

Numeryczne wyznaczanie rozwiązań równania Burgersa przy pomocy metody różnic skończonych

Konrad Bonicki, Tomasz Orzechowski, Maciej Pestka

24 czerwca 2024

Spis treści

1	Wprowadzenie	3
2	Równanie Burgersa	3
3	Transformacja Hopf-Cole'a	3
4	Rozwiązanie analityczne przy pomocy transformacji Hopf-Cole'a	4
5	Metody numeryczne - wstęp	7
6	Wzór Taylora	7
7	Metoda różnic skończonych	7
8	Metoda Eulera	9
9	Metoda Rungego-Kutty	9
10	Rozwiązanie równania Burgersa	10
11	Opis programu transformaty Hopf Cole'a	19
11.1	transformataHopfCole.h	19
11.2	transformataHopfCole.cpp	21
11.3	transformata.cpp	24

12 wyniki	24
13 Podsumowanie	26
14 dodatek A	27
15 dodatek B	49
15.1 transformata.cpp	49
15.2 transformataHopfCole.h	49
15.3 transformataHopfCole.cpp	50

1 Wprowadzenie

W tej pracy przybliżymy równanie Burgersa oraz jego zastosowania, jego rozwiązanie analityczne i metody, które użyliśmy podczas pisania programu, który numerycznie wyznacza rozwiązania przytoczonego równania.

2 Równanie Burgersa

Równanie Burgersa w ogólnej postaci jest fundamentalnym, nieliniowym, parabolicznym równaniem różniczkowym cząstkowym drugiego rzędu występującym w przeróżnych obszarach zastosowań matematyki, takich jak mechanika płynów, dynamika gazów oraz płynność ruchu:

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = v \frac{\partial^2 u}{\partial x^2} \quad (1)$$

gdzie:

- t - zmienna niezależna zwykle interpretowana jako czas,
- x - zmienna niezależna zwykle interpretowana jako położenie,
- $u(x, t)$ - zmienna zależna zwykle interpretowana jako prędkość płynu,
- v - stały parametr, zwykle interpretowany jako lepkość płynu.

3 Transformacja Hopf-Cole'a

Transformacja Cole-Hopf'a jest zmianą zmiennych, która umożliwia przekształcenie specjalnego rodzaju parabolicznych równań różniczkowych cząstkowych z nieliniowością kwadratową w liniowe równanie ciepła. Wygląda następująco:

$$u(x, t) = -2v \frac{\theta_x(x, t)}{\theta(x, t)} \quad (2)$$

$$u = -2v \frac{\theta_x}{\theta}$$

Liczymy pochodne cząstkowe z (2), aby podstawić je potem do równania Burgersa

$$u_x = -2v \frac{\theta_{xx}\theta - \theta_x^2}{\theta^2}, \quad u_t = -2v \frac{\theta_{xt}\theta - \theta_x\theta_t}{\theta^2}$$

$$u_{xx} = -2v \frac{\theta_{xxx}\theta^2 - 3\theta\theta_x\theta_{xx} + 2\theta_x^3}{\theta^3}$$

Podstawiamy teraz to co nam wyszło do równania Burgersa:

$$\begin{aligned} u_t + uu_x &= vu_{xx} \\ -2v \frac{\theta_{xt}\theta - \theta_x\theta_t}{\theta^2} + 4v^2 \frac{\theta\theta_x\theta_{xx} - \theta_x^3}{\theta^3} &= -2v^2 \frac{\theta_{xxx}\theta^2 - 3\theta\theta_x\theta_{xx} + 2\theta_x^3}{\theta^3} \\ -2v\theta(\theta_{xt}\theta - \theta_x\theta_t) + 4v^2(\theta\theta_x\theta_{xx} - \theta_x^3) &= -2v^2(\theta_{xxx}\theta^2 - 3\theta\theta_x\theta_{xx} + 2\theta_x^3) \\ \theta^2\theta_{xt} - \theta\theta_x\theta_t - 2v\theta\theta_x\theta_{xx} + 2\theta_x^3 &= v\theta_{xxx}\theta^2 - 3v\theta\theta_x\theta_{xx} + 2\theta_x^3 \\ \theta\theta_{xt} - \theta_x\theta_t - 2v\theta_x\theta_{xx} &= v\theta_{xxx} - 3v\theta_x\theta_{xx} \\ \theta\theta_{xt} - \theta_x\theta_t &= v(\theta_{xxx} - \theta_x\theta_{xx}) \\ \theta \frac{\partial^2}{\partial x \partial t} - \frac{\partial \theta}{\partial x} \frac{\partial \theta}{\partial t} &= v(\theta \frac{\partial^3 \theta}{\partial x^3} - \frac{\partial \theta}{\partial x} \frac{\partial^2 \theta}{\partial x^2}) \\ \frac{\partial \theta}{\partial t} (\theta \frac{\partial \theta}{\partial x} - \frac{\partial \theta}{\partial x}) &= v \frac{\partial^2 \theta}{\partial x^2} (\theta \frac{\partial \theta}{\partial x} - \frac{\partial \theta}{\partial x}) \\ \theta\theta_x - \theta_x &\neq 0 \\ \theta_t &= v\theta_{xx} \end{aligned}$$

Po podstawieniu transformacji Hopf-Cole'a do równania Burgersa otrzymaliśmy równanie ciepła.

4 Rozwiązanie analityczne przy pomocy transformacji Hopf-Cole'a

Równanie Burgersa:

$$u_t + uu_x = vu_{xx}$$

Warunek początkowy:

$$u(x, 0) = \sin(\pi x), \quad 0 < x < 1,$$

Warunki brzegowe:

$$u(0, t) = u(1, t) = 0, \quad t > 0.$$

Transformacja Hopf-Cole'a:

$$u(x, t) = -2v \frac{\theta_x(x, t)}{\theta(x, t)}$$

Z równania Burgersa z poprzedniego punktu mamy:

$$\theta_t = v\theta_{xx}, \quad 0 < x < 1 \quad t > 0$$

Warunek początkowy θ :

$$u(x, 0) = -2v \frac{\theta_x(x, 0)}{\theta(x, 0)}, \quad \theta(x, 0) \neq 0$$

$$\sin(\pi x) = -2v \frac{\theta_x(x, 0)}{\theta(x, 0)}$$

$$-\frac{1}{2v} \sin(\pi x) = \frac{\theta_x(x, 0)}{\theta(x, 0)}$$

$$-\frac{1}{2v} \int_0^x \sin(\pi x) dx = \int_0^x \frac{\theta_x(x, 0)}{\theta(x, 0)} dx$$

$$-\frac{1}{2v} \left[-\frac{1}{\pi} \cos(\pi x) \right]_0^{x=x} = [\ln[\theta(x, 0)]]_0^{x=x}$$

$$\frac{1}{2v\pi} (\cos(\pi x) - 1) = \ln[\theta(x, 0)] - \ln[\theta(0, 0)]$$

Wyznaczamy $\theta(0, 0)$:

$$u(0, 0) = -2v \frac{\theta_x(0, 0)}{\theta(0, 0)}$$

$$u(0, 0) = \sin(\pi 0) = 0$$

$$0 = \frac{\theta_x(0, 0)}{\theta(0, 0)}$$

$$\int_0^x 0 dx = \int_0^x \frac{\theta_x(0, 0)}{\theta(0, 0)} dx$$

$$\ln(\theta(0, 0)) = 0$$

$$\theta(0, 0) = e^0 = 1$$

Co daje:

$$\theta(x, 0) = e^{\frac{1}{2v\pi} (\cos(\pi x) - 1)}, \quad 0 < x < 1$$

Warunki brzegowe θ :

$$u(0, t) = u(1, t) = 0$$

$$-2v \frac{\theta_x(0, t)}{\theta(0, t)} = -2v \frac{\theta_x(1, t)}{\theta(1, t)} = 0, \quad \theta(0, t) \neq 0 \quad \theta(1, t) \neq 0$$

$$\theta_x(0, t) = \theta_x(1, t) = 0, \quad t > 0$$

Rozwiązanie:

$$\theta_t = v\theta_{xx}, \quad 0 < x < 1 \quad t > 0$$

Do rozwiązania tego równania wykorzystam metodę Fouriera:

$$\theta(x, t) = X(x)T(t)$$

$$X(x)T'(t) = vX''(x)T(t)$$

$$\frac{T'(t)}{vT(t)} = \frac{X''(x)}{X(x)} = -\lambda^2$$

$$T'(t) = -\lambda^2 v T(t)$$

$$T(t) = e^{-\lambda^2 vt}$$

$$X''(x) = -\lambda^2 X(x)$$

$$X(x) = A \sin(\lambda x) + B \cos(\lambda x)$$

$$\theta(x, t) = (A \sin(\lambda x) + B \cos(\lambda x))e^{-\lambda^2 vt}$$

$$\theta_x(x, t) = (A \cos(\lambda x) - B \sin(\lambda x))e^{-\lambda^2 vt}$$

$$\theta_x(0, t) = 0$$

$$(A \cos(\lambda 0) - B \sin(\lambda 0))e^{-\lambda^2 vt} = 0$$

$$Ae^{-\lambda^2 vt} = 0 \Leftrightarrow A = 0$$

$$\theta_x(1, t) = 0$$

$$(-B \sin(\lambda))e^{-\lambda^2 vt} = 0$$

Funkcja sinus przyjmuje wartości 0 dla $\lambda = n\pi$, $n = 0, 1, 2, \dots$, omijamy rozwiązanie trywialne $B = 0$:

Wracamy z wyznaczonymi współczynnikami do momentu przed różniczkowaniem funkcji θ :

$$\theta(x, t) = B_0 + \sum_{n=1}^{\infty} B_n \cos(n\pi x) e^{-n^2 \pi^2 vt}$$

Wyznaczamy współczynniki B_n ($n = 0, 1, 2, \dots$):

$$B_0 = \int_0^1 e^{\frac{1}{2v\pi}(\cos(\pi x)-1)} dx = e^{-\frac{1}{2v\pi}} \int_0^1 e^{\frac{\cos(\pi x)}{2v\pi}} dx$$

$$B_n = 2 \int_0^1 e^{\frac{1}{2v\pi}(\cos(\pi x)-1)} \cos(n\pi x) dx = 2e^{-\frac{1}{2v\pi}} \int_0^1 e^{\frac{\cos(\pi x)}{2v\pi}} \cos(n\pi x) dx$$

Rozwiązanie:

$$u(x, t) = 2v\pi \frac{\sum_{n=1}^{\infty} B_n \sin(n\pi x) e^{-n^2 \pi^2 vt}}{B_0 + \sum_{n=1}^{\infty} B_n \cos(n\pi x) e^{-n^2 \pi^2 vt}}$$

5 Metody numeryczne - wstęp

Metod numerycznych używamy do rozwiązywania problemów matematycznych za pomocą operacji na liczbach. Metody numeryczne są wykorzystywane, gdy problem nie posiada rozwiązania analitycznego lub gdy takie rozwiązanie jest zbyt skomplikowane do zastosowania. W naszym projekcie korzystamy z metody różnic skończonych, metody Rungego-Kuty'ego oraz metody Eulera.

6 Wzór Taylora

Wzór Taylora jest ważnym narzędziem w analizie numerycznej:

Twierdzenie 1 *Jeśli $f \in C^n[a, b]$ i jeśli $f^{(n+1)}$ istnieje w przedziale otwartym (a, b) , to dla dowolnych punktów c i x z przedziału domkniętego $[a, b]$ mamy:*

$$f(x) = \sum_{k=0}^n \frac{1}{k!} f^{(k)}(c)(x-c)^k + E_n(x), \quad (3)$$

gdzie dla pewnego punktu ξ leżącego między c i x :

$$E_n(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi)(x-c)^{n+1}. \quad (4)$$

7 Metoda różnic skończonych

Metoda różnic skończonych polega na przybliżaniu pochodnych za pomocą różnic skończonych. Zarówno dziedziną przestrzenną, jak i czasową są dyskretyzowane, czyli dzielone na skończoną liczbę przedziałów, a wartości rozwiązania na końcach tych przedziałów są przybliżane przez rozwiązywanie równań algebraicznych zawierających różnice skończone i wartości z pobliskich punktów.

Wyznaczanie ilorazów różnicowych przy pomocy wzoru Taylora:

$$u(x+h) = u(x) + u'(x)\frac{(h)}{1!} + u''(x)\frac{(h-x)}{2!} + O(h^2)$$

$$u(x) = u(x)$$

$$u(x-h) = u(x) - u'(x)\frac{(h)}{1!} + u''(x)\frac{(-h-x)}{2!} + O(h^2)$$

Przybliżenie pierwszej pochodnej:

$$\begin{aligned}
u'(x) &= Au(x+h) + Bu(x) + Cu(x-h) \\
&= A(u(x) + hu'(x) + \frac{1}{2}u''(x)h^2) + Bu(x) + C(u(x) - hu'(x) + \frac{1}{2}u''(x)h^2)
\end{aligned}$$

$$\begin{aligned}
u(x) : \quad A + B + C &= 0 \\
u'(x) : \quad Ah - Ch &= 1 \\
u''(x) : \quad \frac{1}{2}h^2A + \frac{1}{2}h^2C &= 0
\end{aligned}$$

$$A = \frac{1}{2h}, \quad B = 0, \quad C = -\frac{1}{2h}$$

$$u'(x) \approx \frac{1}{2h}u(x+h) + 0u(x) - \frac{1}{2h}u(x-h) = \frac{u(x+h) - u(x-h)}{2h}$$

Przybliżenie drugiej pochodnej:

$$u''(x) = Au(x+h) + Bu(x) + Cu(x-h)$$

$$\begin{aligned}
u(x) : \quad A + B + C &= 0 \\
u'(x) : \quad Ah - Ch &= 0 \\
u''(x) : \quad \frac{1}{2}h^2A + \frac{1}{2}h^2C &= 1
\end{aligned}$$

$$A = \frac{1}{h^2}, \quad B = -\frac{2}{h^2}, \quad C = \frac{1}{h^2}$$

$$u''(x) \approx \frac{1}{h^2}u(x+h) - \frac{2}{h^2}u(x) + \frac{1}{h^2}u(x-h) = \frac{u(x+h) - 2u(x) + u(x-h)}{h^2}$$

8 Metoda Eulera

Metoda Eulera jest najprostszą metodą rozwiązywania równań różniczkowych zwyczajnych, opartą na wzorze Taylora. Opisuje ją wzór:

$$x(t+h) \approx x(t) + hf(t, x) \quad (5)$$

Jej zaletą jest to, że nie trzeba różniczkować funkcji f , natomiast wadą jest konieczność wyboru bardzo małego kroku h .

9 Metoda Rungego-Kutty

Rozważmy ogólne równanie różniczkowe zwyczajne pierwszego rzędu:

$$y' = f(t, y) \quad (6)$$

z warunkiem początkowym $y(t_0) = y_0$.

Algorytm

Metoda Rungego-Kutty drugiego stopnia jest jedną z metod numerycznych stosowanych do rozwiązywania równań różniczkowych zwyczajnych (ODE). Jest to rozwinięcie metody Eulera, które zwiększa dokładność przez uwzględnienie dodatkowych informacji o nachyleniu funkcji.

1. Krok początkowy:

$$y_0 = y(t_0) \quad (7)$$

2. Obliczenie przyrostów:

$$k_1 = hf(t_n, y_n) \quad (8)$$

$$k_2 = hf\left(t_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right) \quad (9)$$

3. Obliczenie następnej wartości y :

$$y_{n+1} = y_n + k_2 \quad (10)$$

4. Aktualizacja czasu:

$$t_{n+1} = t_n + h \quad (11)$$

10 Rozwiązanie równania Burgersa

Rozważamy cztery równania różniczkowe cząstkowe.

Równanie transportu

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = 0, 0 \leq x \leq 1, t > 0 \quad (12)$$

Nielepkie równanie Burgersa

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = 0, 0 \leq x \leq 1, t > 0 \quad (13)$$

Równanie ciepła

$$\frac{\partial u}{\partial t} - \beta \frac{\partial^2 u}{\partial x^2} = 0, 0 \leq x \leq 1, t > 0, \beta > 0 \quad (14)$$

Równanie Burgersa (nieliniowe, paraboliczne)

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} - \beta \frac{\partial^2 u}{\partial x^2} = 0, 0 \leq x \leq 1, t > 0, \beta > 0 \quad (15)$$

gdzie zmienne i parametry zostały opisane w rozdziale 2, pod równaniem (1).

Dla przypomnienia, β jest współczynnikiem lepkości.

Naszym celem jest wykorzystanie metod różnic skończonych w celu znalezienia przybliżonych

wartości $u(x, t)$ rozwiązania wszystkich równań.

Najpierw określmy krok przestrzenny h oraz krok czasowy k . Niech

$$\begin{aligned} u_{i,j} &= u(x_i, t_j) = u(i \cdot h, j \cdot k), \\ h &= 0.1, k = 0.005, i \in [0, M-1], j \in [0, N-1], M, N \in \mathbb{N} \end{aligned} \quad (16)$$

Wtedy niech warunek początkowy wszystkich równań (funkcja Gaussa)

$$u_{i,0} = e^{-(x_i - x_{mid})^2}, x_{mid} = \frac{(M-1) \cdot h}{2} = 5 \quad (17)$$

Oraz niech warunki brzegowe wszystkich równań

$$u_{0,j} = u_{M-1,j} = 0 \tag{18}$$

Niech

$$\beta < \frac{h^2}{2k} \wedge \beta = 0.99 \tag{19}$$

w celu uzyskania stabilności rozwiązań.

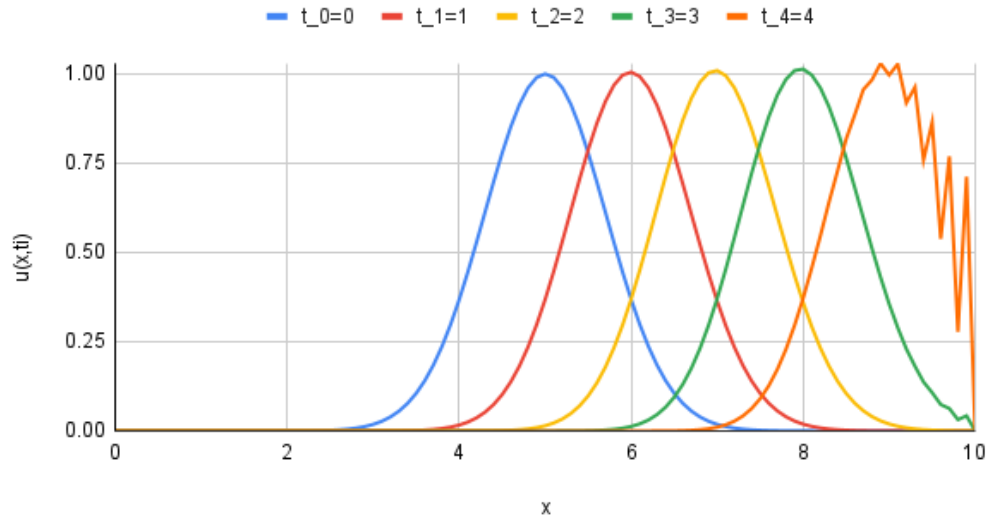
Wtedy stosując metodę Eulera:

dla równania transportu

$$\begin{aligned}\frac{1}{k}(u_{i,j+1} - u_{i,j}) + \frac{1}{2h}(u_{i+1,j} - u_{i-1,j}) &= 0 \\ u_{i,j+1} &= u_{i,j} - \frac{k}{2h}(u_{i+1,j} - u_{i-1,j})\end{aligned}\tag{20}$$

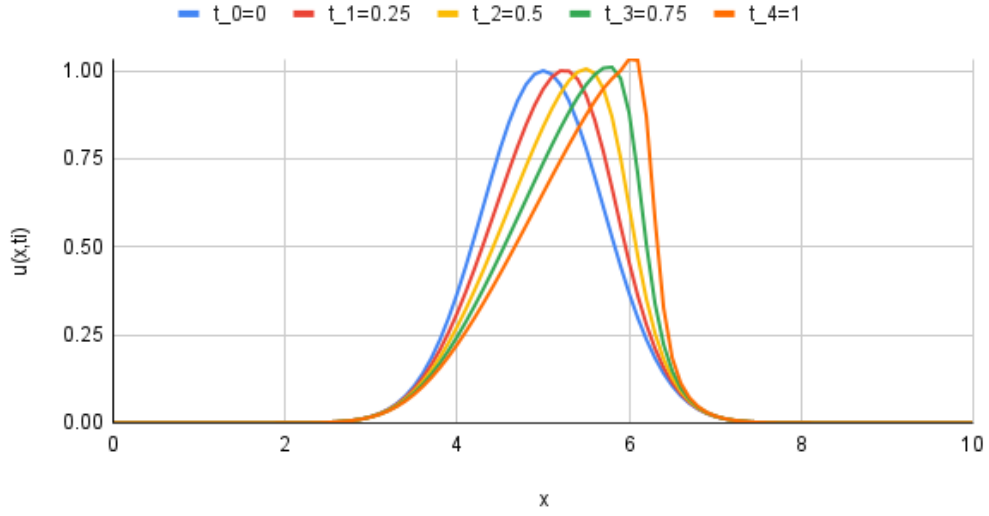
dla nieleckiego równania Burgersa

Rysunek 1: Równanie transportu, metoda Eulera



$$\begin{aligned}\frac{1}{k}(u_{i,j+1} - u_{i,j}) + \frac{1}{4h}(u_{i+1,j}^2 - u_{i-1,j}^2) &= 0 \\ u_{i,j+1} &= u_{i,j} - \frac{k}{4h}(u_{i+1,j}^2 - u_{i-1,j}^2)\end{aligned}\tag{21}$$

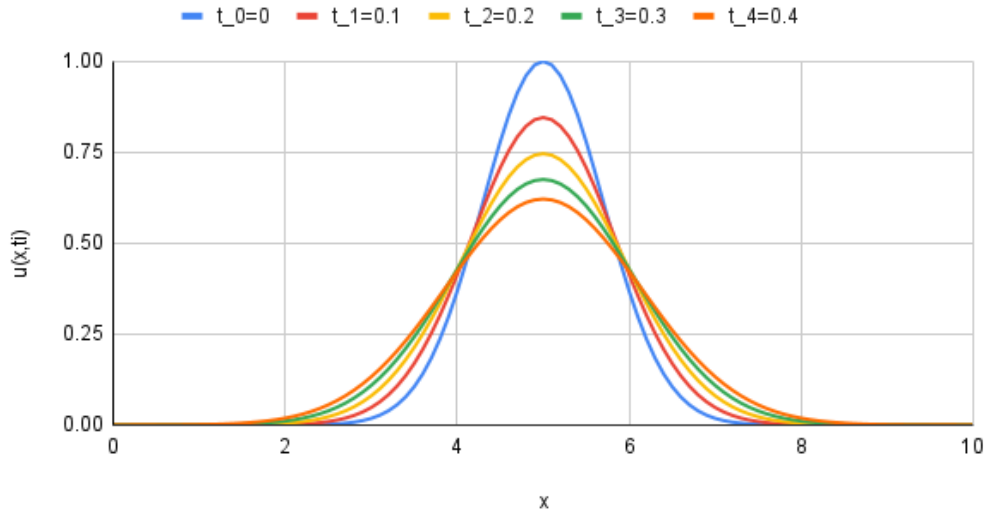
Rysunek 2: Nielepkie równanie Burgersa, metoda Eulera



dla równania ciepła

$$\begin{aligned} \frac{1}{k}(u_{i,j+1} - u_{i,j}) - \frac{\beta}{h^2}(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) &= 0 \\ u_{i,j+1} &= u_{i,j} + \frac{\beta k}{h^2}(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) \end{aligned} \quad (22)$$

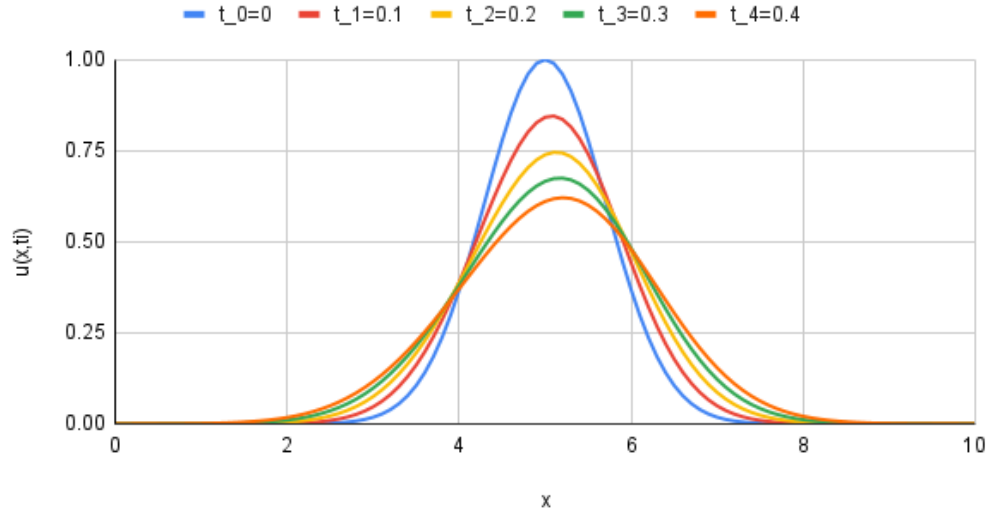
Rysunek 3: Równanie ciepła, metoda Eulera



dla równania Burgersa

$$\begin{aligned} \frac{1}{k}(u_{i,j+1} - u_{i,j}) + \frac{1}{4h}(u_{i+1,j}^2 - u_{i-1,j}^2) - \frac{\beta}{h^2}(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) &= 0 \\ u_{i,j+1} &= u_{i,j} - \frac{k}{4h}(u_{i+1,j}^2 - u_{i-1,j}^2) + \frac{\beta k}{h^2}(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) \end{aligned} \quad (23)$$

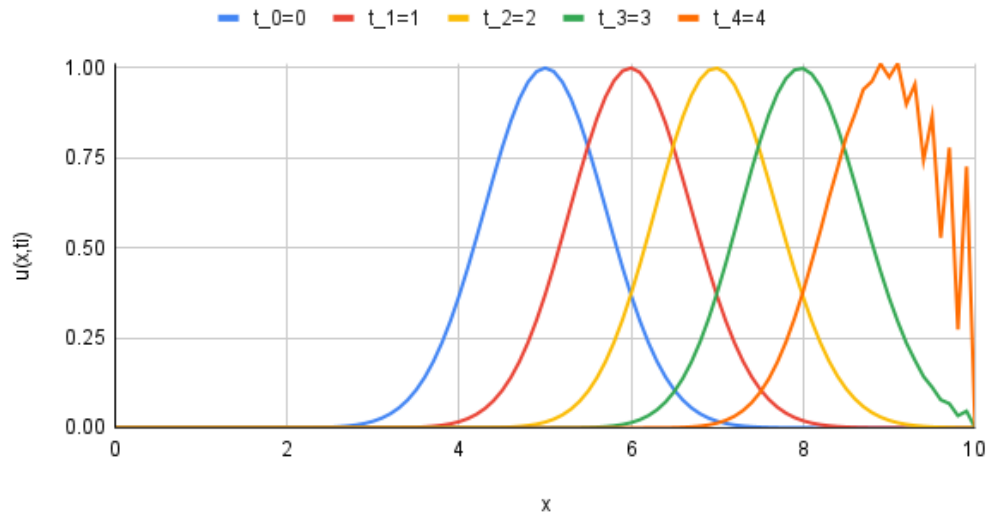
Rysunek 4: Równanie Burgersa, metoda Euler



Stosując metodę Rungego-Kutty drugiego rzędu:
dla równania transportu

$$\begin{aligned} v_{i,j+1} &= u_{i,j} - \frac{k}{4h}(u_{i+1,j} - u_{i-1,j}) \\ u_{i,j+1} &= u_{i,j} - \frac{k}{2h}(v_{i+1,j+1} - v_{i-1,j+1}) \end{aligned} \quad (24)$$

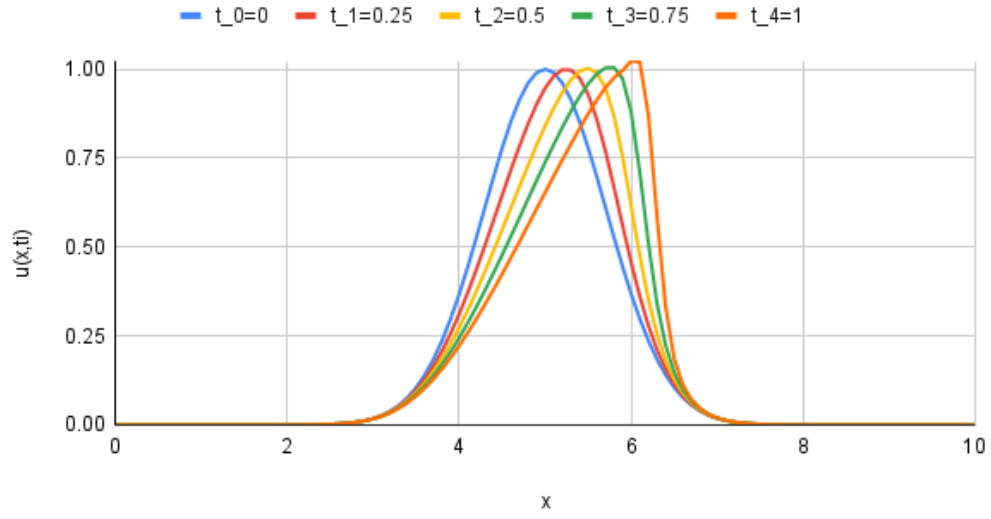
Rysunek 5: Równanie transportu, metoda RK2



dla nielepkiego równania Burgersa

$$\begin{aligned} v_{i,j+1} &= u_{i,j} - \frac{k}{8h}(u_{i+1,j}^2 - u_{i-1,j}^2) \\ u_{i,j+1} &= u_{i,j} - \frac{k}{4h}(v_{i+1,j+1}^2 - v_{i-1,j+1}^2) \end{aligned} \quad (25)$$

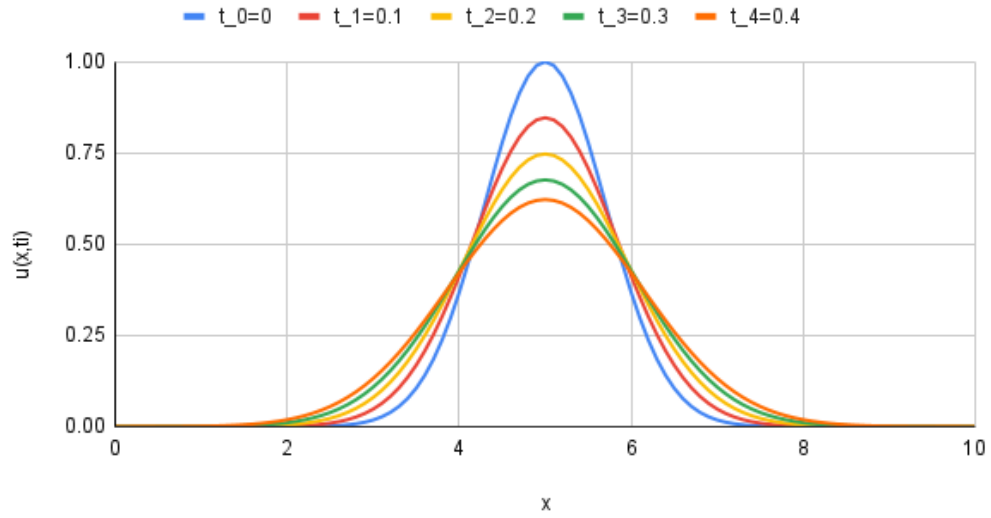
Rysunek 6: Nielepkie równanie Burgersa, metoda RK2



dla równania ciepła

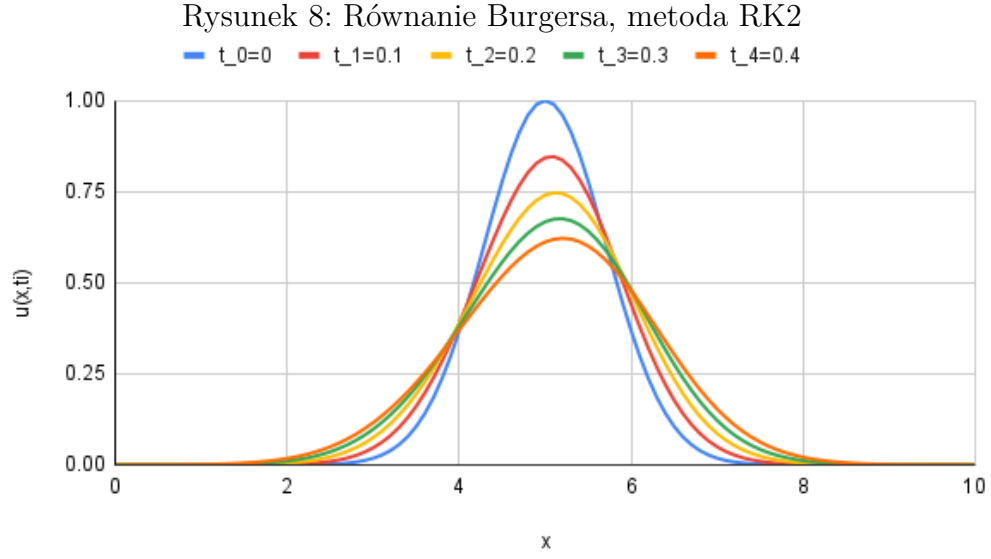
$$\begin{aligned} v_{i,j+1} &= u_{i,j} + \frac{\beta k}{2h^2}(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) \\ u_{i,j+1} &= u_{i,j} + \frac{\beta k}{h^2}(v_{i+1,j+1} - 2v_{i,j+1} + v_{i-1,j+1}) \end{aligned} \quad (26)$$

Rysunek 7: Równanie ciepła, metoda RK2



dla równania Burgersa

$$\begin{aligned} v_{i,j+1} &= u_{i,j} - \frac{k}{8h}(u_{i+1,j}^2 - u_{i-1,j}^2) + \frac{\beta k}{2h^2}(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) \\ u_{i,j+1} &= u_{i,j} - \frac{k}{4h}(v_{i+1,j+1}^2 - v_{i-1,j+1}^2) + \frac{\beta k}{h^2}(v_{i+1,j+1} - 2v_{i,j+1} + v_{i-1,j+1}) \end{aligned} \quad (27)$$



Stosując metodę niejawną:
dla równania ciepła

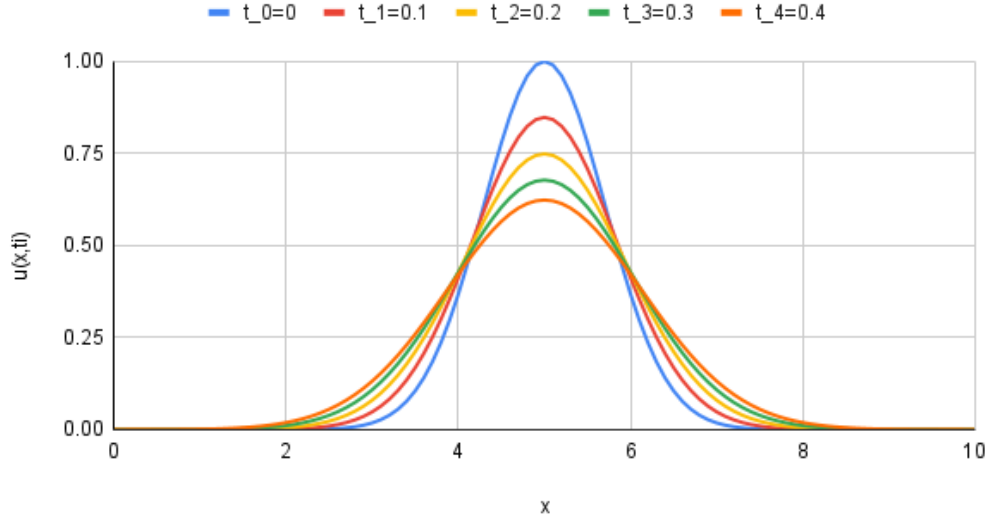
$$\frac{1}{k}(u_{i,j} - u_{i,j-1}) - \frac{\beta}{h^2}(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) = 0 \quad (28)$$

$$u_{i,j-1} = -\frac{\beta k}{h^2}u_{i-1,j} + (1 + 2\frac{\beta k}{h^2})u_{i,j} - \frac{\beta k}{h^2}u_{i+1,j}$$

$$A = \begin{bmatrix} 1+2s & -s & 0 & \cdots & 0 \\ -s & 1+2s & -s & \cdots & 0 \\ 0 & -s & 1+2s & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & 1+2s \end{bmatrix}, s = \frac{\beta k}{h^2}$$

$$u_{i,j} = A^{-1}u_{i,j-1} \quad (29)$$

Rysunek 9: Równanie ciepła, metoda niejawna



dla równania Burgersa

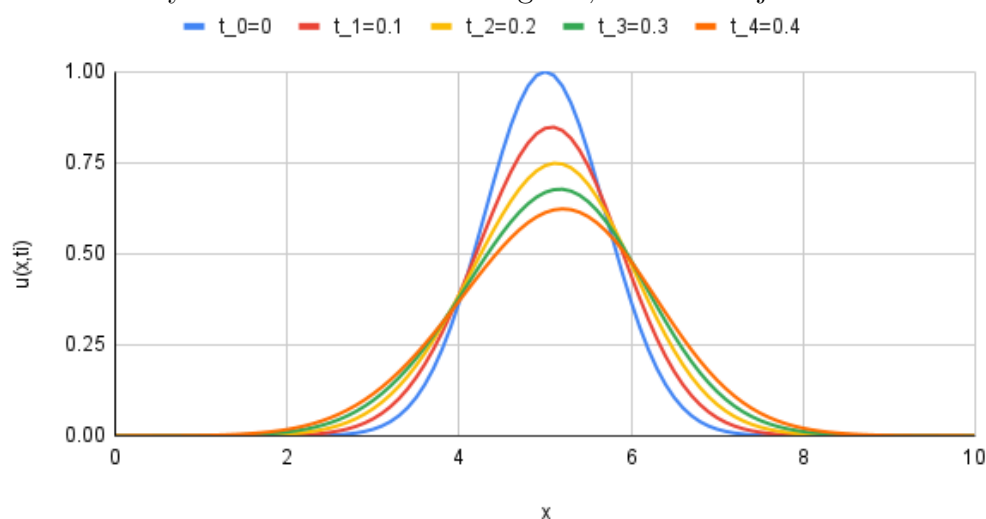
$$\begin{aligned} \frac{1}{k}(u_{i,j} - u_{i,j-1}) + \frac{1}{4h}(u_{i+1,j-1}^2 - u_{i-1,j-1}^2) - \frac{\beta}{h^2}(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) &= 0 \\ u_{i,j-1} - \frac{k}{4h}(u_{i+1,j-1}^2 - u_{i-1,j-1}^2) &= -\frac{\beta k}{h^2}u_{i-1,j} + (1 + 2\frac{\beta k}{h^2})u_{i,j} - \frac{\beta k}{h^2}u_{i+1,j} \end{aligned} \quad (30)$$

$$A = \begin{bmatrix} 1+2s & -s & 0 & \cdots & 0 \\ -s & 1+2s & -s & \cdots & 0 \\ 0 & -s & 1+2s & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & 1+2s \end{bmatrix}, s = \frac{\beta k}{h^2}$$

$$u_{i,j} = A^{-1}(u_{i,j-1} - \frac{k}{4h}(u_{i+1,j-1}^2 - u_{i-1,j-1}^2)) \quad (31)$$

Algorytm, który realizuje opisaną metodę niejawną korzysta z procedury tri rozwiązującej układ równań o macierzy trójkątnej. Metoda tri korzysta ze szczególnego przypadku eliminacji Gaussa w czasie $O(n)$, zamiast czasu $O(n^3)$ dla normalnego przypadku, co ogromnie przyspiesza znajdowanie rozwiązań.

Rysunek 10: Równanie Burgersa, metoda niejawna



Uzyskaliśmy prawie identyczne wyniki za pomocą użycia metody Eulera, RK2 i niejawnej. Nie mogliśmy skorzystać z metody niejawnej dla równania transportu, ani dla nielepkiego równania Burgersa, ponieważ wymaga ona użycia macierzy kwadratowej ($M=N$), a dla tych równań ta równość nie zachodziła.

Dla równania transportu oraz nielepkiego równania Burgersa, jeśli t przekroczy odpowiednią wartość, to przybliżone wartości u zaczynają bardzo mocno odbiegać od rzeczywistych (dla równania transportu $t > 3$, dla nielepkiego równania Burgersa, $t > 1$).

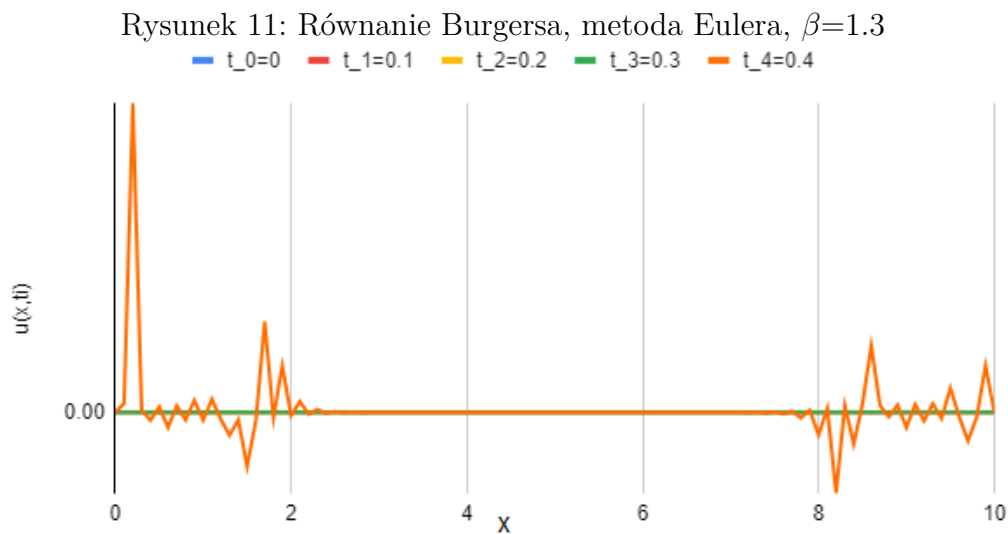
Dla równania transportu, kolejne przesunięcia w czasie powodują uzyskanie maksymalnej wartości u w kolejnych przesunięciach w przestrzeni. Funkcja Gaussa zostaje jedynie przesunięta w prawo. Zobrazowana zostaje liniowość.

Dla nielepkiego równania Burgera, kolejne przesunięcia w czasie powodują uzyskanie maksymalnej wartości u w kolejnych przesunięciach w przestrzeni. Funkcja Gaussa nie zostaje jedynie przesunięta w prawo, tylko wolniej rośnie przed osiągnięciem kresu i szybciej opada po osiągnięciu kresu. Zobrazowana zostaje nieliniowość.

Dla równania ciepła, kolejne przesunięcia w czasie powodują zmniejszenie kresu funkcji Gaussa. Maksymalna wartość u maleje w tym samym punkcie w przestrzeni (5) dla kolejnych czasów.

Dla równania Burgersa, kolejne przesunięcia w czasie powodują zmniejszenie kresu funkcji Gaussa oraz powodują przesunięcie w prawo. Maksymalna wartość u maleje i przesuwa się w przestrzeni dla kolejnych czasów.

Zwiększanie współczynnika β powoduje zmniejszenie kresu $u(x, t)$. Dzieje się tak do około $\beta = 1.2$, po czym rozwiązanie staje się całkowicie niepoprawne.



Program komputerowy w języku C++ wyznaczający powyższe rozwiązania został dołączony w Dodatku A.

11 Opis programu transformaty Hopf Cole'a

11.1 transformataHopfCole.h

Plik nagłówkowy zawiera definicje metod klasy transformaty. Na początku zostały dołączone wszystkie klasy, które są potrzebne do działania programu, jak pokazuje listing 1.

```

1  #pragma once
2  #include <vector>
3  #include <cmath>
4  #include <algorithm>
5  #include <conio.h>

```

```
6      #include <math.h>;
```

Listing 1: początek pliku transformataHopfCole.h

Następnie zostały napisana dyrektywa, która pozwala na używanie wszystkich nazw z przestrzeni nazw "std" bez konieczności ich niefiksowania. Przestrzeń nazw definiuje standardowych klas i funkcji z biblioteki standardowej w C++, takich jak "cout", "vector", "string", co skraca trochę kod. Potem zostały zdefiniowane stała Pi oraz e, jeśli nie są szybciej zdefiniowane, co przedstawia listing ??.

```
7      using namespace std;
8      #ifndef M_PI
9      #define M_PI 3.14159265358979323846
10     #endif
11
12     #ifndef M_E
13     #define M_E 2.71828182845904523536
14     #endif
```

Listing 2: początek pliku definiowanieStałych

Potem została zdefiniowana klasa, w której są wszystkie metody oraz stałe potrzebne do pracy programu.

```
1      class transformataHopfCole {
2          const int N = 11;          // liczba punktów siatki
3          const double k = 0.005, h = 0.1;
4          const double n = ((0.5 * pow(h, 2)) / k) - 0.01;
5          const double r = n*k / pow(h, 2);
6          double t = 0;
7
8          vector<double>theta;
9          vector<double>mu;
10         vector<double> inicjacja_u(); // wzor 4 z pracy
11         vector<double>inicjacjaThetaX0(); // wzor 8 z pracy
12         vector<double>liczenieThetaOdCzasu();
13
14         vector<double>liczenieMu(vector<double>newMu); // wzor 15
15         void zapisDoPliku(vector<double> Theta, double newT);
16         public:
17         void wynikiKoncowe();
```

18 };

Listing 3: klasa transformataHopfCofe

Pierwsza stała oznacza liczbę punktów, "k" odpowiada za krok czasowy, natomiast "h" za krok przestrzenny w dyskretyzacji przestrzennej.

W 8 i 9 linii zostały inicjowanie vectoru, gdzie zostaniom zapisanie wyniki programu. W kolejnych trzech linach zostały już zdefiniowanie klasy, które krok po kroku będą obliczały wynik końcowy oraz w linii 14.

Przedostatnia prywatna metoda służy, by otrzymanie wyniki zostały zapisanie do pliku. Ostatnią metodą jest "wynikiKoncowe()", która jest publiczna.

11.2 transformataHopfCole.cpp

W kolejnym pliku znajdują się ciała metod klasy. Na początku pliku zostały załączone biblioteki oraz własna klasa do prawidłowego działania programu.

```
1 #include "transformataHopfCole.h"
2 #include <iostream>
3 #include <fstream>
4 #include <string>
```

Listing 4: początek pliku transformataHopfCole.cpp

Następne jest ciało metody inicjacjaThetaX0 co przedstawia listing 5. Ta metoda ma na celu inicjacji θ dla czasu zerowego. Pętla służy do wygenerowania dla wszystkich θ wartości początkowych, z których potem zostaną wykorzystane do dalszych obliczeń. Pod koniec metody zostanie wywołanie kolejnej metody, która zapisze wartości do pliku tekstowego oraz zwrócenie tych wartości do vectora.

```
1     vector<double> transformataHopfCole::inicjacjaThetaX0() {
2         vector<double> newTheta;
3         for (int i = 0; i < N; ++i) {
4             double potega = (-1 / (2 * M_PI * n)) * (1 - cos(M_PI * (i * h)));
5             newTheta.push_back(exp(potega));
6         }
7         zapisDoPliku(liczenieMu(newTheta), 0);
8         return newTheta;
9     }
```

Listing 5: ciało metody inicjacjaThetaX0

Kolejną metodą jest "liczenieThetaOdCzasu"(listing 6). Metoda ma na celu policzenie θ dla czasu. Pierwsza pętla służy do policzenia "t", czyli kroku czasowego. Potem jest zdefiniowany nowy vector, w którym zostaną zapisane nowe wartości. Druga pętla służy do policzenia dla θ od 1 do N-1. Po drugiej pętli zostaną nadpisane wartości dla thetaDlaCzasu nowymi wartościami dla konkretnego czasu. Na końcu jest wywołania metoda, która otrzymane wyniki zapisze do pliku. Na samym końcu zostanie zwrócony vector.

```

1 vector<double> transformataHopfCole::liczenieThetaOdCzasu() {
2     vector<double>thetaDlaCzasu(N);
3     thetaDlaCzasu = theta;
4     for (int j = 1; j < 10; ++j) {
5         t = j * k;
6         vector<double>v(N);
7         v[0] = (1 - 2 * r) * thetaDlaCzasu[0] + 2 * r * thetaDlaCzasu[1]; //
            warunki brzegowe
8         v[N - 1] = 2 * r * thetaDlaCzasu[N - 2] + (1 - 2 * r) * thetaDlaCzasu[N
            - 1];
9         for (int i = 1; i < (N - 1); ++i) {
10             double wynik = r * thetaDlaCzasu[i - 1] + (1 - 2 * r) * thetaDlaCzasu
                [i] + r * thetaDlaCzasu[i + 1];
11             v[i] = wynik;
12         }
13         thetaDlaCzasu = v;
14         zapisDoPliku(liczenieMu(thetaDlaCzasu), t);
15
16     }
17     return thetaDlaCzasu;
18 }

```

Listing 6: ciało metody liczenieThetaOdCzasu

Kolejną metodą jest "liczenieMu", która wykorzystuje θ od czasu do policzenia $u(x, t)$. Na wyraz 0 i (N-1) jest przypisana wartość 0, co wynika z warunki początkowego i końcowego. Kolejne wartości są liczone w pętli for, jak pokazuje listing 7.

```

1 vector<double> transformataHopfCole::liczenieMu(vector<double> newMu) {
2     vector<double>newMu1(N);
3     newMu1[0] = 0; //warunek początkowy
4     newMu1[N - 1] = 0; //warunek końcowy
5     for (int i = 1; i < (N - 1); ++i) {

```

```

6     newMu1[i] = (-(n / h) * ((newMu[i + 1] - newMu[i - 1]) / newMu[i]));
7 }
8
9     return newMu1;
10 }

```

Listing 7: ciało metody liczenieMu

Przedostatnią metodą jest "zapisDoPliku", który ma na celu otrzymanych wyników zapisach do pliku. Na początku jest inicjowany string, który będzie nazwą pliku. Potem program tworzy nowy plik, jeśli nie istnieje, jak istnieje to zawartość zostanie zastąpiona. Potem w warunku jest sprawdzenie czy udało się otworzyć plik, jeśli nie to pojawi się odpowiednia informacja. Pętla for ma na celu zapisanie wszystkich punktów do pliku. Na końcu program zamyka plik, gdy już skończył pracę.

```

1 void transformataHopfCole::zapisDoPliku(vector<double> newMu, double newT)
    {
2     string nazwaPliku = "Wyniki1DlaT" + to_string(newT) + ".txt";
3     std::ofstream plik(nazwaPliku);
4
5     // Sprawdzamy, czy plik został otwarty poprawnie
6     if (!plik) {
7         std::cerr << "Nie można otworzyć pliku!" << std::endl;
8         return;
9     }
10    for (int i = 0; i < size(newMu); ++i) {
11
12        plik << (i * h) << " " << newMu[i] << '\n';
13    }
14
15    // Zamykamy plik
16    plik.close();
17 }

```

Listing 8: metoda zapisDoPliku

Ostaną metodą są "wynikiKoncowe", który na początku wywołuje metodę "inicjacjaThetaX0", następnie "liczenieThetaOdCzasu". Kolejne metody nie są potrzebne by je wywołać, z powodu, że metoda "liczenieThetaOdCzasu" wywołuje metodę która liczy $u(x, t)$ i metodą, która zapisuje dane do pliku.

```

1 void transformataHopfCole::wynikiKonczowe() {
2     theta = inicjacjaThetaX0();
3     theta = liczenieThetaOdCzasu();
4 }

```

Listing 9: metoda wynikiKonczowe

11.3 transformata.cpp

Kolejnym plikiem jest transformata.cpp, który ma funkcję główną main.cpp. Która na początku ma dołączona klasę. Następnie w main tworzy się obiekt klasy transformataHopfCole, a następnie wywołuję się metodę klasy, by program policzył nam wyniki.

```

1 #include "transformataHopfCole.h"
2 int main()
3 {
4     transformataHopfCole Burger;
5     Burger.wynikiKonczowe();
6 }

```

Listing 10: transformata.cpp

12 wyniki

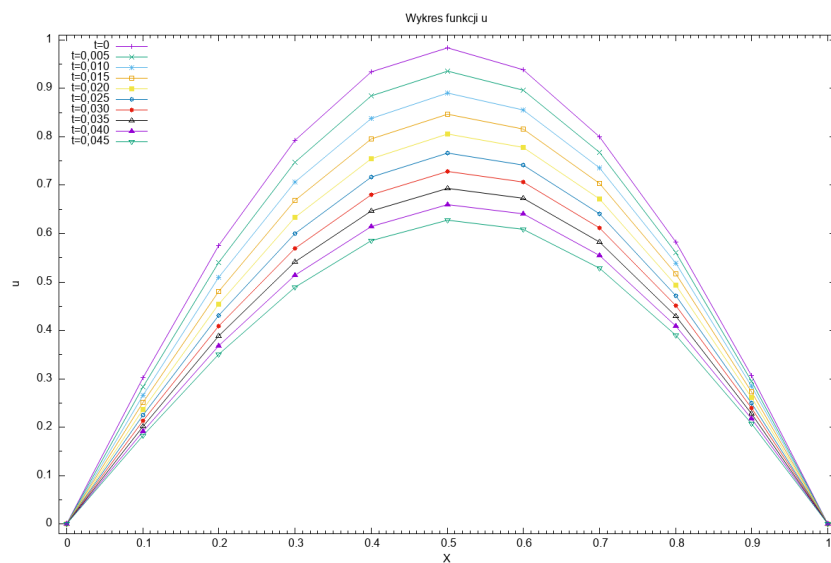
Wygenerowano wykresy dla $k = 0.0005$ i $h = 0.1$. Dla coraz większego czasu można zauważyć, że amplituda rozwiązana maleje, można to zauważyć na rys 12, wyniki punktów wykresu są przedstawione na tabelkach 1 i 2.

Można zauważyć, że jak zmniejszymy k do 0.0005, to można zauważyć, że dla czasu 0 funkcja jest trójkąta. Natomiast dla większego czasu można zauważyć, że funkcja się powoli zaokrągla i przypomina coraz bardziej wykres dla sinusa, można to zauważyć na rys 13.

Na rys 13 można także zobaczyć, jak zachowują się dane dla złych wartości k oraz h , które nie spełniają warunku

$$\frac{k}{h^2} \leq 0,5$$

który musi być spełniony podczas rozprzetywania tego problemu, co także może powodować, że nie ma całego dziedzina x od 0 do 1. Dla $k = 0.05, h = 0.01$ wykres robi się trójkątny



Rysunek 12: $k = 0.005$, $h = 0.1$

$x \backslash t$	0	0.005	0,010	0.015	0.020	0.025
0	0	0	0	0	0	0
0.1	0.301705	0.283068	0.266381	0.251276	0.237487	0.224811
0.2	0.574577	0.540199	0.509139	0.480833	0.454852	0.430863
0.3	0.792316	0.747317	0.706078	0.668073	0.632879	0.600146
0.4	0.933565	0.88415	0.838007	0.794843	0.754384	0.716382
0.5	0.984036	0.93621	0.890555	0.847073	0.805711	0.766389
0.6	0.938116	0.896627	0.856073	0.816677	0.778584	0.741879
0.7	0.799679	0.767504	0.735313	0.703418	0.672066	0.641447
0.8	0.581939	0.560385	0.53838	0.516199	0.494079	0.472219
0.9	0.306254	0.295543	0.284455	0.273144	0.26175	0.250398
1	0	0	0	0	0	0

Tabela 1: Wyniki dla $k=0.005, h=0.1$, dla czasu od 0 do 0.025

x\t	0.030	0.35	0.40	0.45	0.45
0	0	0	0	0	0
0.1	0.213091	0.202201	0.192041	0.182528	0.182528
0.2	0.408602	0.387857	0.368453	0.350247	0.350247
0.3	0.569591	0.540975	0.514101	0.488803	0.488803
0.4	0.680618	0.6469	0.615061	0.584955	0.584955
0.5	0.729015	0.693493	0.659731	0.627636	0.627636
0.6	0.706609	0.672786	0.640407	0.609451	0.609451
0.7	0.611699	0.582919	0.555172	0.528496	0.528496
0.8	0.450776	0.429873	0.409598	0.390013	0.390013
0.9	0.239189	0.228205	0.217507	0.207141	0.207141
1	0	0	0	0	0

Tabela 2: Wyniki dla $k=0.005, h=0.1$, dla czasu od 0.030 do 0.045

oraz niektóre wartości przyjmują wartości ujemne, lub przyjmują wartości zero lub bliskie zero.

13 Podsumowanie

W tej pracy przybliżyliśmy równanie Burgersa oraz jego zastosowania, transformację Hopf-Cole'a, rozwiązanie analityczne oraz metody numeryczne, w tym metodę różnic skończonych, metodę Eulera oraz metodę Rungego-Kutty.

14 dodatek A

```

1 //Utility.h
2
3 #include <cmath>
4 #include <fstream>
5 #include <iostream>
6 #include <string>
7 #include <windows.h>
8 #include <vector>
9
10 class Save
11 {
12 public:
13     //public constructor/s
14     Save(const double h, const double k, const size_t M, const size_t N, const std::
        vector<std::vector<double>>& u, const std::string save_file_name);
15 };
16
17 ///////////////////////////////////////////////////
18
19 class MidPoint
20 {
21 public:
22     //public constructor/s
23     MidPoint(const double h, const size_t M, double& midpoint);
24 };
25
26 ///////////////////////////////////////////////////
27
28 class IniVector
29 {
30 public:
31     //public constructor/s
32     IniVector(const size_t M, std::vector<double>& vec);
33 };
34
35 ///////////////////////////////////////////////////
36
37 class IniMatrix
38 {
39 public:
40     //public constructor/s
41     IniMatrix(const size_t M, const size_t N, std::vector<std::vector<double>>& mat);
42 };
43
44 ///////////////////////////////////////////////////
45
46 class InitialConditions
47 {
48 public:
49     //public constructor/s
50     InitialConditions(const double h, const size_t M, const size_t N, std::vector<std::
        vector<double>>& u, const double midpoint);
51 };

```

```

1 //Utility.cpp
2
3 #include "0.0.Utility.h"
4
5 Save::Save(const double h, const double k, const size_t M, const size_t N, const std::
vector<std::vector<double>>& u, const std::string save_file_name)
6 {
7     std::ofstream out;
8     out.open(save_file_name);
9     if (out.is_open())
10    {
11        for (size_t i = 0; i < M; i++)
12        {
13            for (size_t j = 0; j < N; j++)
14            {
15                out << i * h << "," << j * k << "," << u[i][j] << "\n";
16            }
17        }
18    }
19    else
20    {
21        throw std::string("Could not save to file!");
22    }
23    out.close();
24 }
25
26 ///////////////////////////////////////////////////
27
28 MidPoint::MidPoint(const double h, const size_t M, double& midpoint)
29 {
30     midpoint = ((M - 1) * h) / 2;;
31 }
32
33 ///////////////////////////////////////////////////
34
35 IniVector::IniVector(const size_t M, std::vector<double>& vec)
36 {
37     vec.resize(M);
38     for (size_t i = 0; i < M; i++)
39     {
40         vec[i] = 0;
41     }
42 }
43
44 ///////////////////////////////////////////////////
45
46 IniMatrix::IniMatrix(const size_t M, const size_t N, std::vector<std::vector<double>>&
mat)
47 {
48     mat.resize(M, std::vector<double>(N));
49     for (size_t i = 0; i < M; i++)
50     {
51         for (size_t j = 0; j < N; j++)
52         {
53             mat[i][j] = 0;
54         }
55     }
56 }
57
58 ///////////////////////////////////////////////////
59
60 InitialConditions::InitialConditions(const double h, const size_t M, const size_t N, std
::vector<std::vector<double>>& vec, const double midpoint)
61 {
62     for (size_t i = 0; i < M; i++)

```

```
63     {
64         vec[i][0] = exp(-pow((i * h - midpoint), 2));
65     }
66     for (size_t j = 0; j < N; j++)
67     {
68         vec[0][j] = 0;
69         vec[M - 1][j] = 0;
70     }
71 }
```

```

1  //MatrixOperations.h
2
3  class MatrixCopy
4  {
5  public:
6      MatrixCopy(const size_t M, const std::vector<std::vector<double>>& mat_from, std:::
          vector<std::vector<double>>& mat_to);
7  };
8
9  //////////////////////////////////////
10 //////////////////////////////////////
11 class VectorCopy
12 {
13 public:
14     VectorCopy(const size_t M, const std::vector<double>& vec_from, std::vector<double>&
          vec_to);
15 };

```

```

1  //MatrixOperations.cpp
2
3  #include "0.0.Utility.h"
4  #include "0.1.MatrixOperations.h"
5
6  MatrixCopy::MatrixCopy(const size_t M, const std::vector<std::vector<double>>& mat_from,
7  std::vector<std::vector<double>>& mat_to)
8  {
9      for (size_t i = 0; i < M; i++)
10     {
11         for (size_t j = 0; j < M; j++)
12         {
13             mat_to[i][j] = mat_from[i][j];
14         }
15     }
16
17     //////////////////////////////////////
18     //////////////////////////////////////
19
20     VectorCopy::VectorCopy(const size_t M, const std::vector<double>& vec_from, std::vector<
21     double>& vec_to)
22     {
23         for (size_t i = 0; i < M; i++)
24         {
25             vec_to[i] = vec_from[i];
26         }
27     }

```



```
1 //LUGaussSpecial.h
2
3 class LUGaussSpecial
4 {
5 public:
6     //public constructor/s
7     LUGaussSpecial(const size_t M, const double s_0, const double s_1, const double s_2,
8         std::vector<double>& b, std::vector<double>& x);
9 private:
10    //private function/s
11    void SpecialGaussianElimination(std::vector<double>& b, std::vector<double>& x);
12    void ExtractTriVectors(const double s_0, const double s_1, const double s_2);
13    //private variable/s
14    const size_t M;
15    std::vector<double> a{};
16    std::vector<double> c{};
17    std::vector<double> d{};
18 };
19
```

```

1 //LUGaussSpecial.cpp
2
3 #include "0.0.Utility.h"
4 #include "0.1.MatrixOperations.h"
5 #include "0.2.LUGaussSpecial.h"
6
7 //public constructor/s
8 LUGaussSpecial::LUGaussSpecial(const size_t M, const double s_0, const double s_1, const
double s_2, std::vector<double>& b, std::vector<double>& x)
9     : M(M)
10 {
11     IniVector(M, a);
12     IniVector(M, c);
13     IniVector(M, d);
14
15     ExtractTriVectors(s_0, s_1, s_2);
16     SpecialGaussianElimination(b, x);
17 }
18 //private function/s
19 void LUGaussSpecial::SpecialGaussianElimination(std::vector<double>& b, std::vector<
double>& x)
20 {
21     for (size_t i = 1; i < M; i++)
22     {
23         d[i] = d[i] - (a[i - 1] / d[i - 1]) * c[i - 1];
24         b[i] = b[i] - (a[i - 1] / d[i - 1]) * b[i - 1];
25     }
26     x[M - 1] = b[M - 1] / d[M - 1];
27     for (int i = M - 2; i >= 0; i--)
28     {
29         x[i] = (b[i] - (c[i] * x[i + 1])) / d[i];
30     }
31 }
32 void LUGaussSpecial::ExtractTriVectors(const double s_0, const double s_1, const double
s_2)
33 {
34     for (size_t i = 0; i < M; i++)
35     {
36         a[i] = s_0;
37         d[i] = s_1;
38         c[i] = s_2;
39     }
40 }

```

```

1  //Tri.h
2
3  class Tri
4  {
5  public:
6      //public constructor/s
7      Tri(double s_0, double s_1, double s_2, size_t M, size_t column, std::vector<double>&
        b, std::vector<std::vector<double>>& mat);
8  private:
9      //private function/s
10     void ExecuteTri(const double s_0, const double s_1, const double s_2, const size_t
        column, std::vector<double>& b, std::vector<std::vector<double>>& mat);
11     void AssignResults(const size_t column, std::vector<std::vector<double>>& mat);
12     //private variable/s
13     const size_t M;
14     std::vector<double> x{};
15 };

```

```

1  //Tri.cpp
2
3  #include "0.0.Utility.h"
4  #include "0.2.LUGaussSpecial.h"
5  #include "0.3.Tri.h"
6
7  //public constructor/s
8  Tri::Tri(double s_0, double s_1, double s_2, size_t M, size_t column, std::vector<double>
>& b, std::vector<std::vector<double>>& mat)
9      : M(M)
10 {
11     IniVector(M, x);
12
13     ExecuteTri(s_0, s_1, s_2, column, b, mat);
14 }
15 //private function/s
16 void Tri::ExecuteTri(const double s_0, const double s_1, const double s_2, const size_t
column, std::vector<double>& b, std::vector<std::vector<double>>& mat)
17 {
18     LUGaussSpecial solve(M, s_0, s_1, s_2, b, x);
19     AssignResults(column, mat);
20 }
21 void Tri::AssignResults(const size_t column, std::vector<std::vector<double>>& mat)
22 {
23     for (size_t i = 1; i < M - 1; i++)
24     {
25         mat[i][column + 1] = x[i];
26     }
27 }

```

```

1 //Eq1.h
2
3 #pragma once
4
5 //par_u/par_t + par_u/par_x = 0 differential equation using Euler + Runge-Kutta 2nd
  order methods
6
7 class ParDiffEq1Euler
8 {
9 public:
10     //public constructor/s
11     ParDiffEq1Euler(double h, double k);
12     //public function/s
13     void Euler();
14 private:
15     //private variable/s
16     double midpoint{};
17     const size_t M{ 101 };
18     const size_t N{ 1001 };
19     std::vector<std::vector<double>> u{};
20
21     const double h;
22     const double k;
23 };
24
25 class ParDiffEq1RK2
26 {
27 public:
28     //public constructor/s
29     ParDiffEq1RK2(double h, double k);
30     //public function/s
31     void RK2();
32 private:
33     //private variable/s
34     double midpoint{};
35     const size_t M{ 101 };
36     const size_t N{ 1001 };
37     std::vector<std::vector<double>> u{};
38     std::vector<std::vector<double>> v{};
39
40     const double h;
41     const double k;
42 };

```

```

1 //Eq1.cpp
2
3 #include "0.0.Utility.h"
4 #include "1.Eq1.h"
5
6 //public constructor/s
7 ParDiffEq1Euler::ParDiffEq1Euler(double h, double k)
8     : h(h), k(k)
9 {
10     MidPoint(h, M, midpoint);
11     IniMatrix(M, N, u);
12     InitialConditions(h, M, N, u, midpoint);
13 }
14 //public function/s
15 void ParDiffEq1Euler::Euler()
16 {
17     const double s{ k / (2 * h) };
18
19     for (size_t j = 0; j < N - 1; j++)
20     {
21         for (size_t i = 1; i < M - 1; i++)
22         {
23             u[i][j + 1] = u[i][j] - s * (u[i + 1][j] - u[i - 1][j]);
24         }
25     }
26
27     std::string save_file_name{ "Results/Eq1/Euler_Eq_1_h_" + std::to_string(h) + "_k_" +
28         std::to_string(k) + ".csv" };
29
30     Save(h, k, M, N, u, save_file_name);
31 }
32
33 ///////////////////////////////////////////////////
34 //public constructor/s
35 ParDiffEq1RK2::ParDiffEq1RK2(double h, double k)
36     : h(h), k(k)
37 {
38     MidPoint(h, M, midpoint);
39     IniMatrix(M, N, u);
40     IniMatrix(M, N, v);
41     InitialConditions(h, M, N, u, midpoint);
42     InitialConditions(h, M, N, v, midpoint);
43 }
44 //public function/s
45 void ParDiffEq1RK2::RK2()
46 {
47     const double r{ k / (4 * h) };
48     const double s{ k / (2 * h) };
49
50     for (size_t j = 0; j < N - 1; j++)
51     {
52         for (size_t i = 1; i < M - 1; i++)
53         {
54             v[i][j + 1] = u[i][j] - r * (u[i + 1][j] - u[i - 1][j]);
55         }
56         for (size_t i = 1; i < M - 1; i++)
57         {
58             u[i][j + 1] = u[i][j] - s * (v[i + 1][j + 1] - v[i - 1][j + 1]);
59         }
60     }
61
62     std::string save_file_name{ "Results/Eq1/RK2_Eq_1_h_" + std::to_string(h) + "_k_" +
63         std::to_string(k) + ".csv" };
64
65     Save(h, k, M, N, u, save_file_name);
66 }

```

```

1 //Eq2.h
2
3 //par_u/par_t + u*par_u/par_x = 0 differential equation using Euler + Runge-Kutta 2nd
  order methods
4
5 class ParDiffEq2Euler
6 {
7 public:
8     //public constructor/s
9     ParDiffEq2Euler(double h, double k);
10    //public function/s
11    void Euler();
12 private:
13    //private variable/s
14    double midpoint{};
15    const size_t M{ 101 };
16    const size_t N{ 1001 };
17    std::vector<std::vector<double>> u{};
18
19    const double h;
20    const double k;
21 };
22
23 class ParDiffEq2RK2
24 {
25 public:
26    //public constructor/s
27    ParDiffEq2RK2(double h, double k);
28    //public function/s
29    void RK2();
30 private:
31    //private variable/s
32    double midpoint{};
33    const size_t M{ 101 };
34    const size_t N{ 1001 };
35    std::vector<std::vector<double>> u{};
36    std::vector<std::vector<double>> v{};
37
38    const double h;
39    const double k;
40 };

```

```

1 //Eq2.cpp
2
3 #include "0.0.Utility.h"
4 #include "2.Eq2.h"
5
6 //public constructor/s
7 ParDiffEq2Euler::ParDiffEq2Euler(double h, double k)
8     : h(h), k(k)
9 {
10     MidPoint(h, M, midpoint);
11     IniMatrix(M, N, u);
12     InitialConditions(h, M, N, u, midpoint);
13 }
14 //public function/s
15 void ParDiffEq2Euler::Euler()
16 {
17     const double s{ k / (4 * h) };
18
19     for (size_t j = 0; j < N - 1; j++)
20     {
21         for (size_t i = 1; i < M - 1; i++)
22         {
23             u[i][j + 1] = u[i][j] - s * (pow(u[i + 1][j], 2) - pow(u[i - 1][j], 2));
24         }
25     }
26
27     std::string save_file_name{ "Results/Eq2/Euler_Eq_2_h_" + std::to_string(h) + "_k_" +
28         std::to_string(k) + ".csv" };
29
30     Save(h, k, M, N, u, save_file_name);
31 }
32
33 ///////////////////////////////////////////////////
34 //public constructor/s
35 ParDiffEq2RK2::ParDiffEq2RK2(double h, double k)
36     : h(h), k(k)
37 {
38     MidPoint ini_midpoint(h, M, midpoint);
39     IniMatrix(M, N, u);
40     IniMatrix(M, N, v);
41     InitialConditions ini_initial_conditions_u(h, M, N, u, midpoint);
42     InitialConditions ini_initial_conditions_v(h, M, N, v, midpoint);
43 }
44 //public function/s
45 void ParDiffEq2RK2::RK2()
46 {
47     const double r{ k / (8 * h) };
48     const double s{ k / (4 * h) };
49
50     for (size_t j = 0; j < N - 1; j++)
51     {
52         for (size_t i = 1; i < M - 1; i++)
53         {
54             v[i][j + 1] = u[i][j] - r * (pow(u[i + 1][j], 2) - pow(u[i - 1][j], 2));
55         }
56         for (size_t i = 1; i < M - 1; i++)
57         {
58             u[i][j + 1] = u[i][j] - s * (pow(v[i + 1][j + 1], 2) - pow(v[i - 1][j + 1], 2));
59         }
60     }
61
62     std::string save_file_name{ "Results/Eq2/RK2_Eq_2_h_" + std::to_string(h) + "_k_" +
63         std::to_string(k) + ".csv" };
64
65     Save(h, k, M, N, u, save_file_name);
66 }

```



```

1 //Eq3.h
2
3 //par_u/par_t - beta*par_u^2/par_x^2 = 0 differential equation using Euler + Runge-Kutta
  2nd order + implicit methods
4
5 class ParDiffEq3Euler
6 {
7 public:
8     //public constructor/s
9     ParDiffEq3Euler(double h, double k);
10    //public function/s
11    void Euler();
12 private:
13    //private variable/s
14    double midpoint{};
15    const size_t M{ 101 };
16    const size_t N{ 101 };
17    std::vector<std::vector<double>> u{};
18
19    double beta;
20    const double h;
21    const double k;
22 };
23
24 class ParDiffEq3RK2
25 {
26 public:
27     //public constructor/s
28     ParDiffEq3RK2(double h, double k);
29     //public function/s
30     void RK2();
31 private:
32     //private variable/s
33     double midpoint{};
34     const size_t M{ 101 };
35     const size_t N{ 101 };
36     std::vector<std::vector<double>> u{};
37     std::vector<std::vector<double>> v{};
38
39     double beta;
40     const double h;
41     const double k;
42 };
43
44 class ParDiffEq3Implicit
45 {
46 public:
47     //public constructor/s
48     ParDiffEq3Implicit(double h, double k);
49     //public function/s
50     void Implicit();
51 private:
52     //private function/s
53     void SetVectorb(const size_t column);
54     //private variable/s
55     double midpoint{};
56     const size_t M{ 101 };
57     std::vector<std::vector<double>> u{};
58     std::vector<double> b{};
59
60     double beta;
61     const double h;
62     const double k;
63 };

```

```

1 //Eq3.cpp
2
3 #include "0.0.Utility.h"
4 #include "0.3.Tri.h"
5 #include "3.Eq3.h"
6
7 //public constructor/s
8 ParDiffEq3Euler::ParDiffEq3Euler(double h, double k)
9     : h(h), k(k)
10 {
11     beta = ((0.5 * pow(h, 2)) / k) - 0.01; /*satisfy (beta*k)/h^2 < 0.5 for stability*/
12
13     MidPoint(h, M, midpoint);
14     IniMatrix(M, N, u);
15     InitialConditions(h, M, N, u, midpoint);
16 }
17 //public function/s
18 void ParDiffEq3Euler::Euler()
19 {
20     const double s{ (beta * k) / pow(h,2) };
21
22     for (size_t j = 0; j < N - 1; j++)
23     {
24         for (size_t i = 1; i < M - 1; i++)
25         {
26             u[i][j + 1] = u[i][j] + s * (u[i + 1][j] - 2 * u[i][j] + u[i - 1][j]);
27         }
28     }
29
30     std::string save_file_name{ "Results/Eq3/Euler_Eq_3_h_" + std::to_string(h) + "_k_" +
31         std::to_string(k) + "_beta_" + std::to_string(beta) + ".csv" };
32
33     Save(h, k, M, N, u, save_file_name);
34 }
35
36 ///////////////////////////////////////////////////
37 //public constructor/s
38 ParDiffEq3RK2::ParDiffEq3RK2(double h, double k)
39     : h(h), k(k)
40 {
41     beta = ((0.5 * pow(h, 2)) / k) - 0.01; /*satisfy (beta*k)/h^2 < 0.5 for stability*/
42
43     MidPoint(h, M, midpoint);
44     IniMatrix(M, N, u);
45     IniMatrix(M, N, v);
46     InitialConditions(h, M, N, u, midpoint);
47     InitialConditions(h, M, N, v, midpoint);
48 }
49 //public function/s
50 void ParDiffEq3RK2::RK2()
51 {
52     const double r{ (beta * k) / (2 * pow(h,2)) };
53     const double s{ (beta * k) / pow(h,2) };
54
55     for (size_t j = 0; j < N - 1; j++)
56     {
57         for (size_t i = 1; i < M - 1; i++)
58         {
59             v[i][j + 1] = u[i][j] + r * (u[i + 1][j] - 2 * u[i][j] + u[i - 1][j]);
60         }
61         for (size_t i = 1; i < M - 1; i++)
62         {
63             u[i][j + 1] = u[i][j] + s * (v[i + 1][j + 1] - 2 * v[i][j + 1] + v[i - 1][j + 1]);
64         }
65     }
66 }

```

```

67     std::string save_file_name{ "Results/Eq3/RK2_Eq_3_h_" + std::to_string(h) + "_k_" +
68     std::to_string(k) + "_beta_" + std::to_string(beta) + ".csv" };
69
69     Save(h, k, M, N, u, save_file_name);
70 }
71
72 ///////////////////////////////////////////////////
73
74 //public constructor/s
75 ParDiffEq3Implicit::ParDiffEq3Implicit(double h, double k)
76     : h(h), k(k)
77 {
78     beta = ((0.5 * pow(h, 2)) / k) - 0.01; /*satisfy (beta*k)/h^2 < 0.5 for stability*/
79
80     MidPoint(h, M, midpoint);
81     IniMatrix(M, M, u);
82     IniVector(M, b);
83     InitialConditions(h, M, M, u, midpoint);
84 }
85 //public function/s
86 void ParDiffEq3Implicit::Implicit()
87 {
88     const double s{ (beta * k) / pow(h,2) };
89     const double s_0{ -s };
90     const double s_1{ 1 + 2 * s };
91     const double s_2{ -s };
92
93     for (size_t j = 0; j < M-1; j++)
94     {
95         SetVectorb(j);
96         Tri solve(s_0, s_1, s_2, M, j, b, u);
97     }
98
99     std::string save_file_name{ "Results/Eq3/Implicit_Eq_3_h_" + std::to_string(h) +
100     "_k_" + std::to_string(k) + "_beta_" + std::to_string(beta) + ".csv" };
101
101     Save(h, k, M, M, u, save_file_name);
102 }
103 //private function/s
104 void ParDiffEq3Implicit::SetVectorb(const size_t column)
105 {
106     for (size_t i = 0; i < M; i++)
107     {
108         b[i] = u[i][column];
109     }
110 }

```

```

1 //Burgers.h
2
3 //Burgers' equation:  $\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} - \beta \frac{\partial^2 u}{\partial x^2} = 0$  using
Euler + Runge-Kutta 2nd order + implicit methods
4
5 class ParDiffEqBurgersEuler
6 {
7 public:
8     //public constructor/s
9     ParDiffEqBurgersEuler(double h, double k);
10    //public function/s
11    void Euler();
12 private:
13    //private variable/s
14    double midpoint{};
15    const size_t M{ 101 };
16    const size_t N{ 101 };
17    std::vector<std::vector<double>> u{};
18
19    double beta;
20    double h;
21    double k;
22 };
23
24 class ParDiffEqBurgersRK2
25 {
26 public:
27     //public constructor/s
28     ParDiffEqBurgersRK2(double h, double k);
29    //public function/s
30    void RK2();
31 private:
32    //private variable/s
33    double midpoint{};
34    const size_t M{ 101 };
35    const size_t N{ 101 };
36    std::vector<std::vector<double>> u{};
37    std::vector<std::vector<double>> v{};
38
39    double beta;
40    const double h;
41    const double k;
42 };
43
44 class ParDiffEqBurgersImplicit
45 {
46 public:
47     //public constructor/s
48     ParDiffEqBurgersImplicit(double h, double k);
49    //public function/s
50    void Implicit();
51 private:
52    //private function/s
53    void SetVectorb(const size_t column);
54    //private variable/s
55    double midpoint{};
56    const size_t M{ 101 };
57    std::vector<std::vector<double>> u{};
58    std::vector<double> b{};
59
60    double beta;
61    const double h;
62    const double k;
63 };

```

```

1 //Burgers.cpp
2
3 #include "0.0.Utility.h"
4 #include "0.3.Tri.h"
5 #include "4.Burgers.h"
6
7 //public constructor/s
8 ParDiffEqBurgersEuler::ParDiffEqBurgersEuler(double h, double k)
9 : h(h), k(k)
10 {
11     beta = ((0.5 * pow(h, 2)) / k) - 0.01; /*satisfy (beta*k)/h^2 < 0.5 for stability*/
12
13     MidPoint(h, M, midpoint);
14     IniMatrix(M, N, u);
15     InitialConditions ini_initail_conditions_u(h, M, N, u, midpoint);
16 }
17 //public function/s
18 void ParDiffEqBurgersEuler::Euler()
19 {
20     const double s_0{ k / (4 * h) };
21     const double s_1{ (beta * k) / pow(h,2) };
22
23     for (size_t j = 0; j < N - 1; j++)
24     {
25         for (size_t i = 1; i < M - 1; i++)
26         {
27             u[i][j + 1] = u[i][j] - s_0 * (pow(u[i + 1][j], 2) - pow(u[i - 1][j], 2)) +
28                 s_1 * (u[i + 1][j] - 2 * u[i][j] + u[i - 1][j]);
29         }
30
31         std::string save_file_name{ "Results/Eq4/Euler_Burgers_h_" + std::to_string(h) +
32             "_k_" + std::to_string(k) + "_beta_" + std::to_string(beta) + ".csv" };
33
34         Save(h, k, M, N, u, save_file_name);
35     }
36
37     //////////////////////////////////////
38 //public constructor/s
39 ParDiffEqBurgersRK2::ParDiffEqBurgersRK2(double h, double k)
40 : h(h), k(k)
41 {
42     beta = ((0.5 * pow(h, 2)) / k) - 0.01; /*satisfy (beta*k)/h^2 < 0.5 for stability*/
43
44     MidPoint(h, M, midpoint);
45     IniMatrix(M, N, u);
46     IniMatrix(M, N, v);
47     InitialConditions(h, M, N, u, midpoint);
48     InitialConditions(h, M, N, v, midpoint);
49 }
50 //public function/s
51 void ParDiffEqBurgersRK2::RK2()
52 {
53     const double r_0{ k / (8 * h) };
54     const double r_1{ (beta * k) / (2 * pow(h,2)) };
55
56     const double s_0{ k / (4 * h) };
57     const double s_1{ (beta * k) / pow(h,2) };
58
59     for (size_t j = 0; j < N - 1; j++)
60     {
61         for (size_t i = 1; i < M - 1; i++)
62         {
63             v[i][j + 1] = u[i][j] - r_0 * (pow(u[i + 1][j], 2) - pow(u[i - 1][j], 2)) +
64                 r_1 * (u[i + 1][j] - 2 * u[i][j] + u[i - 1][j]);
65         }
66         for (size_t i = 1; i < M - 1; i++)

```

```

66         {
67             u[i][j + 1] = u[i][j] - s_0 * (pow(v[i + 1][j + 1], 2) - pow(v[i - 1][j + 1],
68                 2)) + s_1 * (v[i + 1][j + 1] - 2 * v[i][j + 1] + v[i - 1][j + 1]);
69         }
70     }
71     std::string save_file_name{ "Results/Eq4/RK2_Burgers_h_" + std::to_string(h) + "_k_"
72         + std::to_string(k) + "_beta_" + std::to_string(beta) + ".csv" };
73     Save(h, k, M, N, u, save_file_name);
74 }
75
76 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
77
78 //public constructor/s
79 ParDiffEqBurgersImplicit::ParDiffEqBurgersImplicit(double h, double k)
80     : h(h), k(k)
81 {
82     beta = ((0.5 * pow(h, 2)) / k) - 0.01; /*satisfy (beta*k)/h^2 < 0.5 for stability*/
83
84     MidPoint(h, M, midpoint);
85     IniMatrix(M, M, u);
86     IniVector(M, b);
87     InitialConditions(h, M, M, u, midpoint);
88 }
89 //public function/s
90 void ParDiffEqBurgersImplicit::Implicit()
91 {
92     const double s{ (beta * k) / pow(h, 2) };
93     const double s_0{ -s };
94     const double s_1{ 1 + 2 * s };
95     const double s_2{ -s };
96
97     for (size_t j = 0; j < M - 1; j++)
98     {
99         SetVectorb(j);
100         Tri solve(s_0, s_1, s_2, M, j, b, u);
101     }
102
103     std::string save_file_name{ "Results/Eq4/Implicit_Burgers_h_" + std::to_string(h) +
104         "_k_" + std::to_string(k) + "_beta_" + std::to_string(beta) + ".csv" };
105     Save(h, k, M, M, u, save_file_name);
106 }
107 void ParDiffEqBurgersImplicit::SetVectorb(const size_t column)
108 {
109     b[0] = u[0][column];
110     for (size_t i = 1; i < M - 1; i++)
111     {
112         b[i] = u[i][column] - (k / (4 * h)) * (pow(u[i + 1][column], 2) - pow(u[i - 1][
113             column], 2));
114     }
115     b[M - 1] = u[M - 1][column];
116 }

```

```

1  //main.cpp
2
3  #include "0.0.Utility.h"
4  #include "1.Eq1.h"
5  #include "2.Eq2.h"
6  #include "3.Eq3.h"
7  #include "4.Burgers.h"
8
9  const static double h{ 0.1 };
10 const static double k{ 0.005 };
11
12 void DemoParDiffEq1()
13 {
14     ParDiffEq1Euler eq1_eul(h, k);
15     eq1_eul.Euler();
16
17     ParDiffEq1RK2 eq1_rk2(h, k);
18     eq1_rk2.RK2();
19 }
20 void DemoParDiffEq2()
21 {
22     ParDiffEq2Euler eq2_eul(h, k);
23     eq2_eul.Euler();
24
25     ParDiffEq2RK2 eq2_rk2(h, k);
26     eq2_rk2.RK2();
27 }
28 void DemoParDiffEq3()
29 {
30     ParDiffEq3Euler eq3_eul(h, k);
31     eq3_eul.Euler();
32
33     ParDiffEq3RK2 eq3_rk2(h, k);
34     eq3_rk2.RK2();
35
36     ParDiffEq3Implicit eq3_implicit(h, k);
37     eq3_implicit.Implicit();
38 }
39 void DemoBurgersEq()
40 {
41     ParDiffEqBurgersEuler Burgers_eul(h, k);
42     Burgers_eul.Euler();
43
44     ParDiffEqBurgersRK2 Burgers_rk2(h, k);
45     Burgers_rk2.RK2();
46
47     ParDiffEqBurgersImplicit Burgers_implicit(h, k);
48     Burgers_implicit.Implicit();
49 }
50
51 void MainMenu(bool& run)
52 {
53     size_t choice{ 0 };
54     const size_t final_choice{ 4 };
55
56     system("cls");
57     do
58     {
59         std::cout << "0.  $\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = 0$ , PARABOLIC" << "\n";
60         std::cout << "1.  $\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = 0$ , PARABOLIC" << "\n";
61         std::cout << "2.  $\frac{\partial u}{\partial t} - \beta \frac{\partial^2 u}{\partial x^2} = 0$ ,  $\beta > 0 \Rightarrow$  ELIPTIC,  

62              $\beta < 0 \Rightarrow$  HYPERBOLIC" << "\n";
63         std::cout << "3. Burgers' equation:  $\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} -$   

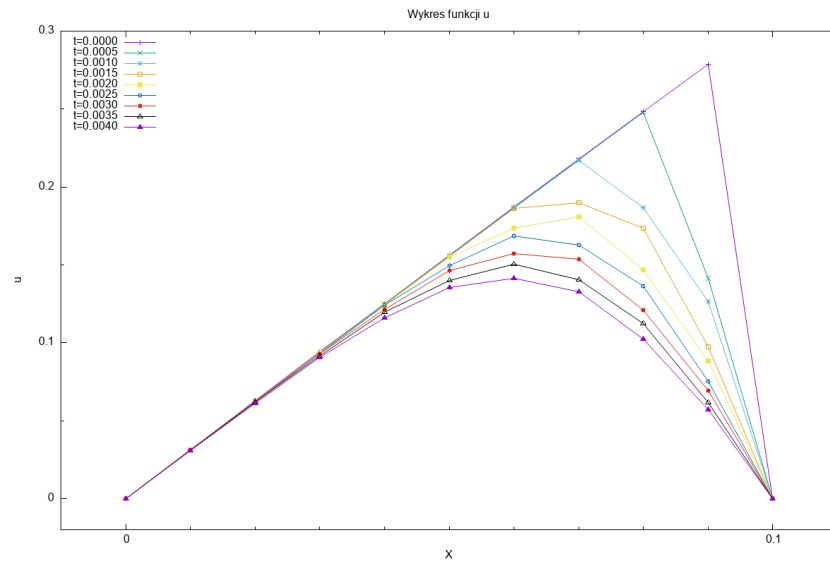
64              $\beta \frac{\partial^2 u}{\partial x^2} = 0$ ,  $\beta > 0 \Rightarrow$  ELIPTIC,  $\beta < 0 \Rightarrow$  HYPERBOLIC" << "\n";
65         ;
66         std::cout << "4. Exit" << "\n";
67         std::cin >> choice;
68         if (choice > final_choice)
69         {

```

```

67         std::cout << "Wrong choice!";
68         Sleep(1000);
69         system("cls");
70     }
71     system("cls");
72 } while (choice > final_choice);
73 if (choice == 0)
74 {
75     DemoParDiffEq1();
76 }
77 else if (choice == 1)
78 {
79     DemoParDiffEq2();
80 }
81 else if (choice == 2)
82 {
83     DemoParDiffEq3();
84 }
85 else if (choice == 3)
86 {
87     DemoBurgersEq();
88 }
89 else if (choice == final_choice)
90 {
91     run = false;
92 }
93 else
94 {
95     throw std::string("Fatal Error!");
96 }
97 }
98 int main()
99 {
100     bool run{ true };
101
102     while (run == true)
103     {
104         MainMenu(run);
105     }
106     return 0;
107 }

```

Rysunek 13: $k = 0.0005$, $h = 0.01$

15 dodatek B

15.1 transformata.cpp

```

1  #include "transformataHopfCole.h"
2  int main()
3  {
4      transformataHopfCole Burger;
5      Burger.wynikiKonczowe();
6  }

```

Listing 11: transformata.cpp

15.2 transformataHopfCole.h

```

1  #pragma once
2  #include <vector>
3  #include <cmath>
4  #include <algorithm>
5  #include <conio.h>
6  #include <math.h>
7  using namespace std;
8  #ifndef M_PI
9  #define M_PI 3.14159265358979323846

```

```

10 #endif
11
12 #ifndef M_E
13 #define M_E 2.71828182845904523536
14 #endif
15
16 class transformataHopfCole {
17
18     const int N = 11;          // liczba punktów siatki
19     const double k = 0.005, h = 0.1;
20     const double n = ((0.5 * pow(h, 2)) / k) - 0.01;
21     const double r = n * k / pow(h, 2);
22     double t = 0;
23
24     vector<double>theta;
25     vector<double>mu;
26     vector<double>inicjacjaThetaX0(); //wzor 8 z pracy
27     vector<double>liczenieThetaOdCzasu();
28
29     vector<double>liczenieMu(vector<double>newMu); //wzor 15
30     void zapisDoPliku(vector<double> Theta, double newT);
31     public:
32     void wynikiKoncowe();
33
34 };

```

Listing 12: transformataHopfCole.h

15.3 transformataHopfCole.cpp

```

1 #include "transformataHopfCole.h"
2 #include <iostream>
3 #include <fstream>
4 #include <string>
5
6
7
8 vector<double> transformataHopfCole::inicjacjaThetaX0() {
9     vector<double>newTheta;

```

```

10  for (int i = 0; i < N; ++i) {
11      double potega = (-1 / (2 * M_PI * n)) * (1 - cos(M_PI * (i * h)));
12      newTheta.push_back(exp(potega));
13  }
14  zapisDoPliku(liczenieMu(newTheta), 0);
15  return newTheta;
16 }
17
18 vector<double> transformataHopfCole::liczenieThetaOdCzasu() {
19     vector<double> thetaDlaCzasu(N);
20     thetaDlaCzasu = theta;
21     for (int j = 1; j < 10; ++j) {
22         t = j * k;
23         vector<double> v(N);
24         v[0] = (1 - 2 * r) * thetaDlaCzasu[0] + 2 * r * thetaDlaCzasu[1]; //
warunki brzegowe
25         v[N - 1] = 2 * r * thetaDlaCzasu[N - 2] + (1 - 2 * r) * thetaDlaCzasu[N
- 1];
26         for (int i = 1; i < (N - 1); ++i) {
27             double wynik = r * thetaDlaCzasu[i - 1] + (1 - 2 * r) * thetaDlaCzasu
[i] + r * thetaDlaCzasu[i + 1];
28             v[i] = wynik;
29         }
30         thetaDlaCzasu = v;
31         zapisDoPliku(liczenieMu(thetaDlaCzasu), t);
32
33     }
34     return thetaDlaCzasu;
35 }
36
37 vector<double> transformataHopfCole::liczenieMu(vector<double> newMu) {
38     vector<double> newMu1(N);
39     newMu1[0] = 0; //warunek poczatkowy
40     newMu1[N - 1] = 0; //warunek koncowy
41     for (int i = 1; i < (N - 1); ++i) {
42         newMu1[i] = -(n / h) * ((newMu[i + 1] - newMu[i - 1]) / newMu[i]);
43     }
44
45     return newMu1;

```

```

46 }
47
48 void transformataHopfCole::zapisDoPliku(vector<double> newMu, double newT)
    {
49     string nazwaPliku = "Wyniki1DlaT" + to_string(newT) + ".txt";
50     std::ofstream plik(nazwaPliku);
51
52     // Sprawdzamy, czy plik została otwarty poprawnie
53     if (!plik) {
54         std::cerr << "Nie zmona utworzy pliku!" << std::endl;
55         return;
56     }
57     for (int i = 0; i < size(newMu); ++i) {
58
59         plik << (i * h) << " " << newMu[i] << '\n';
60     }
61
62     // Zamykamy plik
63     plik.close();
64 }
65
66 void transformataHopfCole::wynikiKoncowe() {
67     theta = inicjacjaThetaX0();
68     theta = liczenieThetaOdCzasu();
69 }

```

Listing 13: transformataHopfCole.cpp