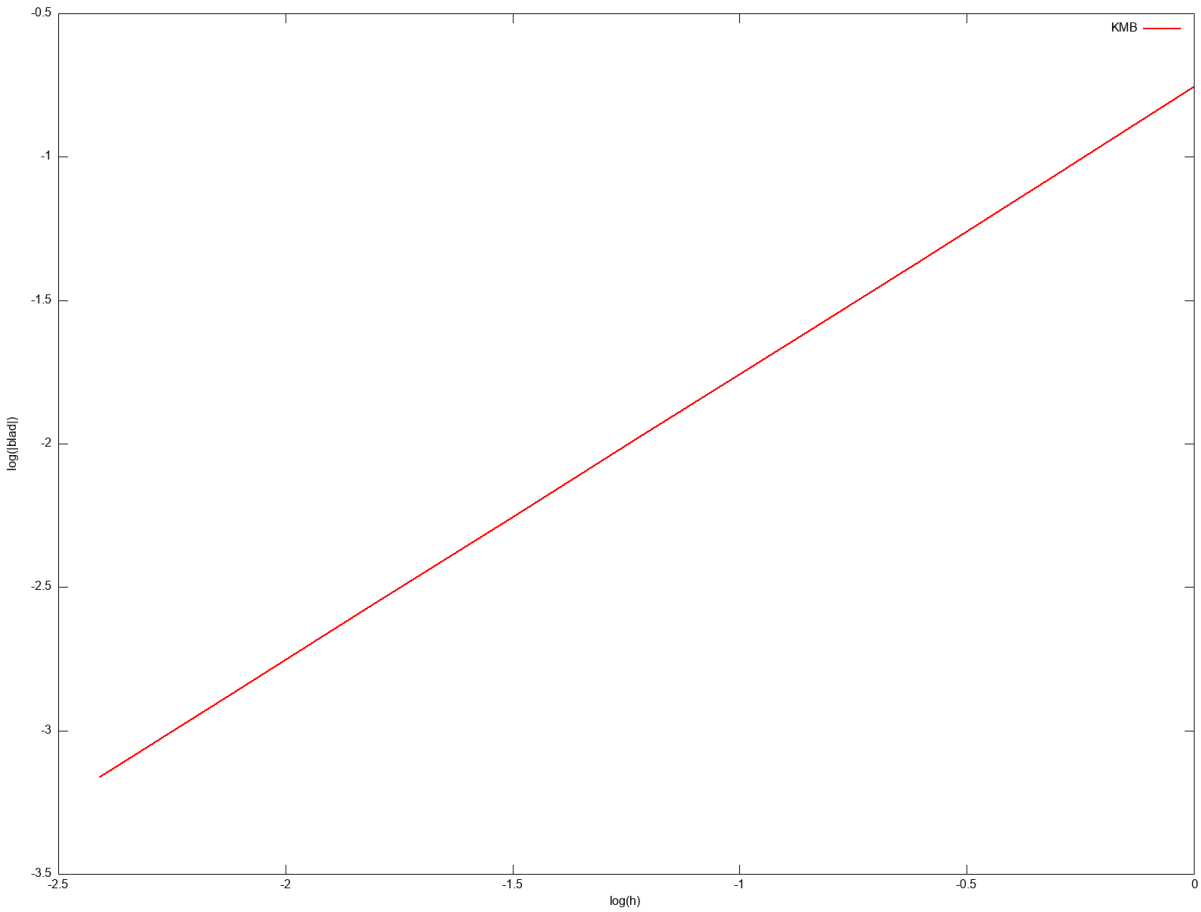
Sprawozdanie Laboratorium 11

Jakub Łopata gr. lab 03

1. Wykresy logarytmu z maksymalnej wartości bezwzględnej błędu od różnych wartości kroku przestrzennego w chwili t_{max}=2

a. Klasyczna metoda bezpośrednia:

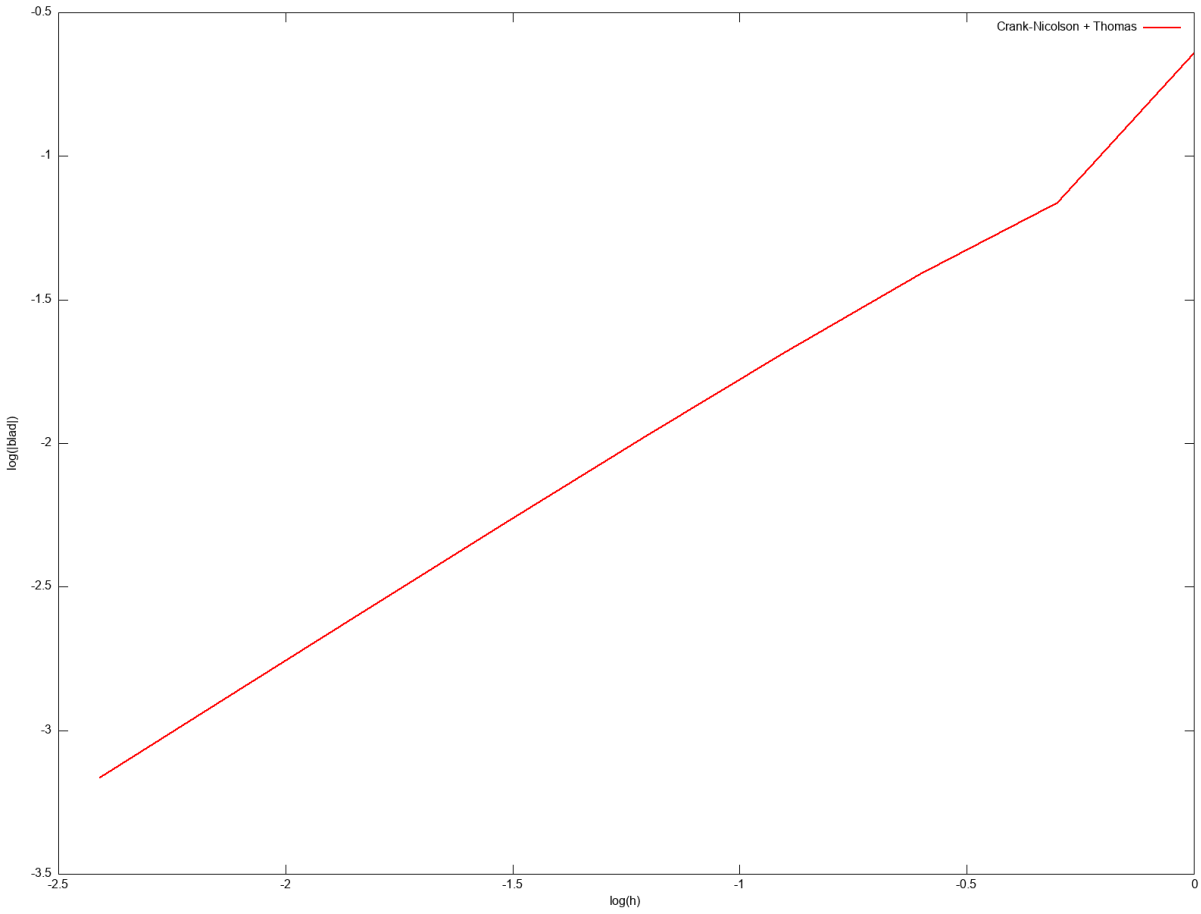


Rząd dokładności możemy wyznaczyć obliczając tangens kąta nachylenia wykresu do osi Ox

$$tg\alpha = \frac{-1.66091}{-0.90309} \approx 1{,}84 \approx 2$$

Co zgadza się z teoretycznym rzędem dokładności metody

b. Metoda pośrednia Cranka-Nicolson z algorytmem Thomasa

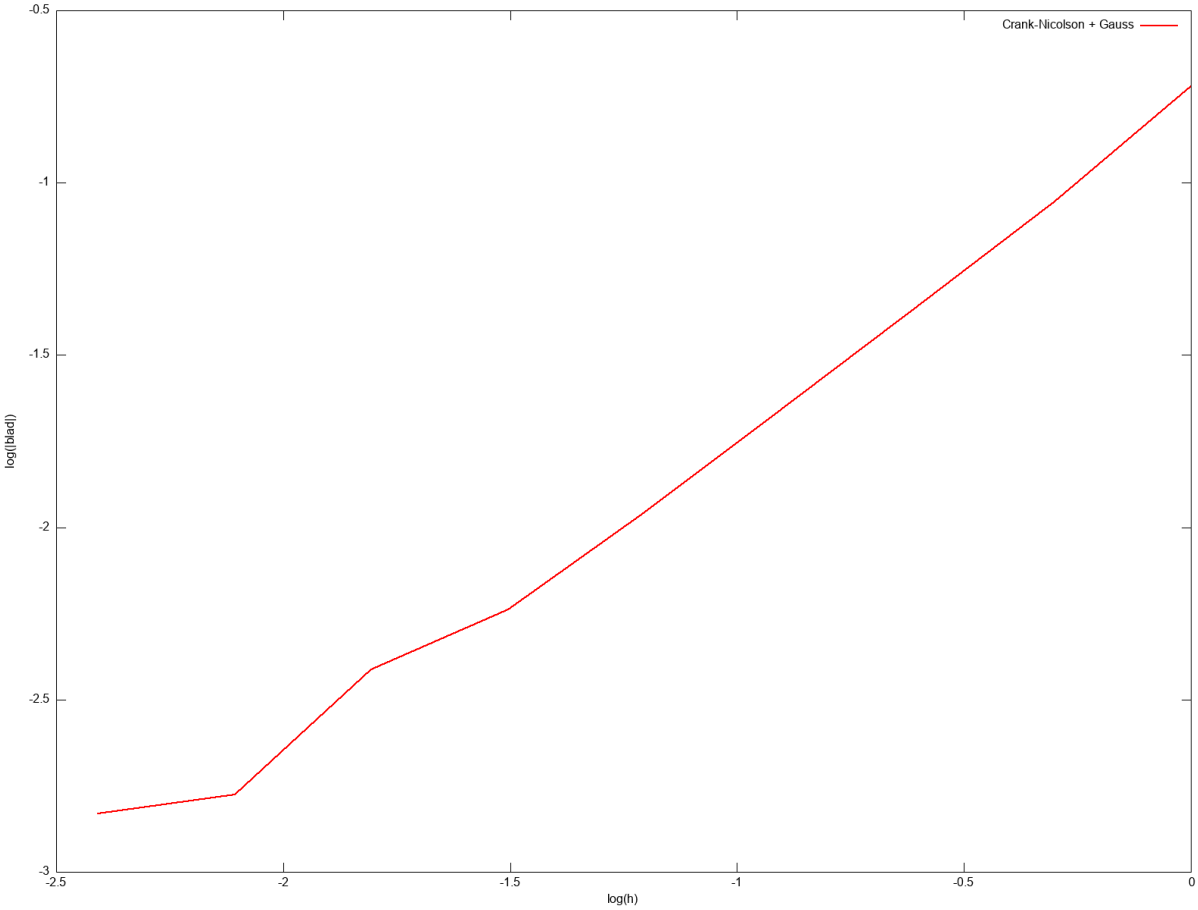


Rząd dokładności możemy wyznaczyć obliczając tangens kąta nachylenia wykresu do osi Ox

$$tg\alpha = \frac{-1.68505}{-0.90309} \approx 1{,}87 \approx 2$$

Co zgadza się z teoretycznym rzędem dokładności metody

c. Metoda pośrednia Cranka-Nicolson z algorytmem Gaussa-Seidela



Rząd dokładności możemy wyznaczyć obliczając tangens kąta nachylenia wykresu do osi Ox

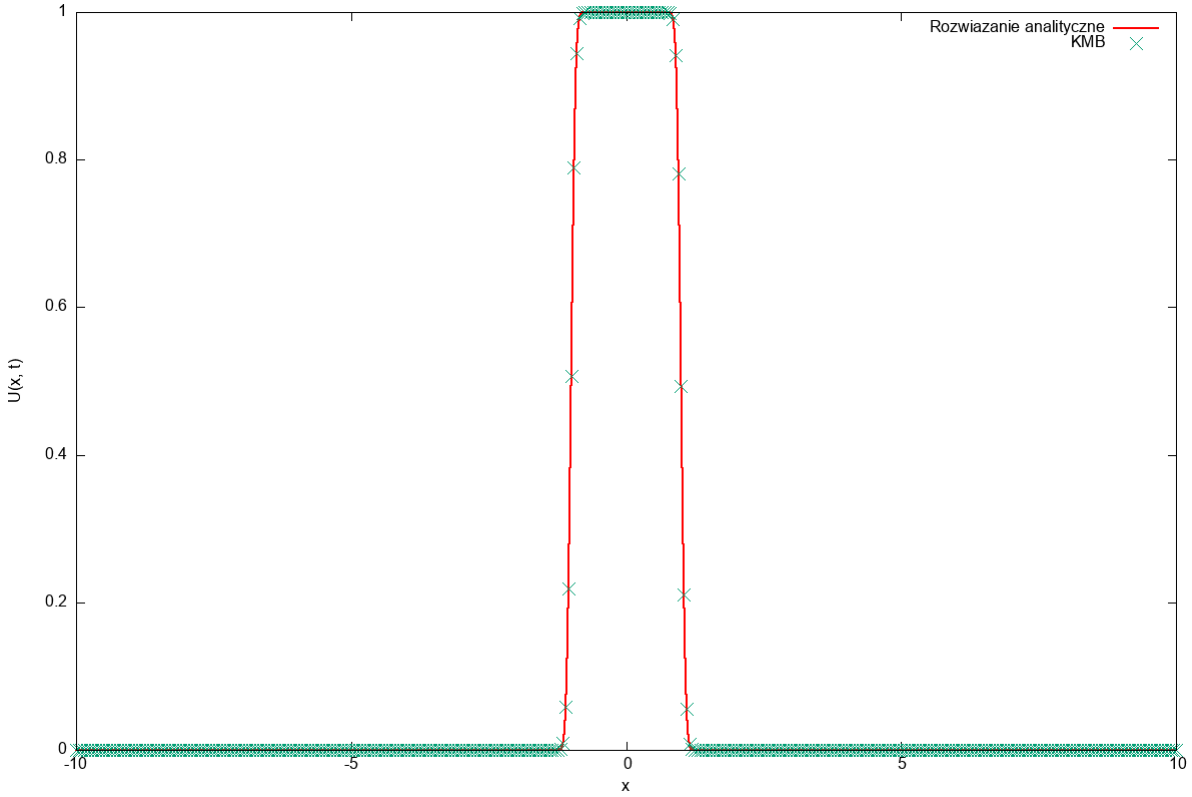
$$tg\alpha = \frac{-1.65819}{-0.90309} \approx 1{,}84 \approx 2$$

Co zgadza się z teoretycznym rzędem dokładności metody

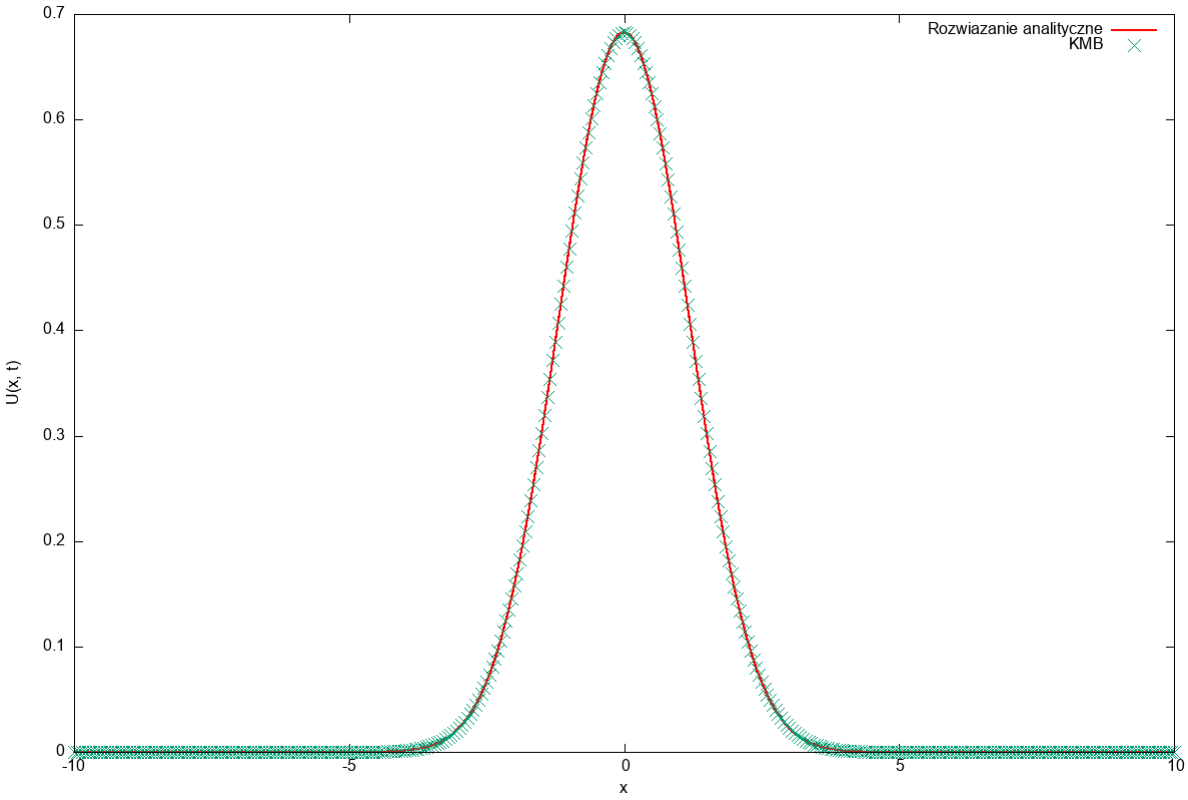
2. Wykresy rozwiązań numerycznych i rozwiązania analitycznego
   a. Klasyczna Metoda Bezpośrednia

$$h = 0{,}002$$
$$\lambda = 0{,}4$$
$$dt = 0{,}0000016$$

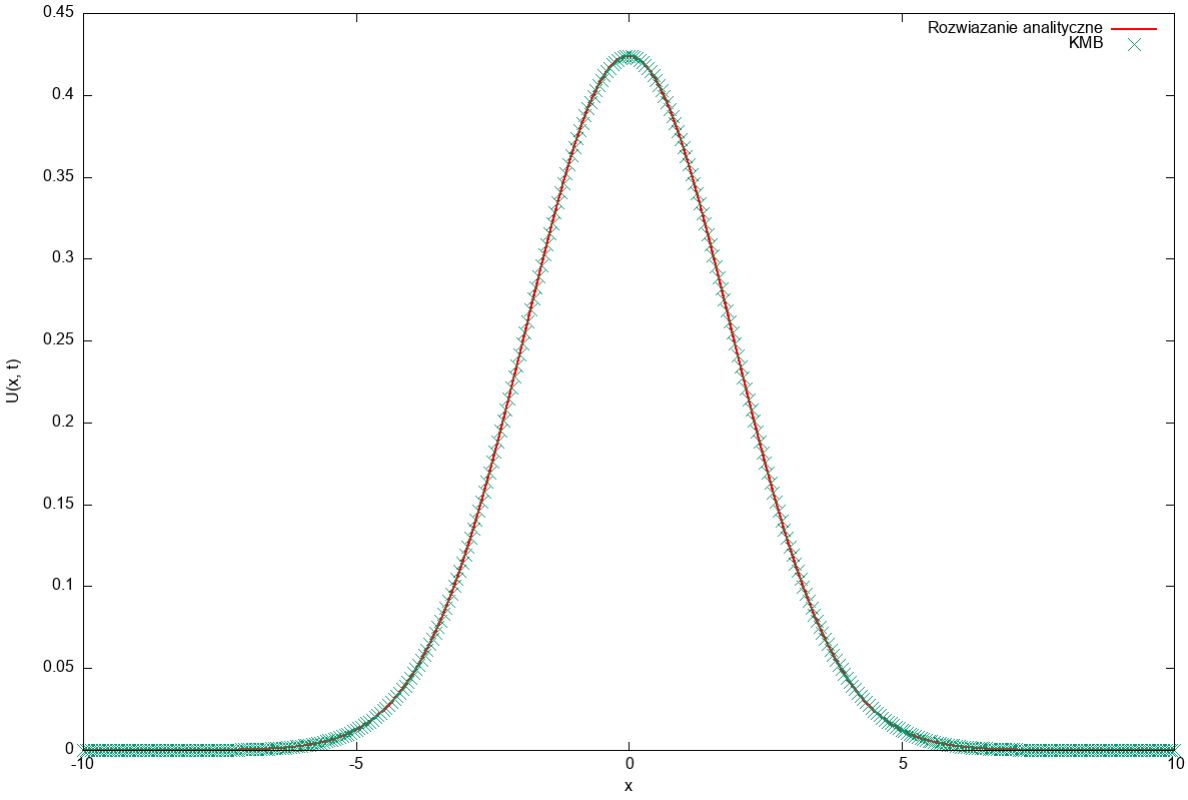
$$t = 0{,}0019984$$



$t = 0{,}499999999998$



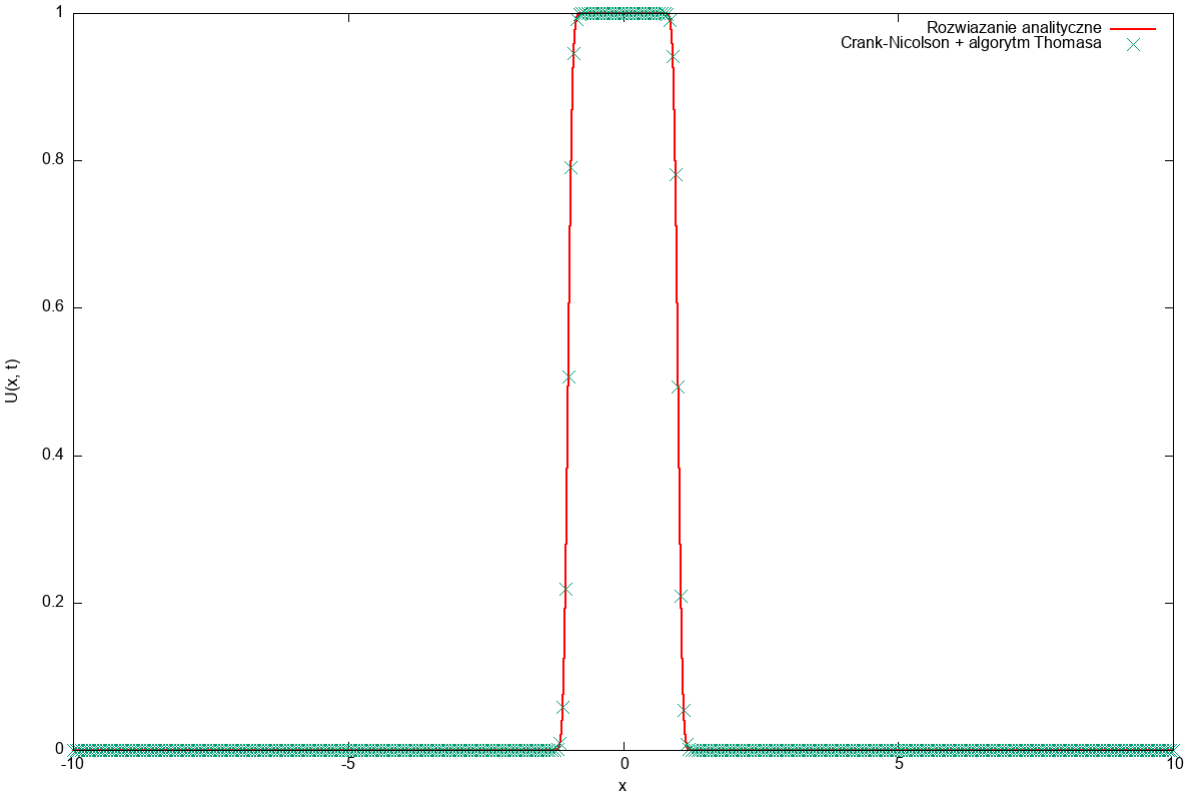
$t = 1{,}5999984$



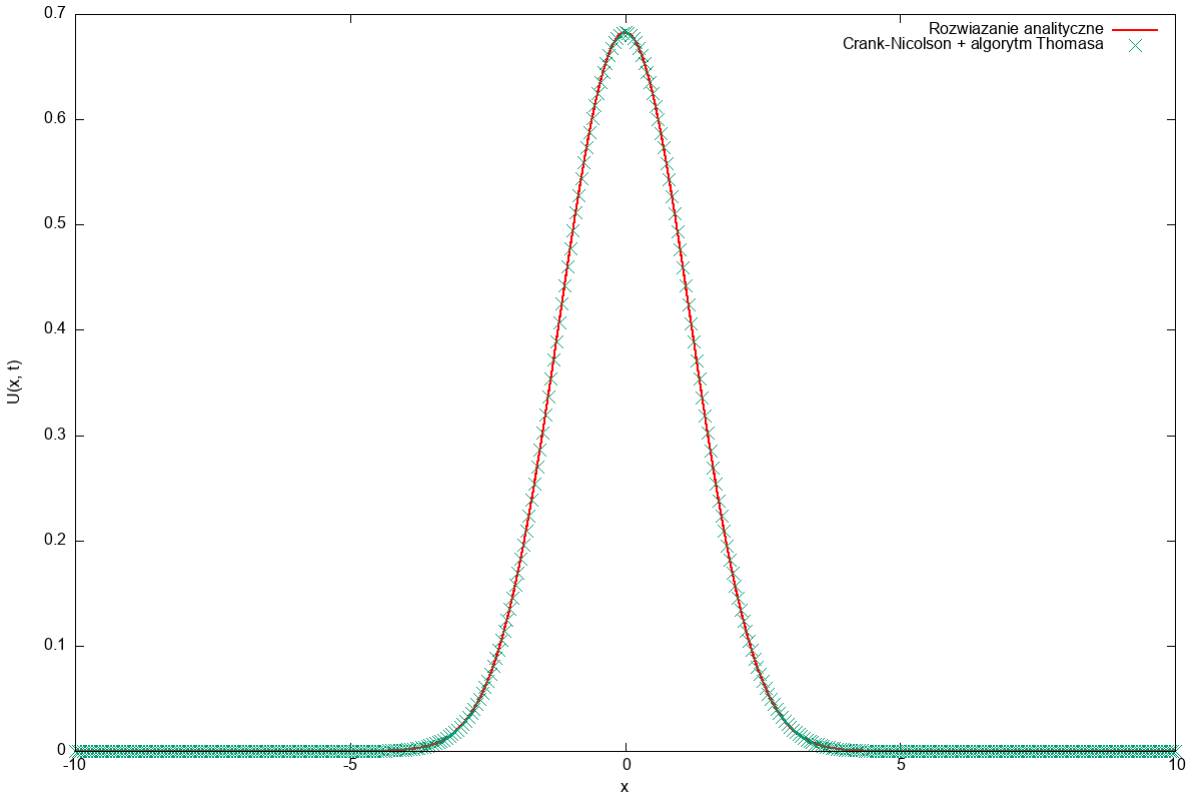
b. Metoda pośrednia Cranka-Nicolson z algorytmem Thomasa

$$h = 0{,}002$$
$$\lambda = 1$$
$$dt = 0{,}000004$$

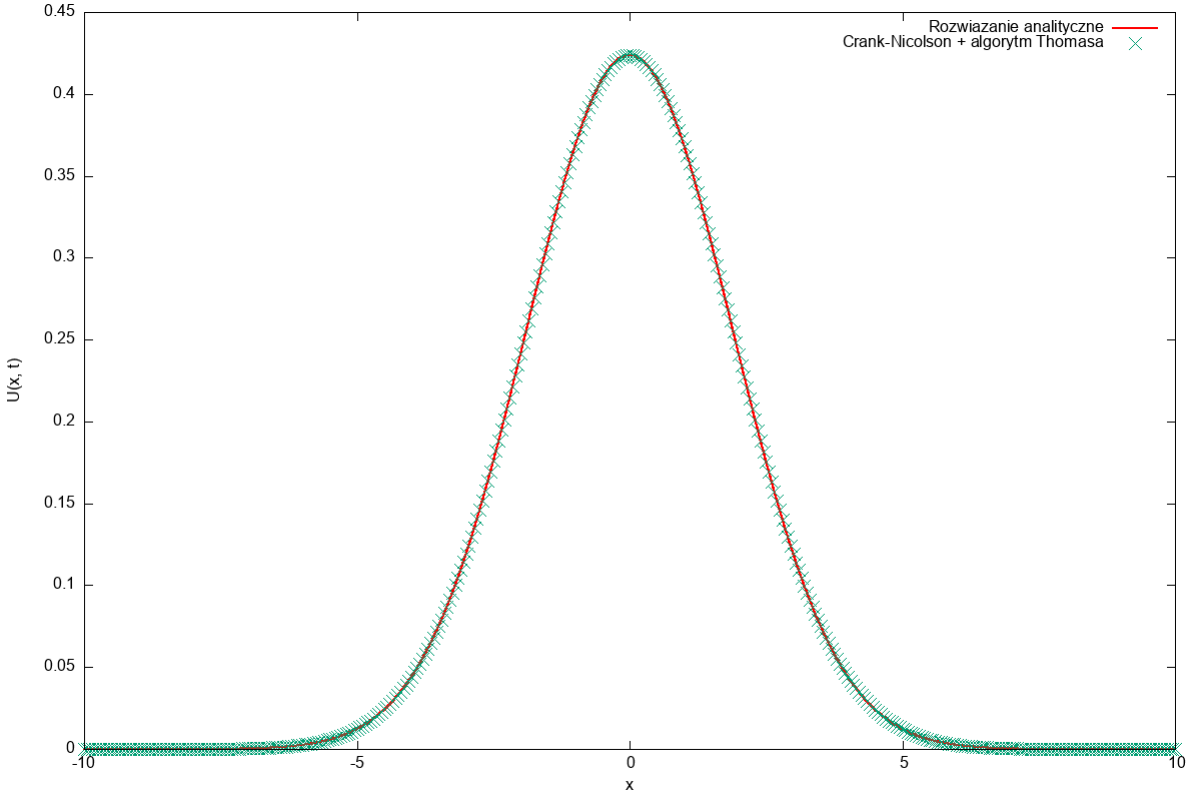
$$t = 0{,}001996$$



$t = 0{,}499996$



$t = 1{,}6$



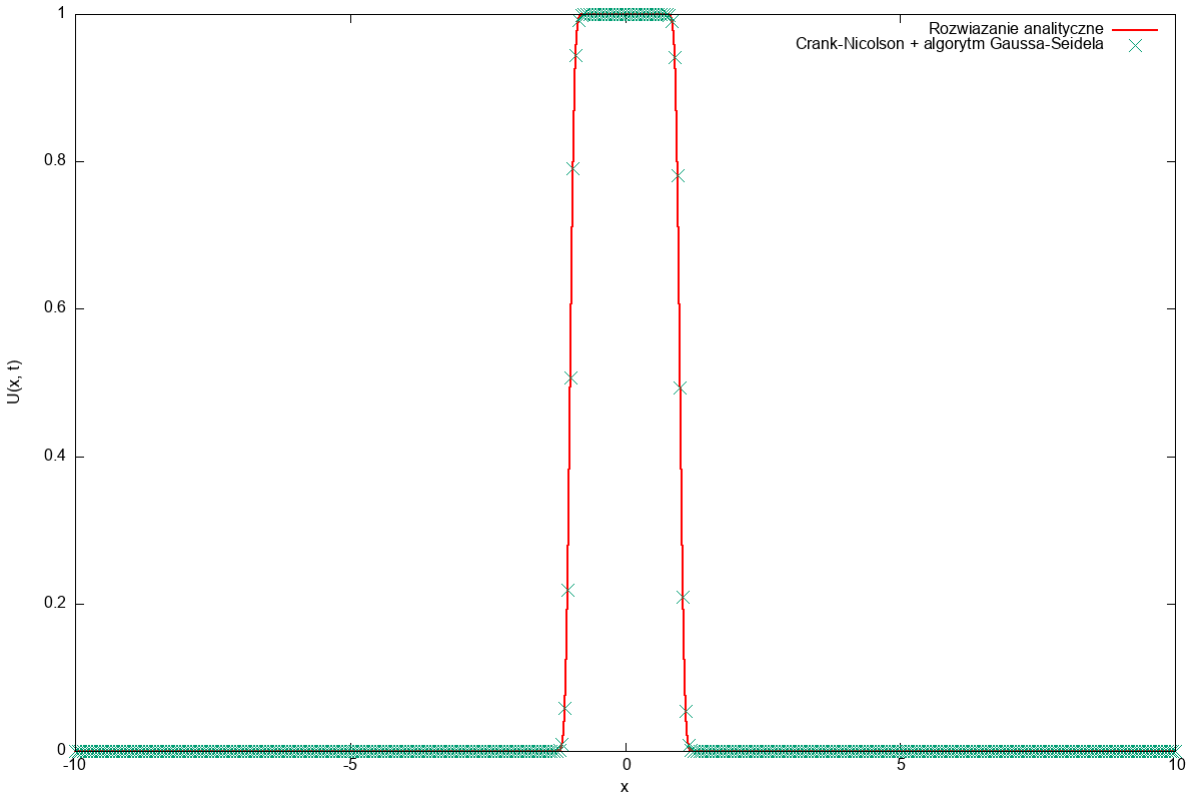
c. Metoda pośrednia Cranka-Nicolson z algorytmem Gaussa-Seidela

$$h = 0{,}002$$

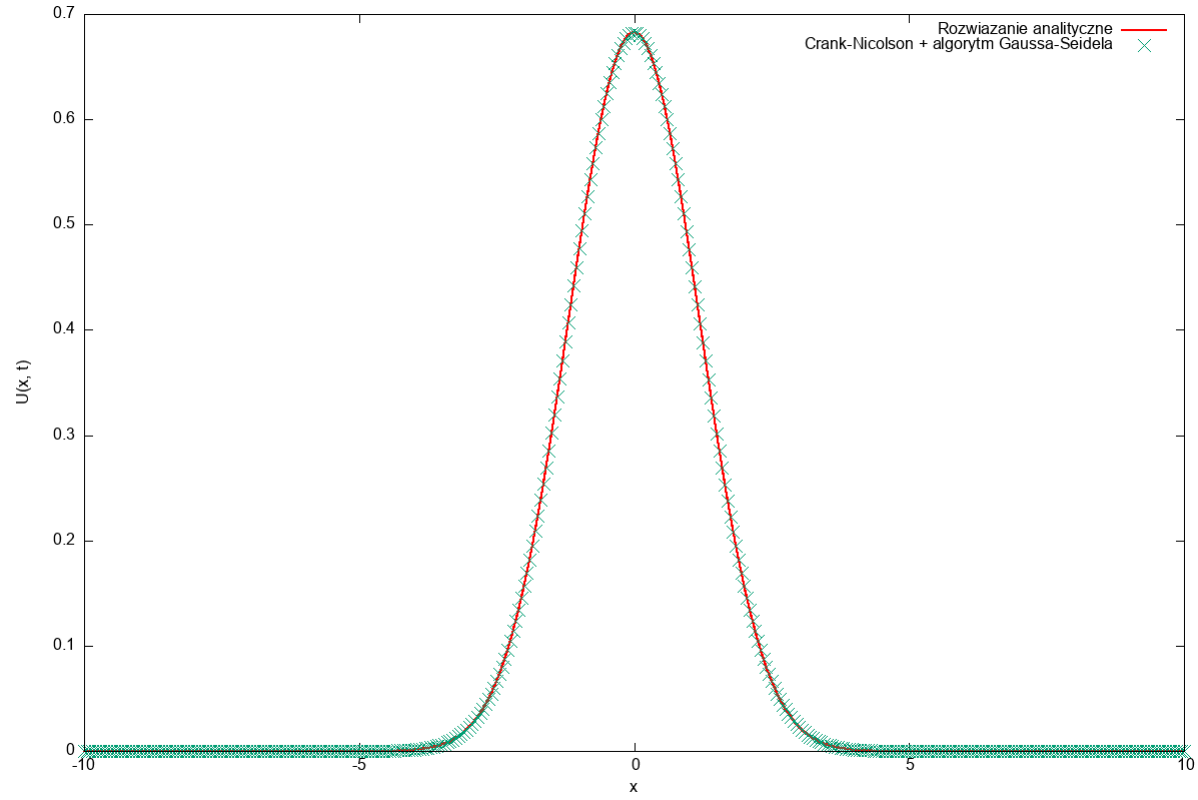
$$\lambda = 1$$

$$dt = 0{,}000004$$

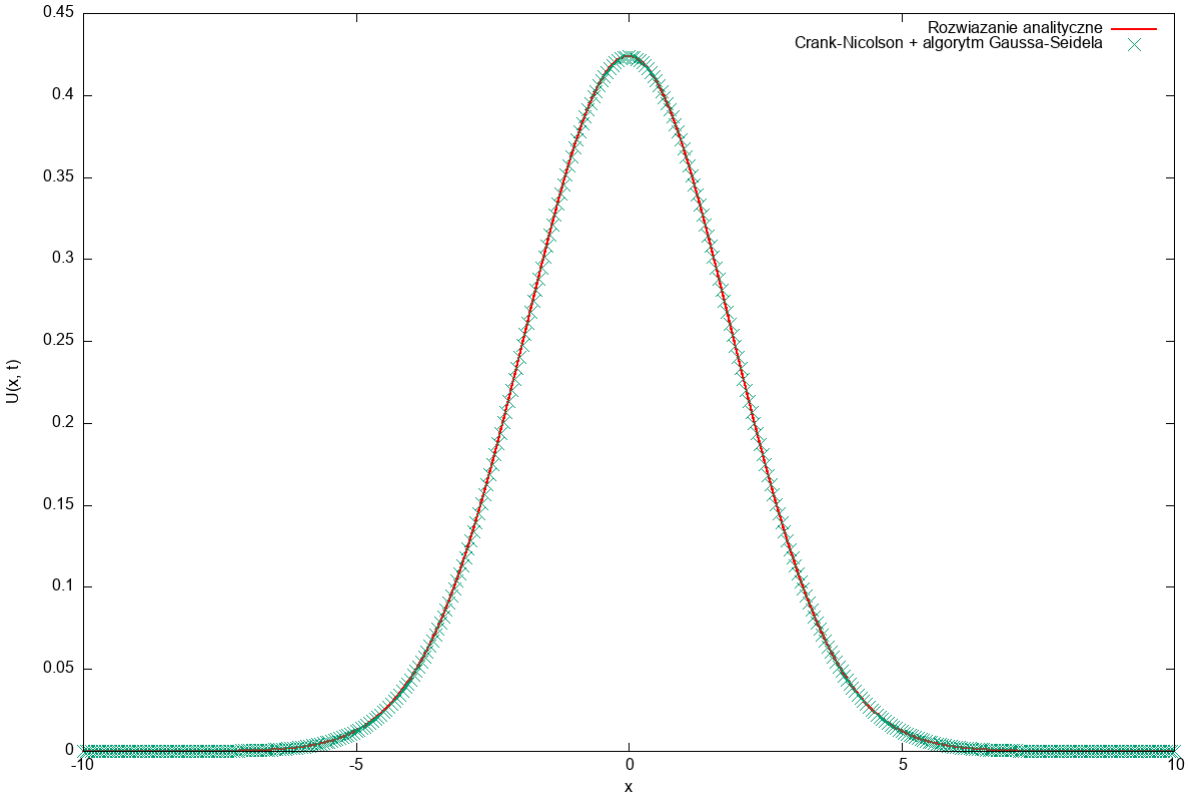
$$t = 0{,}001996$$



$t = 0{,}499996$



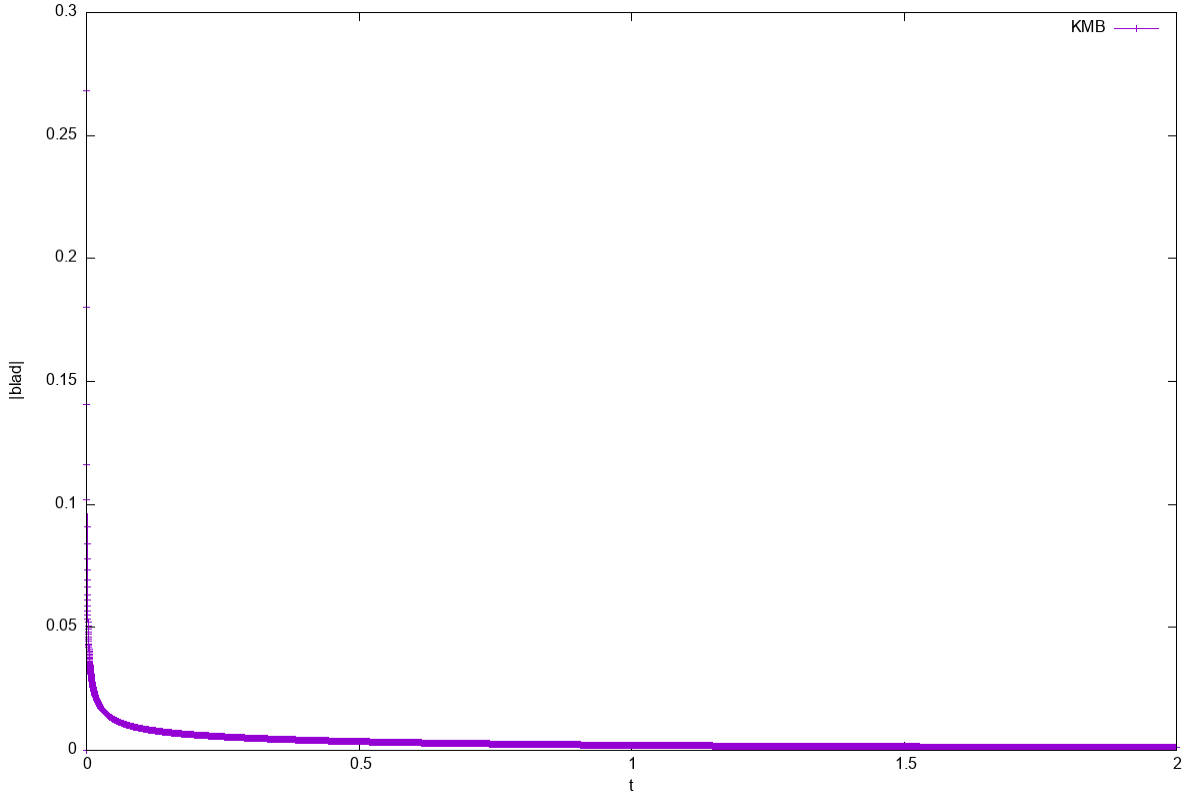
$t = 1{,}6$



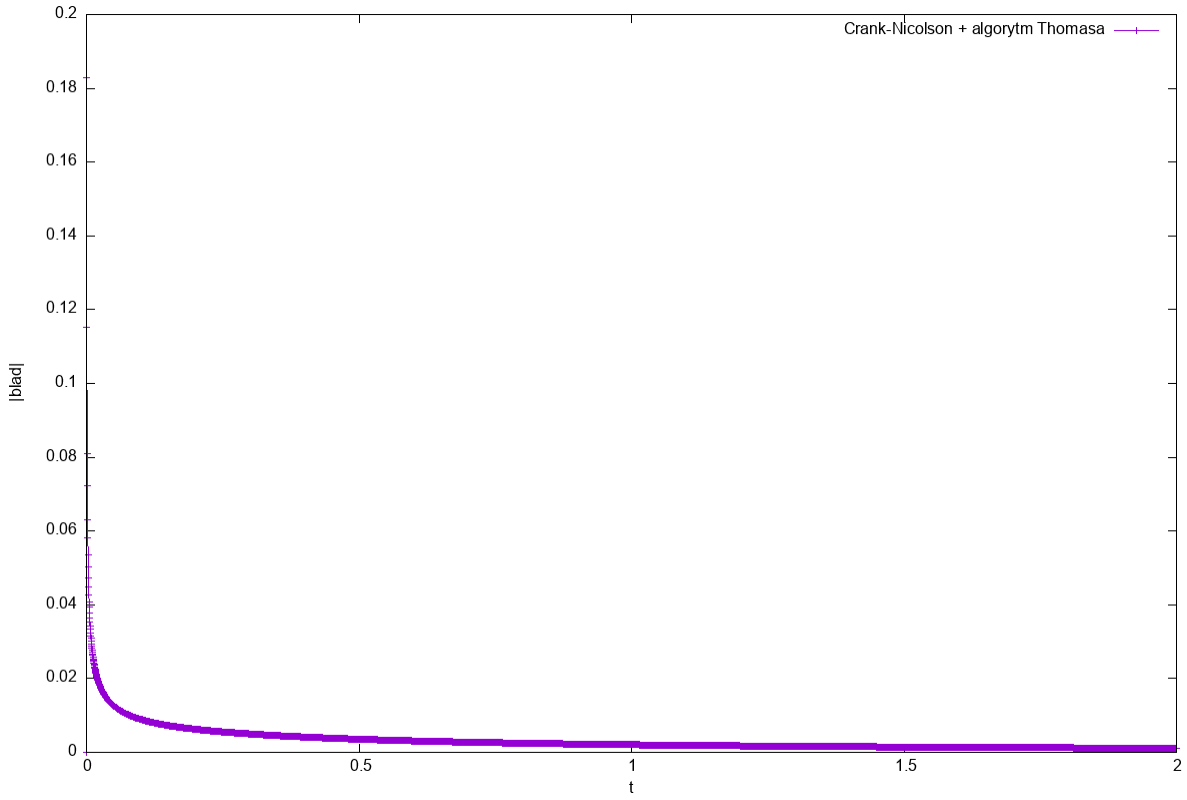
W przypadku wszystkich wykresów rozwiązania numeryczne pokrywają się wizualnie z rozwiązaniem analitycznym. W przypadku algorytmu Gaussa-Seidela do uzyskania takiego efektu niezbędne było zastosować odpowiednią tolerancję błędu oraz tolerancję reziduum.

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

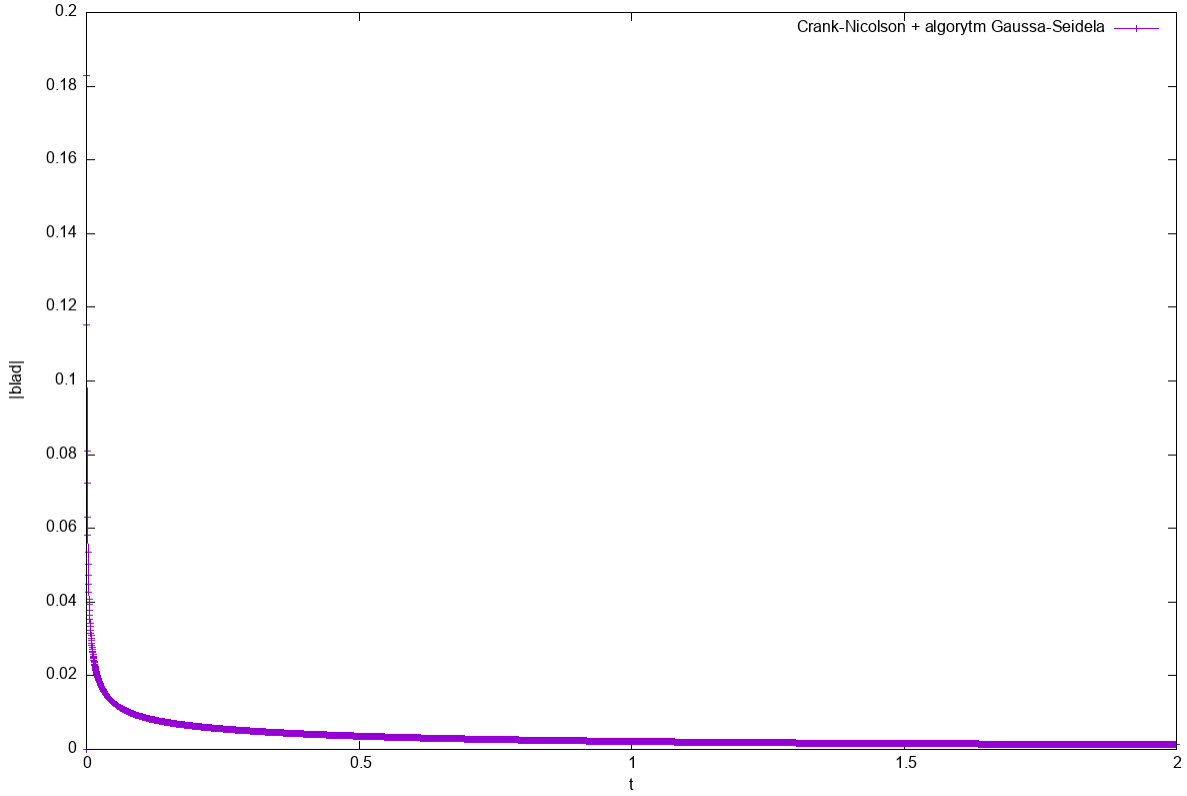
a. Klasyczna Metoda Bezpośrednia



b. Metoda pośrednia Cranka-Nicolson z algorytmem Thomasa



c. Metoda pośrednia Cranka-Nicolson z algorytmem Gaussa-Seidela



W pierwszej chwili czasowej t = 0 błąd dla każdej metody wynosi 0. W następnej chwili czasowej błąd rośnie, w przypadku Klasycznej Metody Bezpośredniej wynosi on ok. 0,26, natomiast w przypadku metody Cranka-Nicolson ok. 0,18. W obu przypadkach widać następnie wyhamowujący spadek wartości maksymalnej błędu.

Program:

Gauss.cpp

```
//funkcja sprawdzająca estymator błędu
bool checkEstimatorNew(const double* prev, const double* next, int n){
    double tol = 0.000001;
    for (int i = 0; i < n; ++i) {
        if(fabs(next[i] - prev[i])>tol){
            return false;
        }
    }
    return true;
}
//funkcja sprawdzająca reziduum
bool checkResiduumNew(const double* v, const double* upperDiagonal, const
double* lowerDiagonal, const double* diagonal, const double* b, int n){
    double tol = 0.00000001;
    if((b[0] - diagonal[0] * v[0] - upperDiagonal[0] * v[1])>tol || (b[n -
1] - lowerDiagonal[n - 2] * v[n - 2] - diagonal[n - 1] * v[n - 1])>tol)
        return false;
    for (int i = 1; i < n-1; ++i) {
        if((b[i] - lowerDiagonal[i-1] * v[i - 1] - diagonal[i] * v[i] -
upperDiagonal[i] * v[i + 1])>tol){
            return false;
        }
    }
    return true;
}


//funkcja wykonująca algorytm gaussa-seidla na macierzy trójdiagonalnej
podanej przez 3 wektory przekątnych
void gaussSeidelTridiagonal(int n, const double* upperDiagonal, const
double* lowerDiagonal, const double* diagonal, double* b, double* xo, int
iterations) {
    auto* x1 = new double[n];


    for (int k = 0; k < iterations; ++k) {
        for (int i = 0; i < n; ++i) {
            if (i == 0)
                xo[i] = (b[i] - upperDiagonal[i]*x1[i+1])/diagonal[i];
            if (i == (n - 1))
                xo[i] = (b[i] - lowerDiagonal[i-1]*x1[i-1])/diagonal[i];
            else
                xo[i] = (b[i] - (upperDiagonal[i]*x1[i+1]+lowerDiagonal[i-
1]*x1[i-1]))/diagonal[i];
        }
```

```
        if (k + 1 == iterations) {
            cout << "Nie zbiezne" << '\n';
        }

        if (checkEstimatorNew(xo, x1, n) && checkResiduumNew(x1,
upperDiagonal, lowerDiagonal, diagonal, b, n)) {

            for (int i = 0; i < n; ++i) {
                x1[i] = xo[i];
            }
            break;
        }
        for (int i = 0; i < n; ++i) {
            x1[i] = xo[i];
        }

    }
    for (int i = 0; i < n; ++i) {
        b[i] = xo[i];
    }

    delete[] x1;
}
```

Thomas.cpp

```
//funkcja przeprowadzająca procedurę na macierzy
void matrixProcedure(const double * lowDiagonal, double * diagonal, const
double * upDiagonal, int n){
    for(int i = 1; i<n; i++){
        diagonal[i] = diagonal[i] - lowDiagonal[i-1]*(1.0/diagonal[i-
1])*upDiagonal[i-1];
    }
}
//funkcja przeprowadzająca procedurę na vektorze rozwiązań
void vectorProcedure(double *b, const double * lowDiagonal, const double *
diagonal, const double * upDiagonal, int n){
    for(int i = 1; i<n; i++){
        b[i] = b[i] - lowDiagonal[i-1]*(1.0/diagonal[i-1])*b[i-1];
    }
    b[n-1] = b[n-1]*1.0/diagonal[n-1];
    for(int i = n-2; i>=0; i--){
        b[i] = (1.0/diagonal[i])*(b[i]-upDiagonal[i]*b[i+1]);
    }
}
```

Main.cpp

```
#include <iostream>
#include <fstream>
#include "thomas.h"
#include "gauss.h"
#include "calerf.h"
#include <ctime>
#include <iomanip>

using std::string, std::cout;

//enum używany do wyznaczenia rodzaju dyskretyzacji i algorytmu
enum method{
    KMBEN,
    CRANK_NICOLSON_THOMAS,
    CRANK_NICOLSON_GAUSS
};

//rozwiązanie analityczne
double analiticSol(double x, double t){
    return 0.5* calerfpack::erf_l((x+1.0)/(2.0* sqrt(t))) - 0.5 *
calerfpack::erf_l((x-1.0)/(2.0* sqrt(t)));
}
//funkcja obliczająca ilość podprzedziałów
int calcN(double h, double a){
    return (int)(2.0*a/h)+1.0;
}

//funkcja ustawiająca rozwiązanie w chwili t=0 (warunek początkowy)
double * setFirstVector(int n, double h, double a){
    auto *ui0 = new double [n];
    double xi = -a;
    for(int i = 0; i < n; i++){
        ui0[i] = fabs(xi)<1.0 ? 1.0 : 0.0;
        xi+=h;
    }
    return ui0;
}

double calcTKMB(double h){
    return 2.*h*h/5.;
}

double calcTCrankNicolson(double h){
    return h*h;
}
//realizacja klasycznaj metody bezpośredniej
void KMB(int n, double lambda, double * uk, const double *ui0){
    for(int i = 1; i < n-1; i++){
        uk[i] = lambda*ui0[i-1] + (1.-2.*lambda)*ui0[i] + lambda*ui0[i+1];
    }
    uk[0] = 0.0;
    uk[n-1] = 0.0;
}
//realizacja metody Cranka-Nicolson
void CrankNicolson(int n, double lambda, double *uk, double *ui0, method
method){
    auto *u = new double[n-1];
    auto *d = new double[n];
    auto *l = new double[n-1];
```

```
    u[0] = 0.0;
    d[0] = 1.0;

    d[n-1] = 1.0;
    l[n-2] = 0.0;

    uk[0] = 0.0;
    uk[n-1] = 0.0;

    for (int i = 1; i < n-1; ++i) {
        l[i-1] = lambda/2.;
        d[i] = -(1.+lambda);
        u[i] = lambda/2.;
        uk[i] = -(lambda/2.*ui0[i-1]+(1.-
lambda)*ui0[i]+lambda/2.*ui0[i+1]);
    }
    if(method == CRANK_NICOLSON_THOMAS){
        matrixProcedure(l,d,u, n);
        vectorProcedure(uk, l, d, u, n);
    }else{
        gaussSeidelTridiagonal(n,u,l,d,uk,ui0,400);
    }
    delete[] l;
    delete[] d;
    delete[] u;

}
//funkcja szukająca maksymalnego błędu
double findMaxError(double h, double t, double *a, int n){
    double err = 0.0, tmp;
    double xi = -10.0;
    for (int i = 0; i < n; ++i) {
        tmp = fabs(a[i] - analiticSol(xi,t));
        if(tmp>err)
            err = tmp;
        xi+=h;
    }
    return err;
}
//funkcja zapisująca do pliku rozwiązania analityczne oraz numeryczne dla
danej chwili t
void saveToFile(int n, double h, double t, double dt, double lambda, double
*uk, string filename, string plikanalit, int num){
    cout << t << " lambda: " << lambda << " dt: " << dt << '\n';
    ofstream plik1(filename + std::to_string(num) + ".txt");
    ofstream plik2(plikanalit + std::to_string(num) + ".txt");

    double xi = -10.0;
    for (int i = 0; i < n; ++i) {
        plik1 << xi << " " << uk[i] << '\n';
        plik2 << xi << " " << analiticSol(xi, t) << '\n';
        xi += h;
    }

    plik1.close();
    plik2.close();
}
//funkcja zapisująca wyniki dla t zbliżonych do określonych parametrami
t1,t2,t3
//lub zapisująca maksymalną watrość błędu dla danego czasu t - w zależności
```

```
od parametru type
void solve(double h, string filename, string plikanalit, method method,
double t1, double t2, double t3, int type){
    double dt = method==KMBEN ? calcTKMB(h) : calcTCrankNicolson(h);
    double lambda =  dt/(h*h);
    int n = calcN(h, 10.0);
    double* uk = new double[n], *tmp;
    double* ui0 = setFirstVector(n ,h, 10.0);
    ofstream plik3;
    if(type==1){
        plik3.open(plikanalit +"Error.txt");
        plik3 << 0 << " " << findMaxError(h, 0,ui0, n) << '\n' ;
        plik3.close();
    }
    for(double t = 0.0+dt; t<=2.0; t+=dt){
        if(method == KMBEN){
            KMB(n, lambda, uk, ui0);
        }
        else{
            CrankNicolson(n, lambda, uk, ui0, method);
        }
        if(type==0){
            if(t < t1 && t > t1-dt){
                saveToFile(n, h, t, dt, lambda, uk, filename, plikanalit,
1);
            } else if(t < t2 && t > t2-dt){
                saveToFile(n, h, t, dt, lambda, uk, filename, plikanalit,
2);
            } else if(t < t3 && t > t3-dt){
                saveToFile(n, h, t, dt, lambda, uk, filename, plikanalit,
3);
            }
        }
        if(type==1){
            plik3.open(plikanalit +"Error.txt", ios::app);
            plik3 << t << " " << findMaxError(h, t,uk, n) << '\n' ;
            plik3.close();
        }
        tmp = ui0;
        ui0 = uk;
        uk = tmp;
    }
}


//funkcja zapisująca logarytm z maksymalnego błędu bezwzględnego oraz
logarytm z kroku h
void solveError(double h, string filename, method method){
    double dt = method==KMBEN ? calcTKMB(h) : calcTCrankNicolson(h);
    double lambda =  dt/(h*h);
    double tt, maxError;
    int n = calcN(h, 10.0);
    double* uk = new double[n], *tmp;
    double* ui0 = setFirstVector(n ,h, 10.0);
    ofstream plik1;
    ofstream plik2;
    for(double t = 0.0+dt; t<=2.0; t+=dt){
        if(method == KMBEN){
            KMB(n, lambda, uk, ui0);
        }
        else{
```

```
            CrankNicolson(n, lambda, uk, ui0, method);
        }
        tmp = ui0;
        ui0 = uk;
        uk = tmp;
        tt = t;
    }
    maxError = findMaxError(h, tt, uk, n);
    plik1.open(filename ,ios::app);
    plik1 << log10(h) << " " << log10(maxError) << '\n';
    plik1.close();
    delete[] uk;
    delete[] ui0;
}
```

```
int main() {
    std::time_t currentTime;
    cout << std::setprecision(30);
    for (double h = 1; h > 0.003; h/=2) {
        cout << h << '\n';
        currentTime = std::time(nullptr);
        cout << "KMB START " <<
std::put_time(std::localtime(&currentTime), "%H:%M:%S") << '\n';
        solveError(h, "error2KMB.txt", KMBEN);
        currentTime = std::time(nullptr);
        cout << "THOMAS START " <<
std::put_time(std::localtime(&currentTime), "%H:%M:%S") << '\n';
        solveError(h, "error2Thomas.txt", CRANK_NICOLSON_THOMAS);
        currentTime = std::time(nullptr);
        cout << "GAUSS START " <<
std::put_time(std::localtime(&currentTime), "%H:%M:%S") << '\n';
        solveError(h, "error2Gauss.txt", CRANK_NICOLSON_GAUSS);
    }
    solve(0.002, "wynikKMB1", "KMB1", KMBEN, 0.002, 0.5, 1.6,0);
    solve(0.002, "wynikThomas1", "Thomas1", CRANK_NICOLSON_THOMAS, 0.002,
0.5, 1.6,0);
    solve(0.002, "wynikGauss1", "Gauss1", CRANK_NICOLSON_GAUSS, 0.002, 0.5,
1.6,0);
    solve(0.02, "wynikKMB1", "KMB1", KMBEN, 0.002, 0.5, 1.6,1);
    solve(0.02, "wynikThomas1", "Thomas1", CRANK_NICOLSON_THOMAS, 0.002,
0.5, 1.6,1);
    solve(0.02, "wynikGauss1", "Gauss1", CRANK_NICOLSON_GAUSS, 0.002, 0.5,
1.6,1);

    return 0;
}
```