INSTYTUT FIZYKI
UNIWERSYTET ŚLĄSKI

NUMERYCZNA DYNAMIKA PŁYNÓW
SPRAWOZDANIE

Niestabilność Helmholtza

Imię i nazwisko osoby wykonującej ćwiczenie:
Andrzej Więckowski

Rok studiów, kierunek:
III, Fizyka Techniczna (MISMP)

Adres e-mail:
awiecko@us.edu.pl

Spis treści

1 Wstęp teoretyczny

1.1 Równanie Naviera Stokes'a

Równanie Naviera Stokes'a dla nieściśliwej cieczy opisanej wektorowym polem prędkości $\vec{v}(\vec{x})$ ma postać:

$$\underbrace{\frac{\partial \vec{v}}{\partial t}}_{A} + \underbrace{(\vec{v}\cdot\nabla)\vec{v}}_{B} = -\frac{1}{\rho}\nabla p + \underbrace{\nu\triangle\vec{v}}_{C} \tag{1}$$

gdzie:

- ρ – gęstość płynu;
- p – ciśnienie;
- ν – lepkość kinematyczna.

Dla cieczy nieściśliwej w każdej chwili czasu spełnione jest równanie na dywergencję pola $\vec{v}$:

$$\mathrm{div}\vec{v} = 0$$

W konwencji sumacyjnej Einsteina dywergencję pola $\vec{v}$ zapiszemy w postaci:

$$\mathrm{div}\vec{v} = \frac{\partial v_i}{\partial x_i}$$

Policzmy teraz dywergencję z równania (1). Dywergencja jest liniowym operatorem różniczkowym, policzmy dywergencję pierwszego składnika: div$A$, gdzie zamieniono kolejność pochodnych:

$$\mathrm{div}\frac{\partial \vec{v}}{\partial t} = \frac{\partial}{\partial x_i}\frac{\partial v_i}{\partial t} = \frac{\partial}{\partial t}\underbrace{\frac{\partial v_i}{\partial x_i}}_{0} = 0$$

Policzmy teraz wyraz div$B$ korzystając z wzoru na iloczyn pochodnych ($(fg)' = f'g + g'f$):

$$\mathrm{div}[(\vec{v}\cdot\nabla)\vec{v}] = \frac{\partial}{\partial x_j}\left(v_i\frac{\partial v_j}{\partial x_i}\right) = \frac{\partial v_i}{\partial x_j}\frac{\partial v_j}{\partial x_i} + v_i\frac{\partial}{\partial x_i}\underbrace{\frac{\partial v_j}{\partial x_j}}_{0} = \frac{\partial v_i}{\partial x_j}\frac{\partial v_j}{\partial x_i}$$

Policzmy teraz ostatni składnik div C:

$$\operatorname{div}(\nu\triangle\vec{v}) = \nu\frac{\partial}{\partial x_i}(\triangle\vec{v})_i = \nu\triangle\underbrace{\frac{\partial v_i}{\partial x_i}}_{0} = 0$$

Ostatecznie otrzymaliśmy równanie:

$$\frac{\partial v_i}{\partial x_j}\frac{\partial v_j}{\partial x_i} = -\frac{1}{\rho}\nabla^2 p$$

$$\triangle p = -\rho\frac{\partial v_i}{\partial x_j}\frac{\partial v_j}{\partial x_i}$$

Otrzymane równianie to równanie Poissona. Dalej korzystając z tego równania można wyprowadzić schematy do numerycznego rozwiązywania równania Naviera-Sokesa.

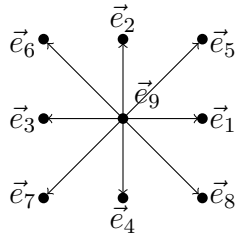
1.2 Metoda siatkowa Boltzmana

Współcześnie w komputerowej dynamice płynów (CFD) stosuje się raczej metody siatkowe. Metoda siatkowa Boltzmana (LBM), na której opierają się obliczenia w dalszej części pracy, zamiast rozwiązywać równanie Naviera-Stokesa to rozwiązywane jest dyskretne równanie transportu Boltzmana.

Równanie transportowe Boltzmana jest następujące:

$$\frac{\partial f}{\partial t} + \vec{u}\cdot\nabla f = \Omega$$

gdzie $f(\vec{x},t)$ to dystrybucja cząstek, $\vec{u}$ to prędkość cząstki, Ω operator kolizji. Metoda LBM upraszcza koncepcję Boltzmana redukując liczbę cząstek i ogarniczając je do węzłów siatki. Dla dwuwymiarowych modeli cząstka, a raczej węzeł sieci może „oddziaływać" w 9 kierunkach siatki. Taki model nazywany jest $D2Q9$ każdy węzeł zawiera 9 wektorów prędkości $\vec{e}_i$:



Rysunek 1: Schemat węzła sieci wraz z odpowiadającymi prędkościami prędkościami $\vec{e}_i$

2 Opracowanie wyników

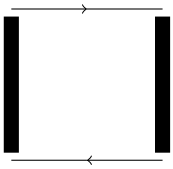
2.1 „Tutorial”

W pierwszej części zadania przyswojono podstawowe informacje odnośnie pakietu `Sailfish`. `Sailfish` jest darmową biblioteką do obliczeń komputerowej dynamiki płynów bazująca na metodzie *siatek Boltzmanna*. Pakiet został zoptymalizowany pod obliczenia równoległe – w szczególności pod obliczenia `GPGPU`.

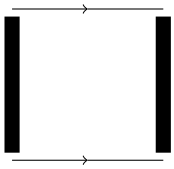
Na początku nauczono się inicjalizować warunki początkowe symulacji oraz warunki brzegowe. Cały `Sailfish` został napisany wykorzystując klasy, szkielet najbardziej prostego programu jest następujący:

```
1  class MyBlock(Subdomain2D):
2
3      def boundary_conditions(self, hx, hy):
4
5      def initial_conditions(self, sim, hx, hy):
6
7  class MySim(LBFluidSim):
8
9      subdomain = MyBlock
10
11 LBSimulationController(MySim).run()
```

W funkcji `boundary_conditions` określa się warunki brzegowe. Na potrzeby ćwiczenia zbadano następujące sytuacje:



a)



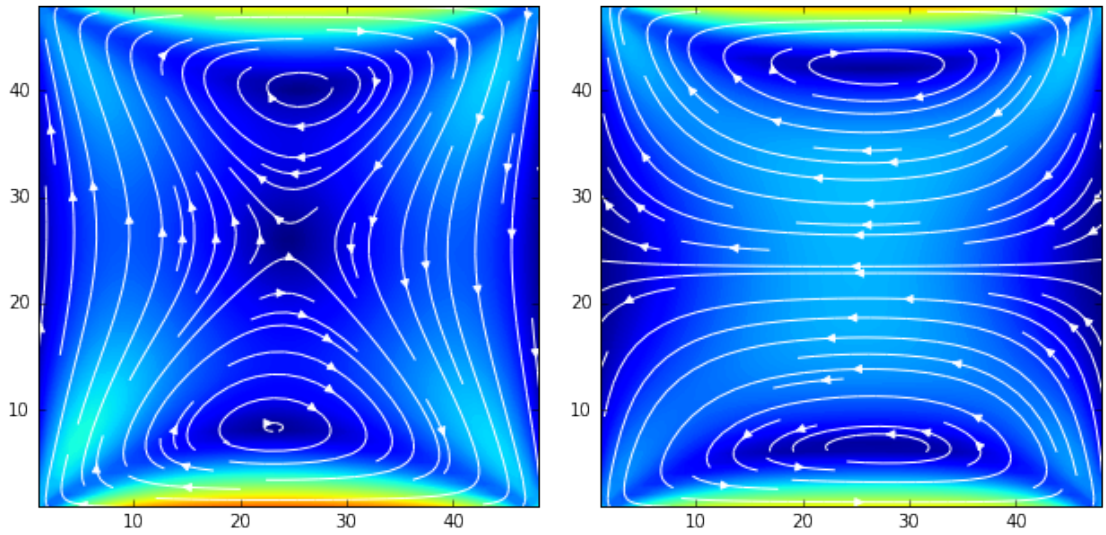
b)

Na brzegach (grube linie) ustawiono warunek `NTFullBBWall`, a na górze i dole (strzałki) ustawiono stałą prędkość w kierunku x `NTEquilibriumVelocity`. Prędkości zarówno dla przypadku a) i b) powiększono o losową liczbę z przedziału $[0, 0.1)$. w celu zaburzenia symetrii. Początkową gęstość ρ ustalono na 1 w całej objętości cieczy, lepkość na 0.3. Obliczenia dokonywano na kwadratowej siatce 50×50.

```
import numpy as np
import random as rnd
from sailfish.subdomain import Subdomain2D
from sailfish.node_type import NTFullBBWall, NTEquilibriumVelocity
from sailfish.controller import LBSimulationController
from sailfish.lb_single import LBFluidSim

class MyBlock(Subdomain2D):
    max_v = 0.1
    a = rnd.random() * 0.1
    b = rnd.random() * 0.1

    def boundary_conditions(self, hx, hy):
        wall_map = (hx==0) | (hx==self.gx-1)
        vel_up   = (hy == self.gy-1) & (hx>0) & (hx<self.gx-1)
        vel_down = (hy == 0) & (hx>0) & (hx<self.gx-1)
        self.set_node( vel_up,   NTEquilibriumVelocity(( self.max_v+
    self.a, 0.0)) )
        self.set_node( vel_down, NTEquilibriumVelocity((-self.max_v-
    self.b, 0.0)) )
        self.set_node(wall_map, NTFullBBWall)
        np.savez("datasim/hx.npz",hx=hx,hy=hy)

    def initial_conditions(self, sim, hx, hy):

        sim.rho[:] = 1.0

class MySim(LBFluidSim):

    subdomain = MyBlock

LBSimulationController(MySim).run()
```

Po wykonaniu symulacji zebrane dane przedstawiono na tak zwanym wykresie typu „stream plot". Wykorzystano narzędzie, które rozwiązywało równania różniczkowe dla danych i tym sposobem wykreślano linie pola prędkości cieczy. Wykreślono zależność $v_x^2 + v_y^2$ w funkcji położenia x, y (gdzie prędkość dana jest poprzez wektor $\vec{v} = [v_x, v_y]$).

Poniżej zamieszczono odpowiednio otrzymane rozwiązania po 100 krokach symulacji dla obu przypadków warunków początkowych: a), b).



a) b)

Dla przypadku a) uzyskane pole prędkości zorientowane jest raczej w kierunku pionowym y wyraźne są dwie struktury przypominające wiry. W przypadku b) struktura pola prędkości zorientowana jest raczej w kierunku poziomym, wyraźnie widać „podział wpływów" – prosta $y = 25$.

2.2 Zaburzenie w LBM

Bardziej jako ciekawostkę zamieszczono tutaj badanie zaburzenia w LBM – propagacja dźwięku. Zaburzenie wprowadza się poprzez zmianę gęstości jednego węzła jako warunek początkowy. Tak wytworzona niejednorodność generuje falę poruszającą się z maksymalną prędkością możliwą w tym układzie (prędkość przenoszenia informacji → prędkość dźwięku). Oczywiście jest to dalekie od eksperymentu fizycznego, ponieważ zaniedbywane jest tutaj charakter adiabatyczny towarzyszący wytwarzaniu dźwięku[1], to jednak pozwala to analizować rozchodzenia się informacji w badanych układach oraz weryfikację symulacji – w naszym modelu ta prędkość powinna wynosić $\frac{1}{\sqrt{3}}$.

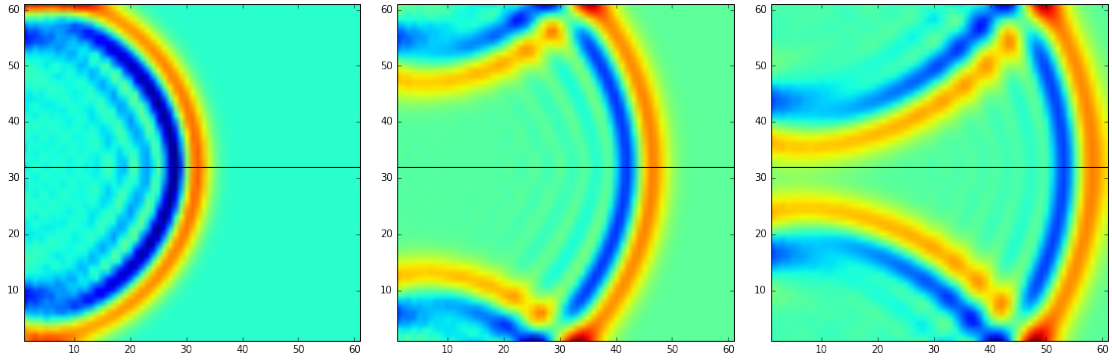
Obliczenia dokonywano na kwadratowej siatce o wymiarach 63×63, z ustawioną lepkością $\nu = 0.001$. Warunki początkowe i brzegowe ustawiono na następujące:

[1] `https://en.wikipedia.org/wiki/Sound`

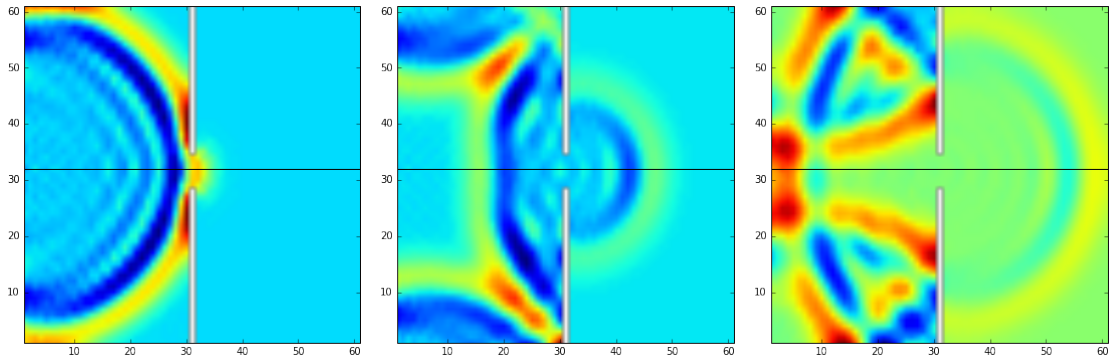
```
    def boundary_conditions(self, hx, hy):
        wall_map = ( (hy == self.gy-1) | (hx == self.gx-1)\
         | (hx==0) | (hy==0) )
        self.set_node(wall_map, NTFullBBWall)

    def initial_conditions(self, sim, hx, hy):
        nx,ny = self.gx,self.gy
        sim.rho[:] = 1.0
        sim.rho[32,1] = 1.5
```

Obserwowano również analizę przemieszczającego się zaburzenia z układem ze szczeliną i dwoma szczelinami, które zostały dodane wykorzystując do tego spójniki logiczne:

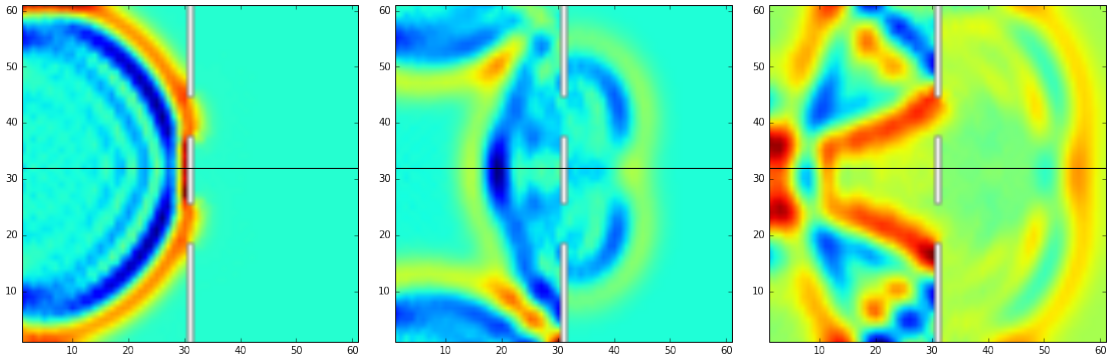
```
wall_map |= (hx == self.gx/2) & ((hy < 0.45*self.gy) | (hy > 0.55*self.gy))))
wall_map |= ((hx == self.gx/2) & ((hy < 0.3*self.gy) |
 ((hy > 0.4*self.gy) & (hy < 0.6*self.gy))|(hy > 0.7*self.gy))))
```



Ewolucja propagacji zaburzenia dla układu bez szczelin, odpowiednio 55,80,100 krok symulacji.



Ewolucja propagacji zaburzenia dla układu z jedną szczeliną, odpowiednio 55,80,100 krok symulacji.



Ewolucja propagacji zaburzenia dla układu z dwiema szczelinami, odpowiednio 55,80,100 krok symulacji.

2.3 Niestabilność Helmholtza

Niestabilność Helmholtza pojawia się kiedy mamy do czynienia z układem gdzie występuje granica z różnicą prędkości w płynie (czy też ogólniej w płynach – może to np. być znana sytuacja: wiatr wiejący nad taflą wody, czy „kłębienie" się chmur). W tej części ćwiczenia badano zjawisko niestabilność Helmholtza w zakresie czynników geometryczne lub niejednorodności warunków początkowych.

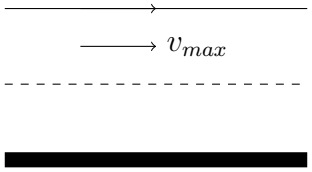
Rysunek 2: Zjawisko niestabilności Helmholtza obserwowane w chmurach
[źródło: pl.wikipedia.org]

Układ 1

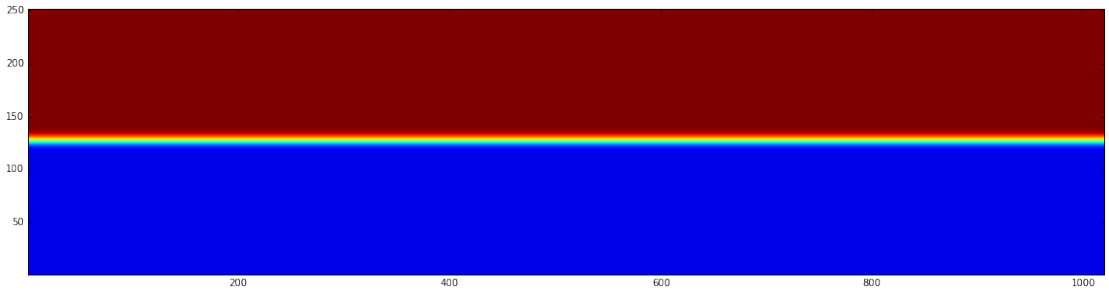
Każda badana sytuacja była pewną modyfikacją opisanego tutaj układu. Warunki brzegowe ustalono następująco:

- dolną krawędź ustawiono na sztywno jako ścianę;
- dla górnej krawędzi węzły ustawiono na stałe z prędkością w kierunku x: $\vec{v} = (v_{max}, 0)$.

Warunki początkowe ustalono nadając początkową prędkość dla wszystkich węzłów znajdujących się powyżej połowy obszaru symulacji.



Obliczenia dokonywano na prostokątnej siatce o wymiarach 1022×254, z ustawioną lepkością $\nu = 0.001$. Dla tak symetycznego układu po 10000 krokach symulacji nic szczególnego się nie dzieje. W wyniku obserwujemy jedynie powolną dyfuzję prędkości:



Rysunek 3: Wyniki symulacji dla symetrycznego układu po 10000 krokach symulacji – pole prędkości składowa x: $\vec{v}_x$. Skala wielkości: niebieski: najmniejsza, czerwony: największa wartość (wartości z przedziału [-0.01,0.1]).

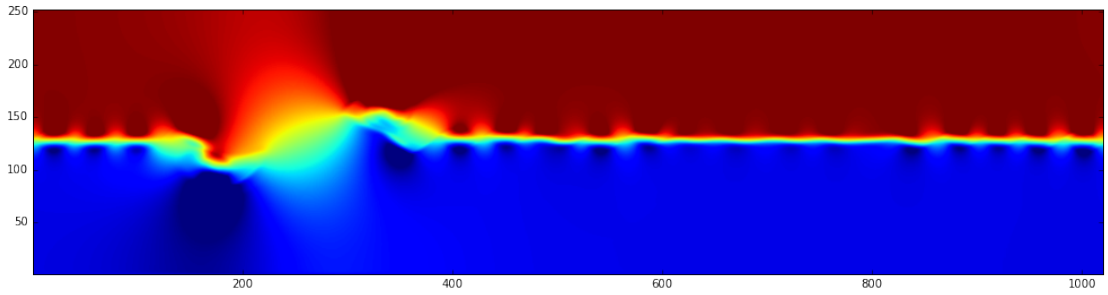
Poniżej zamieszczono kod funkcji `def boundary_conditions(self, hx, hy):` oraz `def initial_conditions(self, sim, hx, hy):`

```
    def boundary_conditions(self, hx, hy):

        wall_mapv1 = ( (hy == self.gy-1) )
        wall_mapv0 = ( (hy == 0) )

        self.set_node(wall_mapv1, NTEquilibriumVelocity((self.max_v,
    0.0)) )
        self.set_node(wall_mapv0, NTFullBBWall)

    def initial_conditions(self, sim, hx, hy):

        nx,ny = self.gx,self.gy
        sim.rho[:] = 1.0
        border = (hy>ny/2)

        sim.vx[border] = self.max_v
```

Układ 2

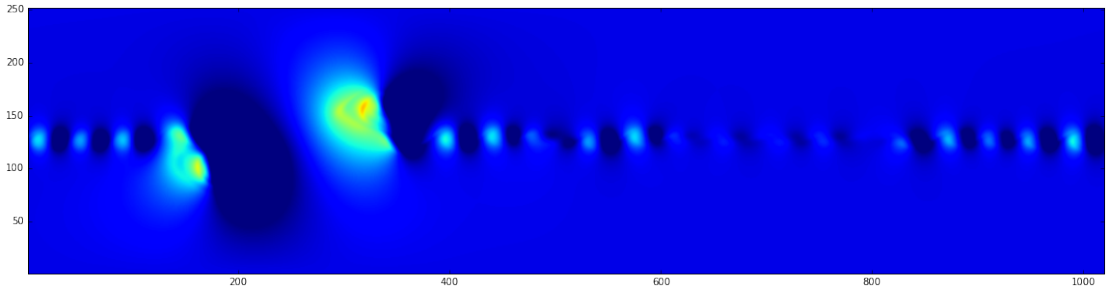
Zbadajmy co się dzieje z układem, jeśli początkowa prędkość nie będzie tak jednorodna. Ustalmy w warunkach początkowych niezerową prędkość w kierunku y dla jednego węzła w punkcie (2, n_y/2-1) (gdzie n_y to są wymiary symulacji w kierunku y) na wartość $v_y = -0.01$ (oczywiście znaki jakie podajemy na wartościach prędkości mają jedynie charakter umowny i informują nas o zwrocie kierunku wektora prędkości).

```
    def initial_conditions(self, sim, hx, hy):
        nx,ny = self.gx,self.gy
        sim.rho[:] = 1.0
        border = (hy>ny/2)
        sim.vx[border] = self.max_v
        sim.vy[(hx==2)&(hy==ny/2-1)] = -0.01
```



Rysunek 4: Wyniki symulacji dla układu z jednym węzłem ustawionym inaczej niż w standardowej wersji układu po 5000 krokach symulacji – pole prędkości składowa x: $\vec{v}_x$. Skala wielkości: niebieski: najmniejsza, czerwony: największa wartość (wartości z przedziału [-0.01,0.1])

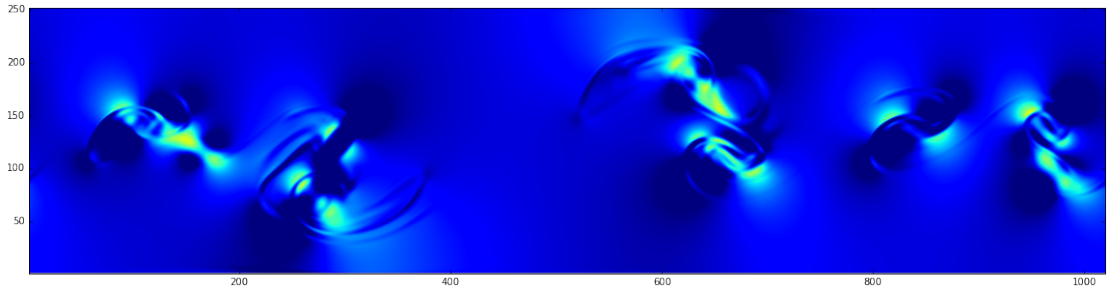
W odróżnieniu od przypadku z układu 1 tutaj można również sensownie wykreślić pole prędkości dla kierunku y: $\vec{v}_y$:



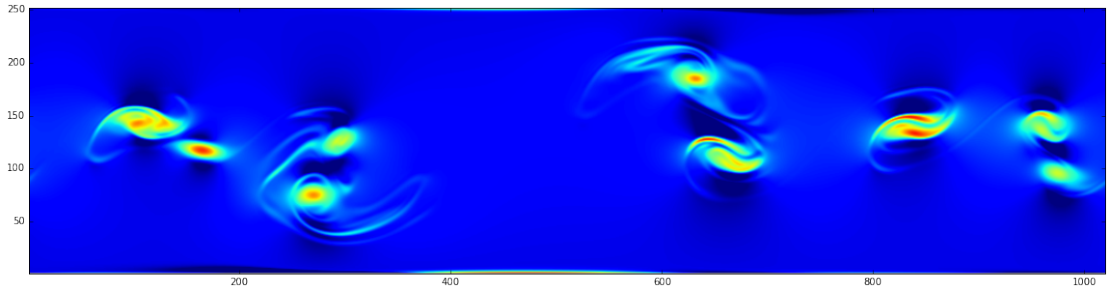
Rysunek 5: Wyniki symulacji dla układu z jednym węzłem ustawionym inaczej niż w standardowej wersji układu po 5000 krokach symulacji – pole prędkości składowa y: $\vec{v}_y$. Skala wielkości: niebieski: najmniejsza, czerwony: największa wartość (wartości z przedziału [-0.01,0.1])

Oprócz wyznaczanych pól prędkości $\vec{v}_x$, $\vec{v}_y$ wykreślono inne pola. Wyznaczano gradient (z pakietu `numpy`):

```
Dxvx,Dyvx = np.gradient(vx)
Dxvy,Dyvy = np.gradient(vy)
```



Rysunek 6: Wyniki symulacji dla układu z jednym węzłem ustawionym inaczej niż w standardowej wersji układu po 9000 krokach symulacji – gradient w kierunku y pola prędkości składowej x pomniejszone o gradient w kierunku x pola prędkości skladowej y: $D_y v_x - D_x v_y$. Skala wielkości: niebieski: najmniejsza, czerwony: największa wartość (wartości z przedziału [-0.001,0.01]).

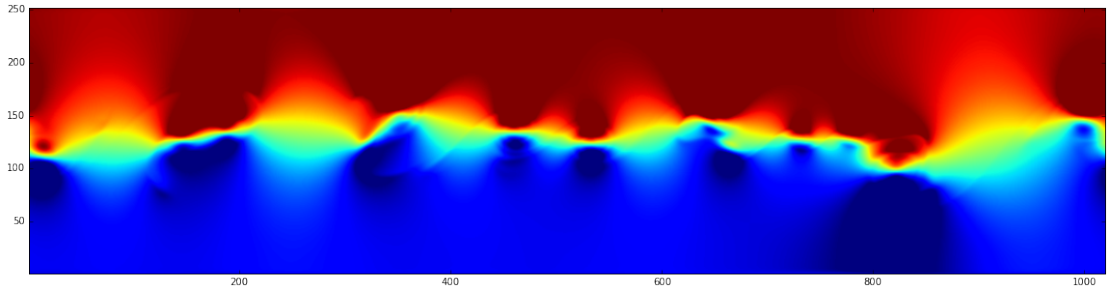


Rysunek 7: Wyniki symulacji dla układu z jednym węzłem ustawionym inaczej niż w standardowej wersji układu po 9000 krokach symulacji – gradient w kierunku x pola prędkości składowa x: $D_x v_x$. Skala wielkości: niebieski: najmniejsza, czerwony: największa wartość (wartości z przedziału [-0.001,0.01]).

Układ 3

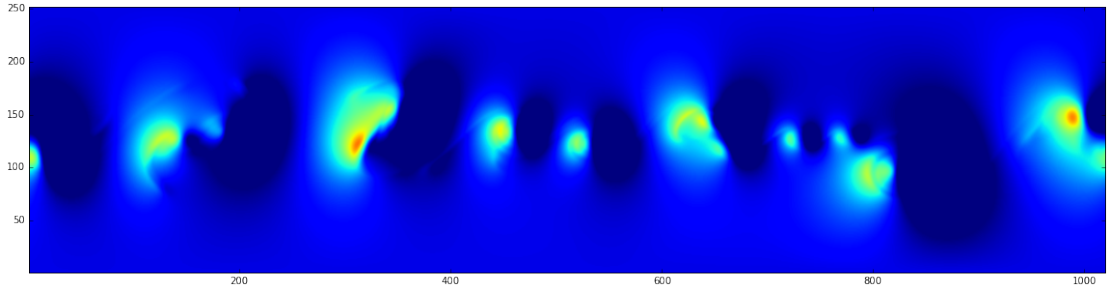
W układzie 3 niejednorodność wprowadzono poprzez narzucenie niesymetrycznych warunków początkowych. Granicę płynu w połowie wysokości oddzielającą początkową prędkość v_{max} od prędkości zerowej, zmodulowano funkcją cosinus:

```
border = (hy>2*np.cos(hx/float(nx)*2*np.pi)+ny/2)
```

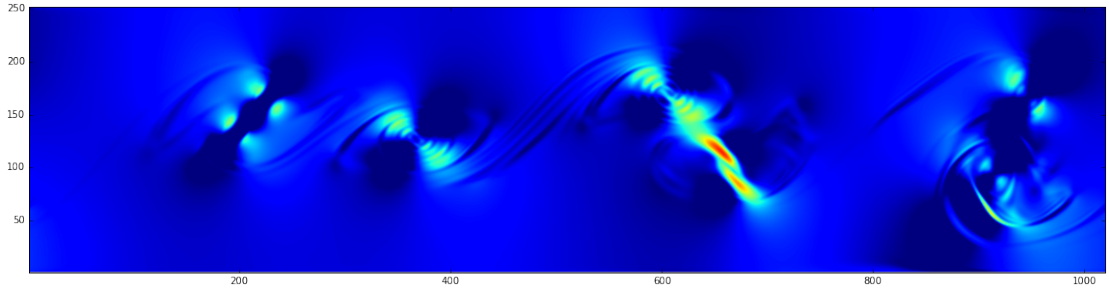


Rysunek 8: Wyniki symulacji dla układu ze zmienioną geometrią początkową po 5000 krokach symulacji – pole prędkości składowa x: $\vec{v}_x$. Skala wielkości: niebieski: najmniejsza, czerwony: największa wartość (wartości z przedziału [-0.01,0.1])

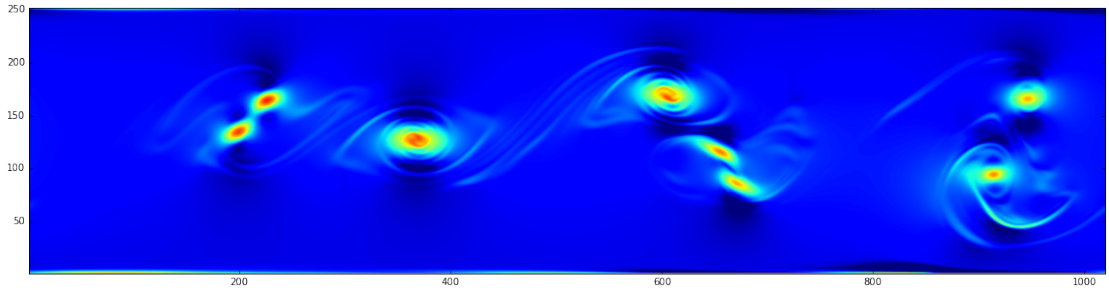
Pojawiły się zdecydowanie większe niestabilności w odróżnieniu od układu 2 po takim czasie symulacji.



Rysunek 9: Wyniki symulacji dla układu ze zmienioną geometrią początkową po 5000 krokach symulacji – pole prędkości składowa y: $\vec{v}_y$. Skala wielkości: niebieski: najmniejsza, czerwony: największa wartość (wartości z przedziału [-0.01,0.1])



Rysunek 10: Wyniki symulacji dla układu ze zmienioną geometrią początkową po 9000 krokach symulacji – gradient w kierunku x pola prędkości składowa x: $D_x v_x$. Skala wielkości: niebieski: najmniejsza, czerwony: największa wartość (wartości z przedziału [-0.001,0.01]).

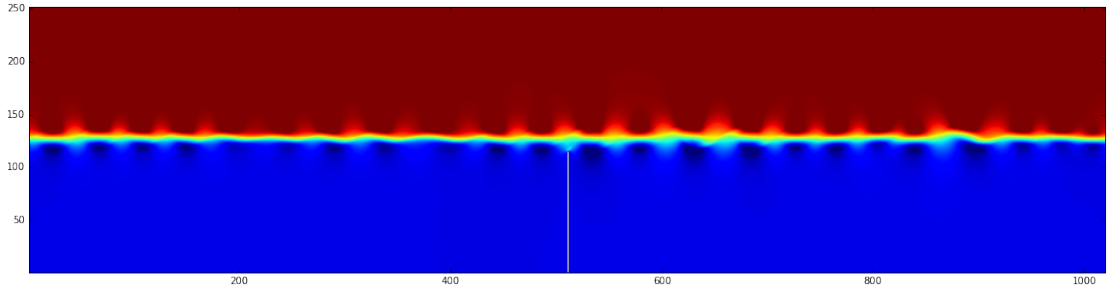


Rysunek 11: Wyniki symulacji dla układu ze zmienioną geometrią początkową po 9000 krokach symulacji – gradient w kierunku x pola prędkości składowa x: $D_x v_x$. Skala wielkości: niebieski: najmniejsza, czerwony: największa wartość (wartości z przedziału [-0.001,0.01]).

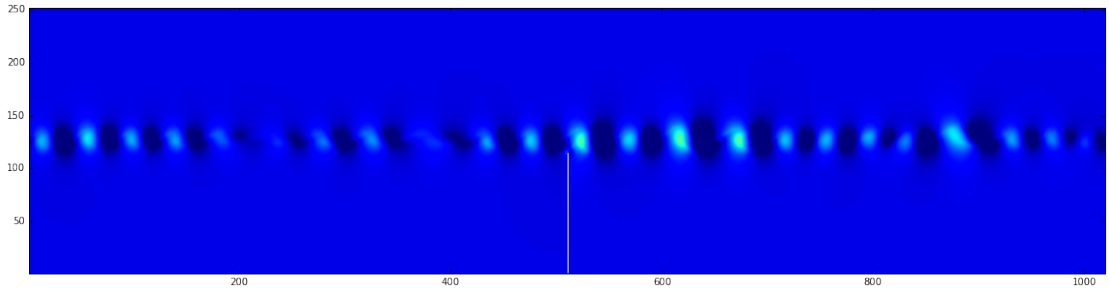
Układ 4

W ostatnim mierzonym układzie dodano ściankę w połowie układu w kierunku x na wysokość 45% wysokości układu:

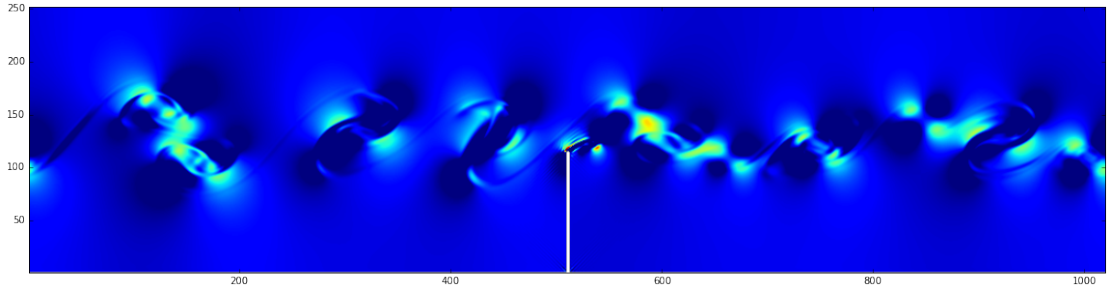
```
wall_map_obstacle = ( (hx == self.gx/2) & (hy < 0.45*self.gy) & (hy > 1))
self.set_node(wall_map_obstacle,NTFullBBWall)
```



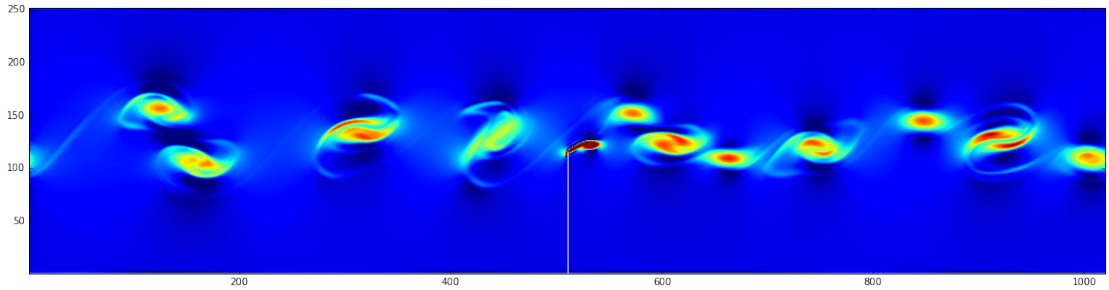
Rysunek 12: Wyniki symulacji dla układu z barierą po środku po 5000 krokach symulacji – pole prędkości składowa x: $\vec{v}_x$. Skala wielkości: niebieski: najmniejsza, czerwony: największa wartość (wartości z przedziału [-0.01,0.1])



Rysunek 13: Wyniki symulacji dla układu z barierą po środku po 5000 krokach symulacji – pole prędkości składowa y: $\vec{v}_y$. Skala wielkości: niebieski: najmniejsza, czerwony: największa wartość (wartości z przedziału [-0.01,0.1]).



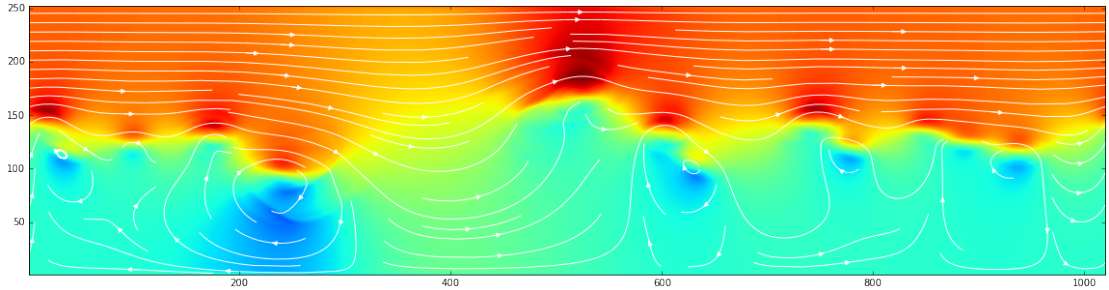
Rysunek 14: Wyniki symulacji dla układu z barierą po środku po 9000 krokach symulacji – gradient w kierunku x pola prędkości składowa x: $D_x v_x$. Skala wielkości: niebieski: najmniejsza, czerwony: największa wartość (wartości z przedziału [-0.001,0.01]).



Rysunek 15: Wyniki symulacji dla układu ze zmienioną geometrią początkową po 9000 krokach symulacji – gradient w kierunku x pola prędkości składowa x: $D_x v_x$. Skala wielkości: niebieski: najmniejsza, czerwony: największa wartość (wartości z przedziału [-0.001,0.01]).

Wszystkie wcześniejsze wykresy z tej części były rysowane za pomocą `matplotliba`, korzystając z metody `.imshow`. Metoda ta praktycznie kopiuje zawartości macierzy do bitmapy (ewentualnie dodając jakieś dodatkowe opcje wygładzenia np. wykorzystując jądro gaussa). W czasie analizy otrzymanych wyników warto czasami skorzystać z bardziej wyrafinowanych metod takich jak `.contourf` czy `.streamplot`.

Pierwsza metoda, `contourf`, wykreśla i wypełnia kontury dla wskazanych danych. Druga metoda `streamplot` wykreśla linie prądu – zaznacza na wykresie strzałki styczne do linii pola.



Rysunek 16: Wyniki symulacji dla układu z jednym węzłem ustawionym inaczej niż w standardowej wersji układu po 7500 krokach symulacji – pole prędkości składowa x: $\vec{v}_x$. Skala wielkości: niebieski: najmniejsza, czerwony: największa wartość (wartości z przedziału [-0.1,0.15]). Skorzystano tutaj z metody `contourf` z nałożonymi `streamplot`.

3 Wnioski

Wykonane ćwiczenia pozwoliły zaznajomić się z podstawowymi problemami w CFD oraz przyswoić wiedzę z zakresu obsługi biblioteki `sailfish`. Biblioteka `sailfish` jest idealnym narzędziem o zastosowaniach edukacyjnych, gdzie użytkownik dzięki prostocie obsługi jest zainteresowany tematem i chętnie bada wymyślone przez siebie zagadnienia. Dzięki wielkiej integracji z obliczeniami GPGPU oraz możliwą „klasteryzacją” liczonych zadań biblioteka znajduje się również w profesjonalnych zastosowaniach.