Numeryczne wyznaczanie rozwiązań równania Burgersa przy pomocy metody różnic skończonych

Konrad Bonicki, Tomasz Orzechowski, Maciej Pestka

25 czerwca 2024

Spis treści

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:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = v\frac{\partial^2 u}{\partial x^2} \tag{1}$$

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 cieplne. Wygląda następująco:

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

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

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:

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

$$-2v\frac{\theta_{xt}\theta - \theta_x\theta_t}{\theta^2} + 4v^2\frac{\theta\theta_x\theta_{xx} - \theta_x^3}{\theta^3} = -2v^2\frac{\theta_{xxx}\theta^2 - 3\theta\theta_x\theta_{xx} + 2\theta_x^3}{\theta^3}$$

$$-2v\theta(\theta_{xt}\theta - \theta_x\theta_t) + 4v^2(\theta\theta_x\theta_{xx} - \theta_x^3) = -2v^2(\theta_{xxx}\theta^2 - 3\theta\theta_x\theta_{xx} + 2\theta_x^3)$$

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

$$\theta\theta_{xt} - \theta_x\theta_t - 2v\theta_x\theta_{xx} = v\theta\theta_{xxx} - 3v\theta_x\theta_{xx}$$

$$\theta\theta_{xt} - \theta_x\theta_t = v(\theta\theta_{xxx} - \theta_x\theta_{xx})$$

$$\theta\frac{\partial\theta^2}{\partial x\partial t} - \frac{\partial\theta}{\partial x}\frac{\partial\theta}{\partial t} = v(\theta\frac{\partial^3\theta}{\partial x^3} - \frac{\partial\theta}{\partial x}\frac{\partial^2\theta}{\partial x^2})$$

$$\frac{\partial\theta}{\partial t}(\theta\frac{\partial\theta}{\partial x} - \frac{\partial\theta}{\partial x}) = v\frac{\partial^2\theta}{\partial x^2}(\theta\frac{\partial\theta}{\partial x} - \frac{\partial\theta}{\partial x})$$

$$\theta\theta_x - \theta_x \neq 0$$

$$\theta_t = v\theta_{xx}$$

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 :

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

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

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

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

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

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

Wyznaczamy $\theta(0,0)$:

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

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

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

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

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

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

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:

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

$$X(x)T`(t) = vX``(x)T(t)$$
$$\frac{T`(t)}{vT(t)} = \frac{X``(x)}{X(x)} = -\lambda^2$$
$$T`(t) = -\lambda^2 vT(t)$$

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

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

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

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

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

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

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

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

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

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

Funkcja sinus przyjmuje wartości 0 dla $\lambda = n\pi, \quad n = 0, 1, 2, ...$, 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 $B_n \quad (n = 0, 1, 2, ...)$:

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

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

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:*

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

gdzie dla pewnego punktu ξ leżącego między c i x:

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

7 Metoda różnic skończonych

Metoda różnic skończonych polega na przybliżaniu pochodnych za pomocą różnic skończonych. Zarówno dziedzina przestrzenna, jak i czasowa 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:

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

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:

$$x(t+h) \approx x(t) + hf(t,x) \tag{5}$$

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:

$$y' = f(t,y) \tag{6}$$

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:**

$$y_0 = y(t_0) \tag{7}$$

2. **Obliczenie przyrostów:**

$$k_1 = hf(t_n, y_n) \tag{8}$$

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

3. **Obliczenie następnej wartości y:**

$$y_{n+1} = y_n + k_2 \tag{10}$$

4. **Aktualizacja czasu:**

$$t_{n+1} = t_n + h \tag{11}$$

10 Rozwiązanie równania Burgersa

Rozważamy cztery równania różniczkowe cząstkowe.

Równanie transportu

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = 0, 0 \leq x \leq 10, t > 0 \tag{12}$$

Nielepkie równanie Burgersa

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0, 0 \leq x \leq 10, t > 0 \tag{13}$$

Równanie ciepła

$$\frac{\partial u}{\partial t} - \beta\frac{\partial^2 u}{\partial x^2} = 0, 0 \leq x \leq 10, t > 0, \beta > 0 \tag{14}$$

Równanie Burgersa (nieliniowe, paraboliczne)

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \beta\frac{\partial^2 u}{\partial x^2} = 0, 0 \leq x \leq 10, t > 0, \beta > 0 \tag{15}$$

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

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

Wtedy niech warunek początkowy wszystkich równań (funkcja Gaussa)

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

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.

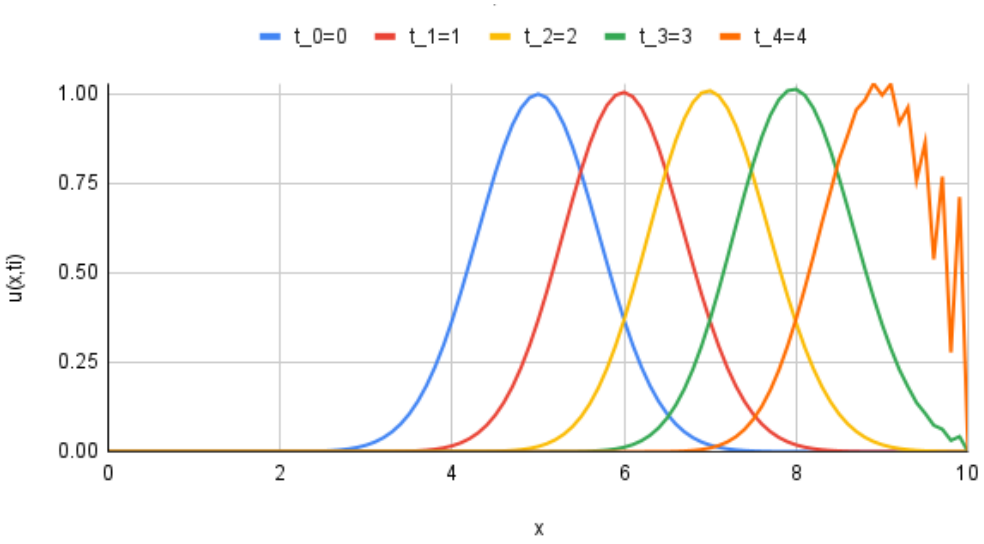
Wtedy stosując metodę Eulera:

dla równania transportu

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

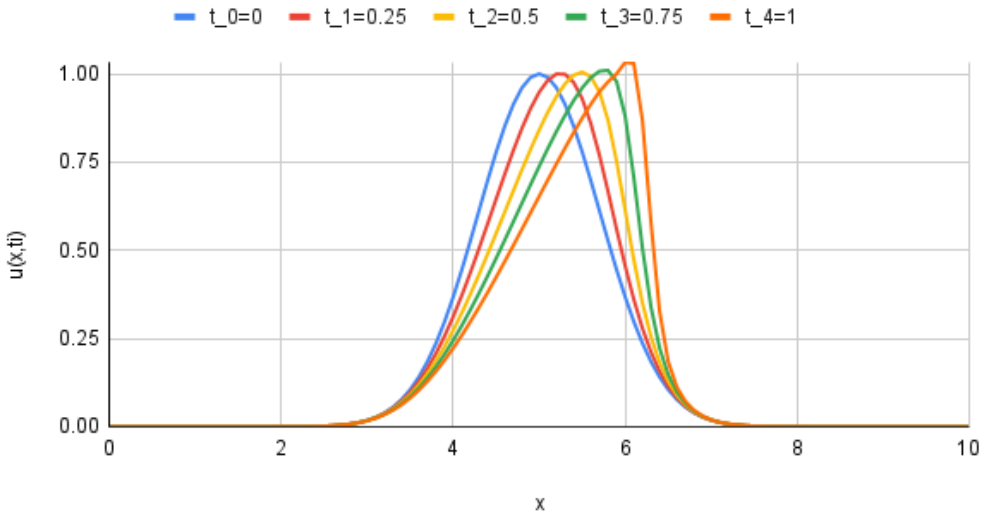
dla nielepkiego równania Burgersa

Rysunek 1: Równanie transportu, metoda Eulera



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

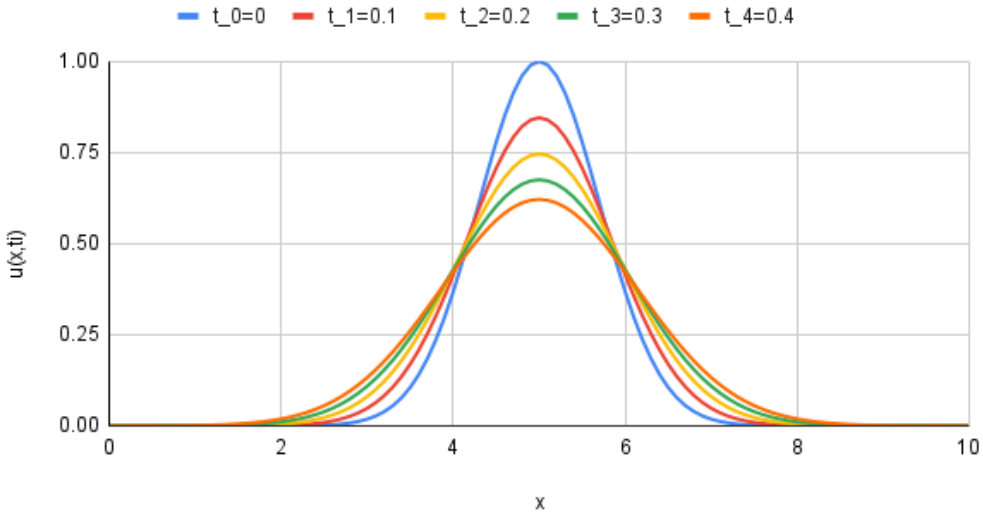
Rysunek 2: Nielepkie równanie Burgersa, metoda Eulera



dla równania ciepła

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

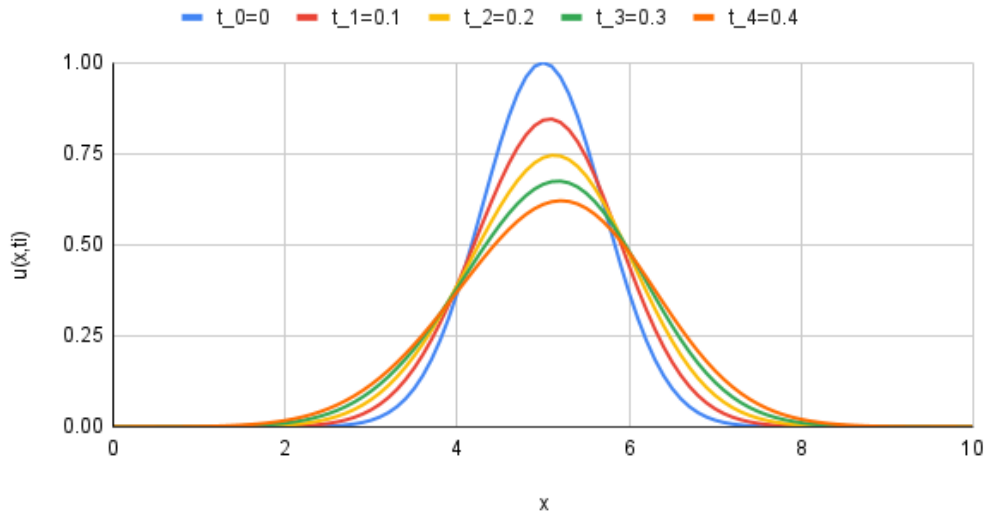
Rysunek 3: Równanie ciepła, metoda Eulera



dla równania Burgersa

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

Rysunek 4: Równanie Burgersa, metoda Eulera

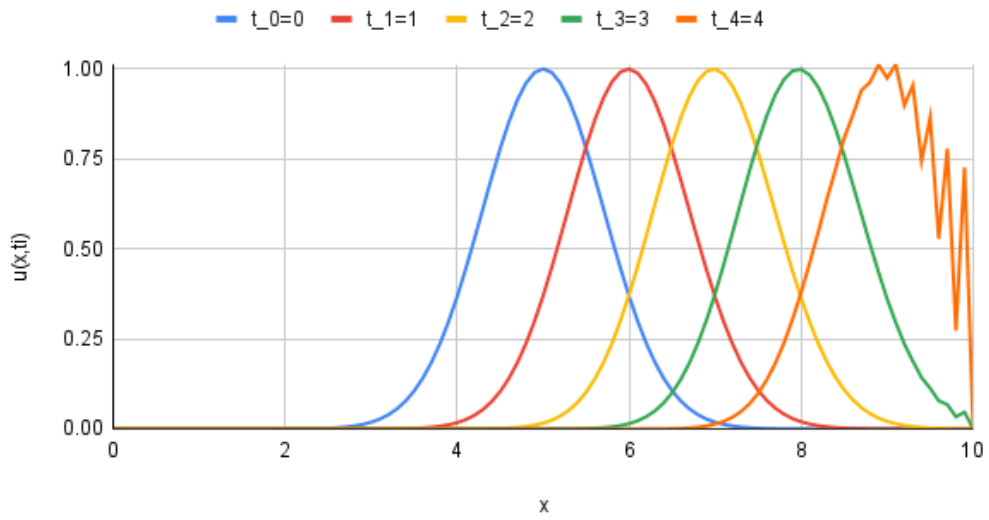


Stosując metodę Rungego-Kutty drugiego rzędu:

dla równania transportu

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

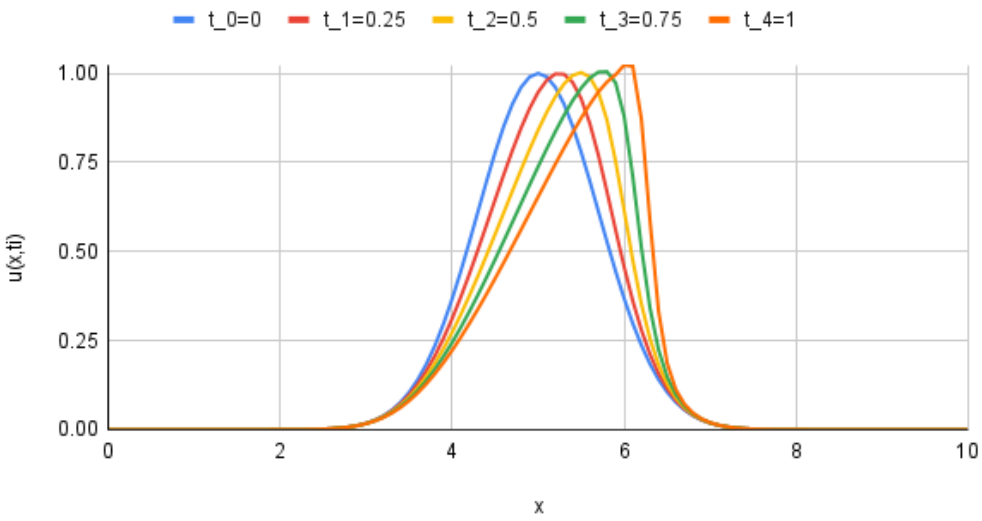
Rysunek 5: Równanie transportu, metoda RK2



dla nielepkiego równania Burgersa

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

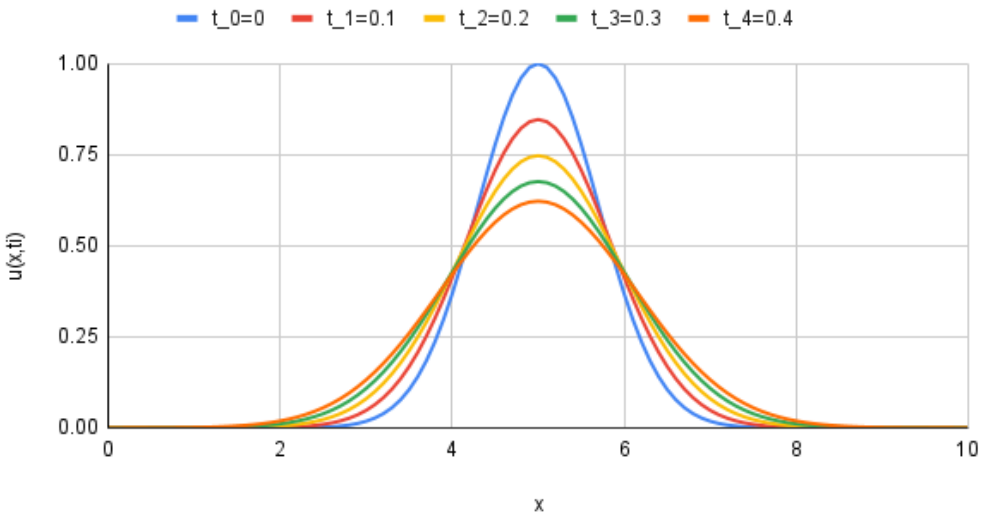
Rysunek 6: Nielepkie równanie Burgersa, metoda RK2



dla równania ciepła

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

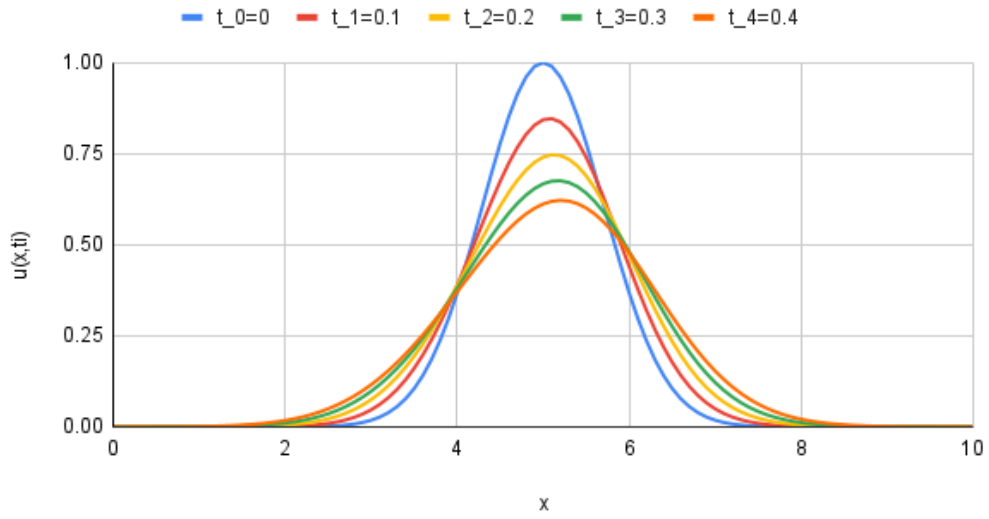
Rysunek 7: Równanie ciepła, metoda RK2



dla równania Burgersa

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

Rysunek 8: Równanie Burgersa, metoda RK2



Stosując metodę niejawną:

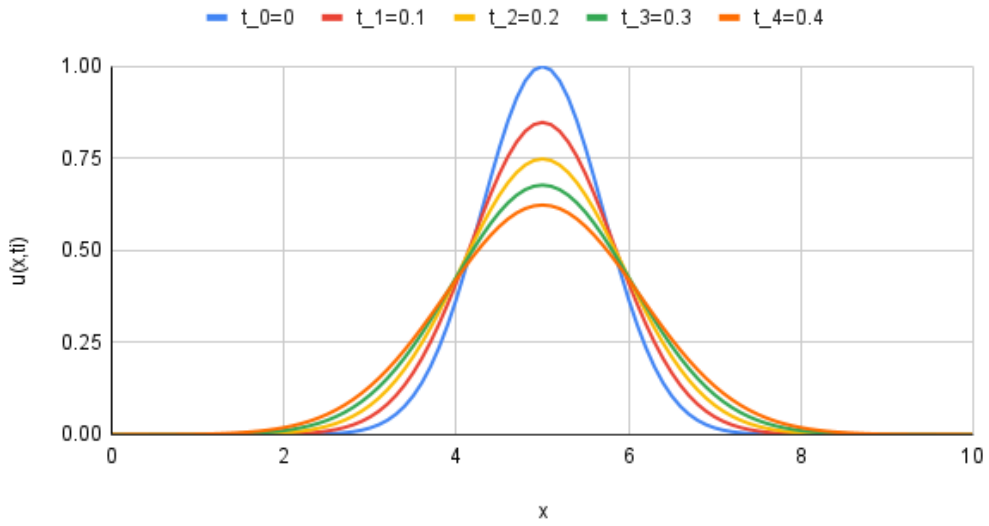
dla równania ciepła

$$\begin{aligned} &\frac{1}{k}(u_{i,j} - u_{i,j-1}) - \frac{\beta}{h^2}(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) = 0 \\ &u_{i,j-1} = -\frac{\beta k}{h^2}u_{i-1,j} + (1 + 2\frac{\beta k}{h^2})u_{i,j} - \frac{\beta k}{h^2}u_{i+1,j} \end{aligned} \tag{28}$$

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

$$u_{i,j} = A^{-1}u_{i,j-1} \tag{29}$$

Rysunek 9: Równanie ciepła, metoda niejawna



dla równania Burgersa

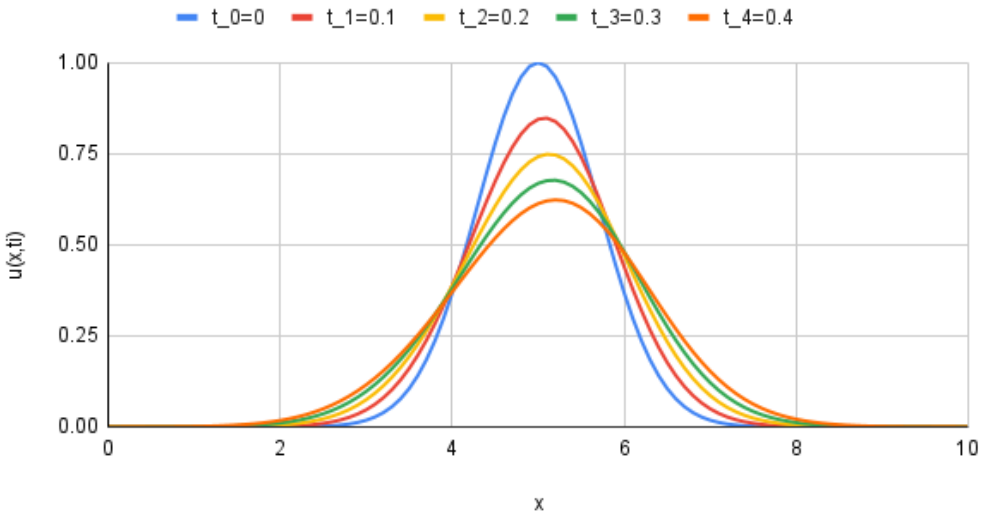
$$\begin{aligned} &\frac{1}{k}(u_{i,j} - u_{i,j-1}) + \frac{1}{4h}(u^2_{i+1,j-1} - u^2_{i-1,j-1}) - \frac{\beta}{h^2}(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) = 0 \\ &u_{i,j-1} - \frac{k}{4h}(u^2_{i+1,j-1} - u^2_{i-1,j-1}) = -\frac{\beta k}{h^2}u_{i-1,j} + (1 + 2\frac{\beta k}{h^2})u_{i,j} - \frac{\beta k}{h^2}u_{i+1,j}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

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

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

Algorytm, który realizuje opisaną metodę niejawną korzysta z procedury tri rozwiązującej układ równań o macierzy trójprzekątniowej. 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. Zobrazowna 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 osiągnieciem kresu i szybciej opada po osiągnięciu kresu. Zobrazowna zostaje nieliniowość.

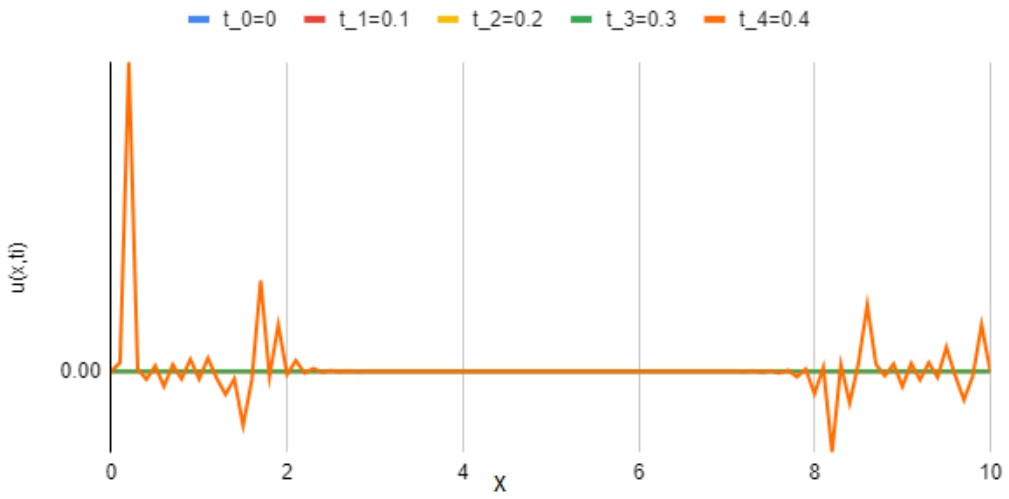
Dla równania ciepła, kolejne przesunięcia w czasie powodują zmniejszenie kresu funkcji Gaussa. Maksymalna wartość u maleje w tym samym punkcie w przestrzeni (5)

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.

Rysunek 11: Równanie Burgersa, metoda Eulera, β=1.3



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

[a4paper,11pt]article [T1]fontenc [utf8]inputenc xcolor geometry subfig left=2.5cm,right=2.5cm,t
multicol fancyhdr listings subcaption pdfpages

amsmath graphicx hyperref fancyhdr tocloft

11 Opis programu transformaty Hopf Cole'a

Program komputerowy został napisany na podstawie wzorów 32:

$$\begin{aligned} \theta_{i,j+1} &= (1-2r)\theta_{i,j} + 2r\theta_{i+1,j}, \quad i = 0 \\ \theta_{i,j+1} &= 2r\theta_{i-1,j} + (1-2r)\theta_{i,j} + r\theta_{i+1,j}, \quad 1 \leq i \leq N-1 \\ \theta_{i,j+1} &= 2r\theta_{i-1,j} + (1-2r)\theta_{i,j} \end{aligned} \tag{32}$$

. Natomiast równanie na u zostało zrobione z wzoru:

$$u(x_i, t_j) = -\frac{\beta}{h}\left(\frac{\theta_{i+1,j} - \theta_{i-1,j}}{\theta_{i,j}}\right) \quad 1 \leq i \leq N-1 \tag{33}$$

. Gdzie β jest współczynnikiem lepkości, h krokiem przestrzennym, a k - krokiem czasowym.

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 definiuję standardowych klas i funkcji z biblioteki standardowej w C++, takich jak "cout", "vecotr", "string", co skraca trochę kod. Potem zostały definiowanie stała Pi oraz e, jeśli nie są szybciej zdefiniowanie, 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 zdefiniowania klasa, w której są wszystkie metody oraz stale potrzebnie do pracy programu.

```
1    class transformataHopfCole {
```

```
2         const double k = 0.005, h = 0.1;
3                 const int N = 1 / h + 1;        // liczba punktów siatki
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(); //wzror 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 wynikiKonczowe();
18      };
```

Listing 3: klasa tranformataHopfCofe

Pierwsza stała oznacza liczbę punktów, ”k”odpowiada za krok czasowy, natomiast ”h”za krok przestrzennego 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. Ostaną 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: poczadek 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 wygenerowana dla wszystkich θ wartości początkowych, z których potem zostaną wykorzystanie do dalszych obliczeń. Pod koniec metody zostanie wywołania kolejna metoda, która zapiszę 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 policzenie "t", czyli kroku czasowego. Potem jest zdefiniowany nowy vector, w którym zostaną zapisanie nowe wartości. Druga pętla służy do policzenia dla θ od 1 do N-1. Po drugiej pętli zostaną nadpisanie wartości dla theyaDlaCzasu nowymi wartościami dla konkretnego czasu. Na końcu jest wywołania metoda, która otrzymanie wyniki zapiszę 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 < 3000; ++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 wykorzystuję 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 poczatkowy
4   newMu1[N - 1] = 0;//warunek koncowy
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 łzosta otwarty poprawnie
6   if (!plik) {
7     std::cerr << "Nie żmona ćotworzy 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 zapisuję dane do pliku.

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

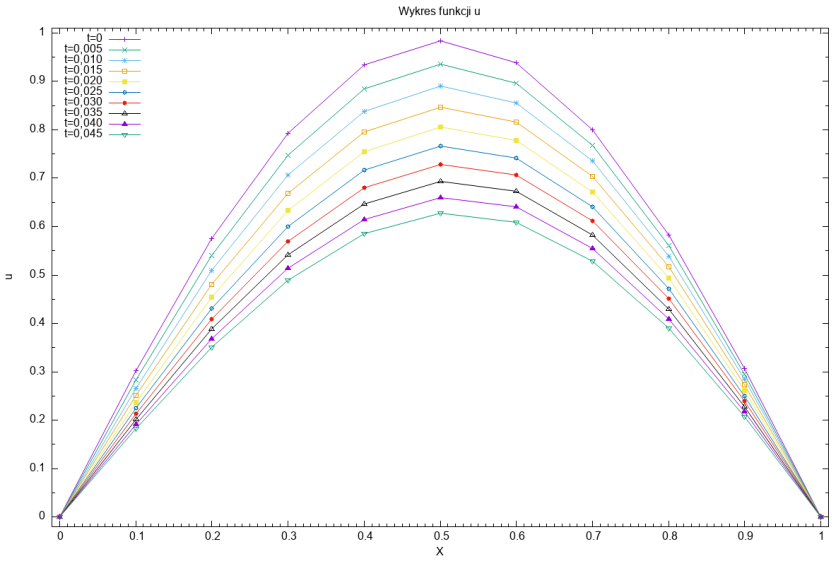
Listing 9: metoda wynikiKonczowe

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



Rysunek 12: k = 0.005, h = 0.1

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 **??**.

Na pozostały wykresach **??**, można zauważyć jak zmiana współczynników wpływa na generowanie wykresów. Jak zmienia się współczynnik lepkości to można zauważyć że amplituda zwiększa się lub zamieszcza, zależy od wartości. Natomiast gdy się zmienia krok przestrzeni to zmienia się ilość punktów na osi x, a gdy ostatnią wartość się zmieni (krok czasowy), to powstaje więcej lub mniej rozwiązań θ od czasu.Na każdych z wykresów można zauważyć, że dla coraz większego czasu amplituda zamieszcza się oraz lekko przesuwa się w prawą stronę.

Gdy warunek na stabilność, $k/h^2 \leq 0.5$ jest nie zachowany to program wygeneruję wyniki, ale w pliku tekstowym pojawiają się wartości "ind", których nie można przeanalizować oraz naszkicować na wykresie tych punktów.

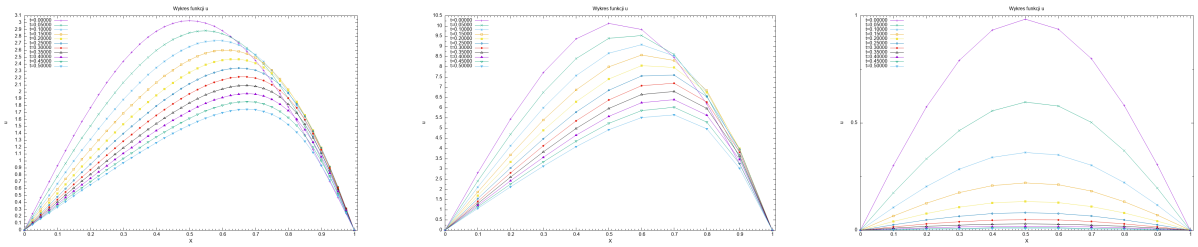
Przedstawione na wykresach 14, można zobaczyć, że dla większego czasu, że wykres coraz mniej przypomina wykres sinusa. Może wynikać to z błędach numerycznych, które z

x\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

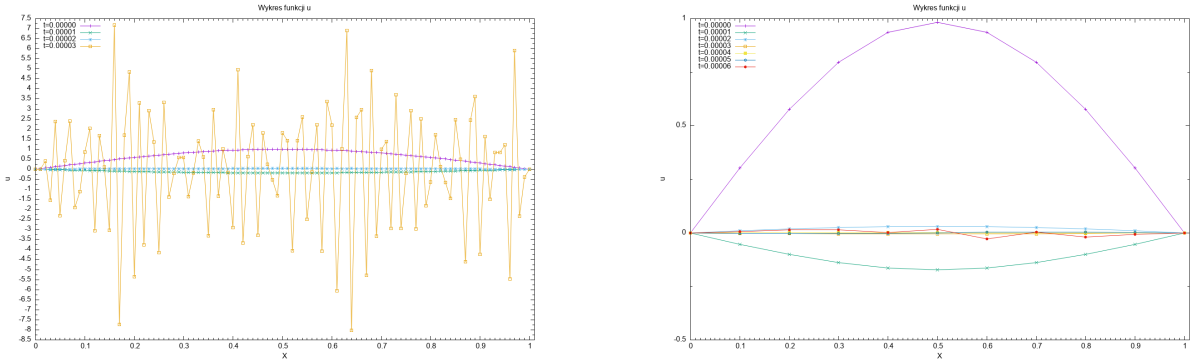
x\t	0.030	0.35	0.40	0.45	0.45
0	0	0	0	0	0
0.1	0.213091	0.202201	0.192041	0.182528	0.182528
0.2	0.408602	0.387857	0.368453	0.350247	0.350247
0.3	0.569591	0.540975	0.514101	0.488803	0.488803
0.4	0.680618	0.6469	0.615061	0.584955	0.584955
0.5	0.729015	0.693493	0.659731	0.627636	0.627636
0.6	0.706609	0.672786	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



(a) β=0.1, k=0.001, h=0.025 (b) β=0.1, k=0.005, h=0.1 (c) β=0.99, k=0.005, h=0.1

Rysunek 13: Wykresy dla rozwiązań transformaty



(a) $\beta = 12000$, $k = 0.00001$, $h = 0.01$ (b) $\beta = 12000$, $k = 0.00001$, $h = 0.1$

Rysunek 14: Źle dobrane warunki

każdym rozwiązaniem gromadzą się i robią się coraz większe.

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

```cpp
//Utility.h

#include <cmath>
#include <fstream>
#include <iostream>
#include <string>
#include <windows.h>
#include <vector>

class Save
{
public:
    //public constructor/s
    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);
};

////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////

class MidPoint
{
public:
    //public constructor/s
    MidPoint(const double h, const size_t M, double& midpoint);
};

////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////

class IniVector
{
public:
    //public constructor/s
    IniVector(const size_t M, std::vector<double>& vec);
};

////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////

class IniMatrix
{
public:
    //public constructor/s
    IniMatrix(const size_t M, const size_t N, std::vector<std::vector<double>>& mat);
};

////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////

class InitialConditions
{
public:
    //public constructor/s
    InitialConditions(const double h, const size_t M, const size_t N, std::vector<std::
    vector<double>>& u, const double midpoint);
};
```

```cpp
//Utility.cpp

#include "0.0.Utility.h"

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)
{
    std::ofstream out;
    out.open(save_file_name);
    if (out.is_open())
    {
        for (size_t i = 0; i < M; i++)
        {
            for (size_t j = 0; j < N; j++)
            {
                out << i * h << "," << j * k << "," << u[i][j] << "\n";
            }
        }
    }
    else
    {
        throw std::string("Could not save to file!");
    }
    out.close();
}

////////////////////////////////////////////////////////////////////////////////////////////////////

MidPoint::MidPoint(const double h, const size_t M, double& midpoint)
{
    midpoint = ((M - 1) * h) / 2;;
}

////////////////////////////////////////////////////////////////////////////////////////////////////

IniVector::IniVector(const size_t M, std::vector<double>& vec)
{
    vec.resize(M);
    for (size_t i = 0; i < M; i++)
    {
        vec[i] = 0;
    }
}

////////////////////////////////////////////////////////////////////////////////////////////////////

IniMatrix::IniMatrix(const size_t M, const size_t N, std::vector<std::vector<double>>& mat)
{
    mat.resize(M, std::vector<double>(N));
    for (size_t i = 0; i < M; i++)
    {
        for (size_t j = 0; j < N; j++)
        {
            mat[i][j] = 0;
        }
    }
}

////////////////////////////////////////////////////////////////////////////////////////////////////

InitialConditions::InitialConditions(const double h, const size_t M, const size_t N, std::vector<std::vector<double>>& vec, const double midpoint)
{
    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  };
```

```cpp
//MatrixOperations.cpp

#include "0.0.Utility.h"
#include "0.1.MatrixOperations.h"

MatrixCopy::MatrixCopy(const size_t M, const std::vector<std::vector<double>>& mat_from, std::vector<std::vector<double>>& mat_to)
{
    for (size_t i = 0; i < M; i++)
    {
        for (size_t j = 0; j < M; j++)
        {
            mat_to[i][j] = mat_from[i][j];
        }
    }
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

VectorCopy::VectorCopy(const size_t M, const std::vector<double>& vec_from, std::vector<double>& vec_to)
{
    for (size_t i = 0; i < M; i++)
    {
        vec_to[i] = vec_from[i];
    }
}
```

```cpp
//LUGaussSpecial.h

class LUGaussSpecial
{
public:
    //public constructor/s
    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);
private:
    //private function/s
    void SpecialGaussianElimination(std::vector<double>& b, std::vector<double>& x);
    void ExtractTriVectors(const double s_0, const double s_1, const double s_2);
    //private variable/s
    const size_t M;
    std::vector<double> a{};
    std::vector<double> c{};
    std::vector<double> d{};
};
```

```cpp
//LUGaussSpecial.cpp

#include "0.0.Utility.h"
#include "0.1.MatrixOperations.h"
#include "0.2.LUGaussSpecial.h"

//public contstructor/s
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)
    : M(M)
{
    IniVector(M, a);
    IniVector(M, c);
    IniVector(M, d);

    ExtractTriVectors(s_0, s_1, s_2);
    SpecialGaussianElimination(b, x);
}
//private function/s
void LUGaussSpecial::SpecialGaussianElimination(std::vector<double>& b, std::vector<double>& x)
{
    for (size_t i = 1; i < M; i++)
    {
        d[i] = d[i] - (a[i - 1] / d[i - 1]) * c[i - 1];
        b[i] = b[i] - (a[i - 1] / d[i - 1]) * b[i - 1];
    }
    x[M - 1] = b[M - 1] / d[M - 1];
    for (int i = M - 2; i >= 0; i--)
    {
        x[i] = (b[i] - (c[i] * x[i + 1])) / d[i];
    }
}
void LUGaussSpecial::ExtractTriVectors(const double s_0, const double s_1, const double s_2)
{
    for (size_t i = 0; i < M; i++)
    {
        a[i] = s_0;
        d[i] = s_1;
        c[i] = s_2;
    }
}
```

```cpp
//Tri.h

class Tri
{
public:
    //public constructor/s
    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);
private:
    //private function/s
    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);
    void AssignResults(const size_t column, std::vector<std::vector<double>>& mat);
    //private variable/s
    const size_t M;
    std::vector<double> x{};
};
```

```cpp
//Tri.cpp

#include "0.0.Utility.h"
#include "0.2.LUGaussSpecial.h"
#include "0.3.Tri.h"

//public contstructor/s
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)
    : M(M)
{
    IniVector(M, x);

    ExecuteTri(s_0, s_1, s_2, column, b, mat);
}
//private function/s
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)
{
    LUGaussSpecial solve(M, s_0, s_1, s_2, b, x);
    AssignResults(column, mat);
}
void Tri::AssignResults(const size_t column, std::vector<std::vector<double>>& mat)
{
    for (size_t i = 1; i < M - 1; i++)
    {
        mat[i][column + 1] = x[i];
    }
}
```

```cpp
//Eq1.h

#pragma once

//par_u/par_t + par_u/par_x = 0  differential equation using Euler + Runge-Kutta 2nd order methods

class ParDiffEq1Euler
{
public:
    //public constructor/s
    ParDiffEq1Euler(double h, double k);
    //public function/s
    void Euler();
private:
    //private variable/s
    double midpoint{};
    const size_t M{ 101 };
    const size_t N{ 1001 };
    std::vector<std::vector<double>> u{};

    const double h;
    const double k;
};

class ParDiffEq1RK2
{
public:
    //public constructor/s
    ParDiffEq1RK2(double h, double k);
    //public function/s
    void RK2();
private:
    //private variable/s
    double midpoint{};
    const size_t M{ 101 };
    const size_t N{ 1001 };
    std::vector<std::vector<double>> u{};
    std::vector<std::vector<double>> v{};

    const double h;
    const double k;
};
```

```cpp
//Eq1.cpp

#include "0.0.Utility.h"
#include "1.Eq1.h"

//public contstructor/s
ParDiffEq1Euler::ParDiffEq1Euler(double h, double k)
    : h(h), k(k)
{
    MidPoint(h, M, midpoint);
    IniMatrix(M, N, u);
    InitialConditions(h, M, N, u, midpoint);
}
//public function/s
void ParDiffEq1Euler::Euler()
{
    const double s{ k / (2 * h) };

    for (size_t j = 0; j < N - 1; j++)
    {
        for (size_t i = 1; i < M - 1; i++)
        {
            u[i][j + 1] = u[i][j] - s * (u[i + 1][j] - u[i - 1][j]);
        }
    }

    std::string save_file_name{ "Results/Eq1/Euler_Eq_1_h_" + std::to_string(h) + "_k_" +
     std::to_string(k) + ".csv" };

    Save(h, k, M, N, u, save_file_name);
}

////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////

//public contstructor/s
ParDiffEq1RK2::ParDiffEq1RK2(double h, double k)
    : h(h), k(k)
{
    MidPoint(h, M, midpoint);
    IniMatrix(M, N, u);
    IniMatrix(M, N, v);
    InitialConditions(h, M, N, u, midpoint);
    InitialConditions(h, M, N, v, midpoint);
}
//public function/s
void ParDiffEq1RK2::RK2()
{
    const double r{ k / (4 * h) };
    const double s{ k / (2 * h) };

    for (size_t j = 0; j < N - 1; j++)
    {
        for (size_t i = 1; i < M - 1; i++)
        {
            v[i][j + 1] = u[i][j] - r * (u[i + 1][j] - u[i - 1][j]);
        }
        for (size_t i = 1; i < M - 1; i++)
        {
            u[i][j + 1] = u[i][j] - s * (v[i + 1][j + 1] - v[i - 1][j + 1]);
        }
    }

    std::string save_file_name{ "Results/Eq1/RK2_Eq_1_h_" + std::to_string(h) + "_k_" +
    std::to_string(k) + ".csv" };

    Save(h, k, M, N, u, save_file_name);
}
```

```cpp
//Eq2.h

//par_u/par_t + u*par_u/par_x = 0 differential equation using Euler + Runge-Kutta 2nd order methods

class ParDiffEq2Euler
{
public:
    //public constructor/s
    ParDiffEq2Euler(double h, double k);
    //public function/s
    void Euler();
private:
    //private variable/s
    double midpoint{};
    const size_t M{ 101 };
    const size_t N{ 1001 };
    std::vector<std::vector<double>> u{};

    const double h;
    const double k;
};

class ParDiffEq2RK2
{
public:
    //public constructor/s
    ParDiffEq2RK2(double h, double k);
    //public function/s
    void RK2();
private:
    //private variable/s
    double midpoint{};
    const size_t M{ 101 };
    const size_t N{ 1001 };
    std::vector<std::vector<double>> u{};
    std::vector<std::vector<double>> v{};

    const double h;
    const double k;
};
```

```cpp
//Eq2.cpp

#include "0.0.Utility.h"
#include "2.Eq2.h"

//public contstructor/s
ParDiffEq2Euler::ParDiffEq2Euler(double h, double k)
    : h(h), k(k)
{
    MidPoint(h, M, midpoint);
    IniMatrix(M, N, u);
    InitialConditions(h, M, N, u, midpoint);
}
//public function/s
void ParDiffEq2Euler::Euler()
{
    const double s{ k / (4 * h) };

    for (size_t j = 0; j < N - 1; j++)
    {
        for (size_t i = 1; i < M - 1; i++)
        {
            u[i][j + 1] = u[i][j] - s * (pow(u[i + 1][j], 2) - pow(u[i - 1][j], 2));
        }
    }

    std::string save_file_name{ "Results/Eq2/Euler_Eq_2_h_" + std::to_string(h) + "_k_" +
     std::to_string(k) + ".csv" };

    Save(h, k, M, N, u, save_file_name);
}

////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////

//public contstructor/s
ParDiffEq2RK2::ParDiffEq2RK2(double h, double k)
    : h(h), k(k)
{
    MidPoint ini_midpoint(h, M, midpoint);
    IniMatrix(M, N, u);
    IniMatrix(M, N, v);
    InitialConditions ini_initial_conditions_u(h, M, N, u, midpoint);
    InitialConditions ini_initial_conditions_v(h, M, N, v, midpoint);
}
//public function/s
void ParDiffEq2RK2::RK2()
{
    const double r{ k / (8 * h) };
    const double s{ k / (4 * h) };

    for (size_t j = 0; j < N - 1; j++)
    {
        for (size_t i = 1; i < M - 1; i++)
        {
            v[i][j + 1] = u[i][j] - r * (pow(u[i + 1][j], 2) - pow(u[i - 1][j], 2));
        }
        for (size_t i = 1; i < M - 1; i++)
        {
            u[i][j + 1] = u[i][j] - s * (pow(v[i + 1][j + 1], 2) - pow(v[i - 1][j + 1], 2
            ));
        }
    }

    std::string save_file_name{ "Results/Eq2/RK2_Eq_2_h_" + std::to_string(h) + "_k_" +
    std::to_string(k) + ".csv" };

    Save(h, k, M, N, u, save_file_name);
}
```

```cpp
//Eq3.h

//par_u/par_t - beta*par_u^2/par_x^2 = 0 differential equation using Euler + Runge-Kutta 2nd order + implicit methods

class ParDiffEq3Euler
{
public:
    //public constructor/s
    ParDiffEq3Euler(double h, double k);
    //public function/s
    void Euler();
private:
    //private variable/s
    double midpoint{};
    const size_t M{ 101 };
    const size_t N{ 101 };
    std::vector<std::vector<double>> u{};

    double beta;
    const double h;
    const double k;
};

class ParDiffEq3RK2
{
public:
    //public constructor/s
    ParDiffEq3RK2(double h, double k);
    //public function/s
    void RK2();
private:
    //private variable/s
    double midpoint{};
    const size_t M{ 101 };
    const size_t N{ 101 };
    std::vector<std::vector<double>> u{};
    std::vector<std::vector<double>> v{};

    double beta;
    const double h;
    const double k;
};

class ParDiffEq3Implicit
{
public:
    //public constructor/s
    ParDiffEq3Implicit(double h, double k);
    //public function/s
    void Implicit();
private:
    //private function/s
    void SetVectorb(const size_t column);
    //private variable/s
    double midpoint{};
    const size_t M{ 101 };
    std::vector<std::vector<double>> u{};
    std::vector<double> b{};

    double beta;
    const double h;
    const double k;
};
```

```cpp
//Eq3.cpp

#include "0.0.Utility.h"
#include "0.3.Tri.h"
#include "3.Eq3.h"

//public contstructor/s
ParDiffEq3Euler::ParDiffEq3Euler(double h, double k)
    : h(h), k(k)
{
    beta = ((0.5 * pow(h, 2)) / k) - 0.01; /*satisfy (beta*k)/h^2 < 0.5 for stability*/

    MidPoint(h, M, midpoint);
    IniMatrix(M, N, u);
    InitialConditions(h, M, N, u, midpoint);
}
//public function/s
void ParDiffEq3Euler::Euler()
{
    const double s{ (beta * k) / pow(h,2) };

    for (size_t j = 0; j < N - 1; j++)
    {
        for (size_t i = 1; i < M - 1; i++)
        {
            u[i][j + 1] = u[i][j] + s * (u[i + 1][j] - 2 * u[i][j] + u[i - 1][j]);
        }
    }

    std::string save_file_name{ "Results/Eq3/Euler_Eq_3_h_" + std::to_string(h) + "_k_" +
     std::to_string(k) + "_beta_" + std::to_string(beta) + ".csv" };

    Save(h, k, M, N, u, save_file_name);
}

////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////

//public contstructor/s
ParDiffEq3RK2::ParDiffEq3RK2(double h, double k)
    : h(h), k(k)
{
    beta = ((0.5 * pow(h, 2)) / k) - 0.01; /*satisfy (beta*k)/h^2 < 0.5 for stability*/

    MidPoint(h, M, midpoint);
    IniMatrix(M, N, u);
    IniMatrix(M, N, v);
    InitialConditions(h, M, N, u, midpoint);
    InitialConditions(h, M, N, v, midpoint);
}
//public function/s
void ParDiffEq3RK2::RK2()
{
    const double r{ (beta * k) / (2 * pow(h,2)) };
    const double s{ (beta * k) / pow(h,2) };

    for (size_t j = 0; j < N - 1; j++)
    {
        for (size_t i = 1; i < M - 1; i++)
        {
            v[i][j + 1] = u[i][j] + r * (u[i + 1][j] - 2 * u[i][j] + u[i - 1][j]);
        }
        for (size_t i = 1; i < M - 1; i++)
        {
            u[i][j + 1] = u[i][j] + s * (v[i + 1][j + 1] - 2 * v[i][j + 1] + v[i - 1][j +
             1]);
        }
    }

```

```
67      std::string save_file_name{ "Results/Eq3/RK2_Eq_3_h_" + std::to_string(h) + "_k_" +
        std::to_string(k) + "_beta_" + std::to_string(beta) + ".csv" };
68
69      Save(h, k, M, N, u, save_file_name);
70  }
71
72  ////////////////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////
73
74  //public contstructor/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) +
        "_k_" + std::to_string(k) + "_beta_" + std::to_string(beta) + ".csv" };
100
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 }
```

```
//Burgers.h

//Burgers' equation: par_u/par_t + u*par_u/par_x - beta*par_u^2/par_x^2=0 = 0 using Euler + Runge-Kutta 2nd order + implicit methods

class ParDiffEqBurgersEuler
{
public:
    //public constructor/s
    ParDiffEqBurgersEuler(double h, double k);
    //public function/s
    void Euler();
private:
    //private variable/s
    double midpoint{};
    const size_t M{ 101 };
    const size_t N{ 101 };
    std::vector<std::vector<double>> u{};

    double beta;
    double h;
    double k;
};

class ParDiffEqBurgersRK2
{
public:
    //public constructor/s
    ParDiffEqBurgersRK2(double h, double k);
    //public function/s
    void RK2();
private:
    //private variable/s
    double midpoint{};
    const size_t M{ 101 };
    const size_t N{ 101 };
    std::vector<std::vector<double>> u{};
    std::vector<std::vector<double>> v{};

    double beta;
    const double h;
    const double k;
};

class ParDiffEqBurgersImplicit
{
public:
    //public constructor/s
    ParDiffEqBurgersImplicit(double h, double k);
    //public function/s
    void Implicit();
private:
    //private function/s
    void SetVectorb(const size_t column);
    //private variable/s
    double midpoint{};
    const size_t M{ 101 };
    std::vector<std::vector<double>> u{};
    std::vector<double> b{};

    double beta;
    const double h;
    const double k;
};
```

```cpp
//Burgers.cpp

#include "0.0.Utility.h"
#include "0.3.Tri.h"
#include "4.Burgers.h"

//public contstructor/s
ParDiffEqBurgersEuler::ParDiffEqBurgersEuler(double h, double k)
    : h(h), k(k)
{
    beta = ((0.5 * pow(h, 2)) / k) - 0.01; /*satisfy (beta*k)/h^2 < 0.5 for stability*/

    MidPoint(h, M, midpoint);
    IniMatrix(M, N, u);
    InitialConditions ini_inital_conditions_u(h, M, N, u, midpoint);
}
//public function/s
void ParDiffEqBurgersEuler::Euler()
{
    const double s_0{ k / (4 * h) };
    const double s_1{ (beta * k) / pow(h,2) };

    for (size_t j = 0; j < N - 1; j++)
    {
        for (size_t i = 1; i < M - 1; i++)
        {
            u[i][j + 1] = u[i][j] - s_0 * (pow(u[i + 1][j], 2) - pow(u[i - 1][j], 2)) +
            s_1 * (u[i + 1][j] - 2 * u[i][j] + u[i - 1][j]);
        }
    }

    std::string save_file_name{ "Results/Eq4/Euler_Burgers_h_" + std::to_string(h) +
    "_k_" + std::to_string(k) + "_beta_" + std::to_string(beta) + ".csv" };

    Save(h, k, M, N, u, save_file_name);
}

////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////

//public contstructor/s
ParDiffEqBurgersRK2::ParDiffEqBurgersRK2(double h, double k)
    : h(h), k(k)
{
    beta = ((0.5 * pow(h, 2)) / k) - 0.01; /*satisfy (beta*k)/h^2 < 0.5 for stability*/

    MidPoint(h, M, midpoint);
    IniMatrix(M, N, u);
    IniMatrix(M, N, v);
    InitialConditions(h, M, N, u, midpoint);
    InitialConditions(h, M, N, v, midpoint);
}
//public function/s
void ParDiffEqBurgersRK2::RK2()
{
    const double r_0{ k / (8 * h) };
    const double r_1{ (beta * k) / (2 * pow(h,2)) };

    const double s_0{ k / (4 * h) };
    const double s_1{ (beta * k) / pow(h,2) };

    for (size_t j = 0; j < N - 1; j++)
    {
        for (size_t i = 1; i < M - 1; i++)
        {
            v[i][j + 1] = u[i][j] - r_0 * (pow(u[i + 1][j], 2) - pow(u[i - 1][j], 2)) +
            r_1 * (u[i + 1][j] - 2 * u[i][j] + u[i - 1][j]);
        }
        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],
                2)) + s_1 * (v[i + 1][j + 1] - 2 * v[i][j + 1] + v[i - 1][j + 1]);
68          }
69      }
70
71      std::string save_file_name{ "Results/Eq4/RK2_Burgers_h_" + std::to_string(h) + "_k_"
        + std::to_string(k) + "_beta_" + std::to_string(beta) + ".csv" };
72
73      Save(h, k, M, N, u, save_file_name);
74  }
75
76  ////////////////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////
77
78  //public contstructor/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) +
        "_k_" + std::to_string(k) + "_beta_" + std::to_string(beta) + ".csv" };
104
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][
            column], 2));
113     }
114     b[M - 1] = u[M - 1][column];
115 }
```

```cpp
//main.cpp

#include "0.0.Utility.h"
#include "1.Eq1.h"
#include "2.Eq2.h"
#include "3.Eq3.h"
#include "4.Burgers.h"

const static double h{ 0.1 };
const static double k{ 0.005 };

void DemoParDiffEq1()
{
    ParDiffEq1Euler eq1_eul(h, k);
    eq1_eul.Euler();

    ParDiffEq1RK2 eq1_rk2(h, k);
    eq1_rk2.RK2();
}
void DemoParDiffEq2()
{
    ParDiffEq2Euler eq2_eul(h, k);
    eq2_eul.Euler();

    ParDiffEq2RK2 eq2_rk2(h, k);
    eq2_rk2.RK2();
}
void DemoParDiffEq3()
{
    ParDiffEq3Euler eq3_eul(h, k);
    eq3_eul.Euler();

    ParDiffEq3RK2 eq3_rk2(h, k);
    eq3_rk2.RK2();

    ParDiffEq3Implicit eq3_implicit(h, k);
    eq3_implicit.Implicit();
}
void DemoBurgersEq()
{
    ParDiffEqBurgersEuler Burgers_eul(h, k);
    Burgers_eul.Euler();

    ParDiffEqBurgersRK2 Burgers_rk2(h, k);
    Burgers_rk2.RK2();

    ParDiffEqBurgersImplicit Burgers_implicit(h, k);
    Burgers_implicit.Implicit();
}

void MainMenu(bool& run)
{
    size_t choice{ 0 };
    const size_t final_choice{ 4 };

    system("cls");
    do
    {
        std::cout << "0. par_u/par_t + par_u/par_x = 0, PARABOLIC" << "\n";
        std::cout << "1. par_u/par_t + u*par_u/par_x = 0, PARABOLIC" << "\n";
        std::cout << "2. par_u/par_t - beta*par_u^2/par_x^2 = 0, beta > 0 => ELIPTIC,
        beta < 0 => HYPERBOLIC" << "\n";
        std::cout << "3. Burgers' equation: par_u/par_t + u*par_u/par_x -
        beta*par_u^2/par_x^2=0 = 0 , beta > 0 => ELIPTIC, beta < 0 => HYPERBOLIC" << "\n"
        ;
        std::cout << "4. Exit" << "\n";
        std::cin >> choice;
        if (choice > final_choice)
        {
```

```
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    }
```

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 double k = 0.005, h = 0.1;
19     const int N = 1 / h + 1; // liczba punktów siatki
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 wynikiKonczowe();
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 łzosta otwarty poprawnie
53   if (!plik) {
54     std::cerr << "Nie żmona ćotworzy 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::wynikiKonczowe() {
67    theta = inicjacjaThetaX0();
68    theta = liczenieThetaOdCzasu();
69 }
```

Listing 13: transformataHopfCole.cpp