Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Wydział Inżynierii Metali i Informatyki Przemysłowej

# Sprawozdanie z Projektu:
# Warunki Brzegowe LBM

Przedmiot: Modelowanie Dyskretne

Kierunek: Inżynieria Obliczeniowa

Autor: Filip Rak

Prowadzący ćwiczenia: prof. dr hab. Inż. Dmytro Svyetlichnyy

Data: 1 stycznia 2025

Numer zadania: 4

Grupa laboratoryjna: 4

## Wstęp Teoretyczny

Warunki brzegowe w metodzie Lattice Boltzmann (LBM) pełnią ważną rolę w modelowaniu przepływu płynów, umożliwiając symulację interakcji z powierzchniami i granicami obszaru symulacji. Poprawna implementacja tych warunków zapewnia stabilność i realizm przepływu, uwzględniając takie aspekty, jak odbicie, przepływ wymuszony czy swobodne warunki wyjścia.

## Cel Ćwiczenia

Zapoznanie się z najczęściej stosowanymi warunkami brzegowymi dotyczącymi modelowania przepływu.

## Warunki Brzegowe

### Wariant pierwszy:

- Pionowa składowa prędkości: 0.0.
- Pozioma składowa prędkości:
    - Górna granica: 0.02.
    - Dolna granica: 0.0.
    - Prawa i lewa granica: liniowa zmiana z dołu do góry od 0.0 do 0.02.

### Wariant drugi:

- Prawa granica: warunek otwarty z gęstością: 1.0.
- Lewa granica: warunek otwarty z prędkością z dołu do góry od 0.0 do 0.02 na osi Y i prędkością 0.0 na osi X.
- Górna granica: symetryczny warunek brzegowy
- Dolna granica: warunek odbijający.

## Implementacja Warunków

Wybrana funkcja jest wywoływana pojedynczo w ramach inicjalizajci siatki oraz każdorazowo po streamingu.

### Wariant Pierwszy

```
void Automaton::apply_bc1(int x, int y)
{
	bool top = (y == 0);
	bool bottom = (y == height - 1);
	bool left = (x == 0);
	bool right = (x == width - 1);

	// Avoid inner cells
	if (!top && !bottom && !left && !right)
	{
		return;
	}

	int cell_id = grid.get_id(x, y);

	/* Apply to every boundary */

	// 1. Zero Y-axis velocity
	grid.velocity_y[cell_id] = 0.f;

	/* Apply to specific boundary */
	double max = 0.02f;
	double min = 0.f;

	if (top)
	{
		grid.velocity_x[cell_id] = max;
	}

	else if (bottom)
	{
		grid.velocity_x[cell_id] = min;
	}

	else if (left || right)
	{
		double multi = (double)y / (double)(grid.height - 1);
		multi = 1 - multi;

		grid.velocity_x[cell_id] = min + (max - min) * multi;
	}

	// Update input functions
	double u_square = grid.velocity_x[cell_id] * grid.velocity_x[cell_id] +
		grid.velocity_y[cell_id] * grid.velocity_y[cell_id];

	for (int direction = 0; direction < Grid::direction_num; direction++)
	{
		double ci_dot_u = grid.directions_x[direction] * grid.velocity_x[cell_id] +
			grid.directions_y[direction] * grid.velocity_y[cell_id];

		grid.f_in[direction][cell_id] = grid.weights[direction] * grid.density[cell_id] *
			(1.0 + 3.0 * ci_dot_u + 4.5 * ci_dot_u * ci_dot_u - 1.5 * u_square);
	}
}
```

## Wariant Drugi

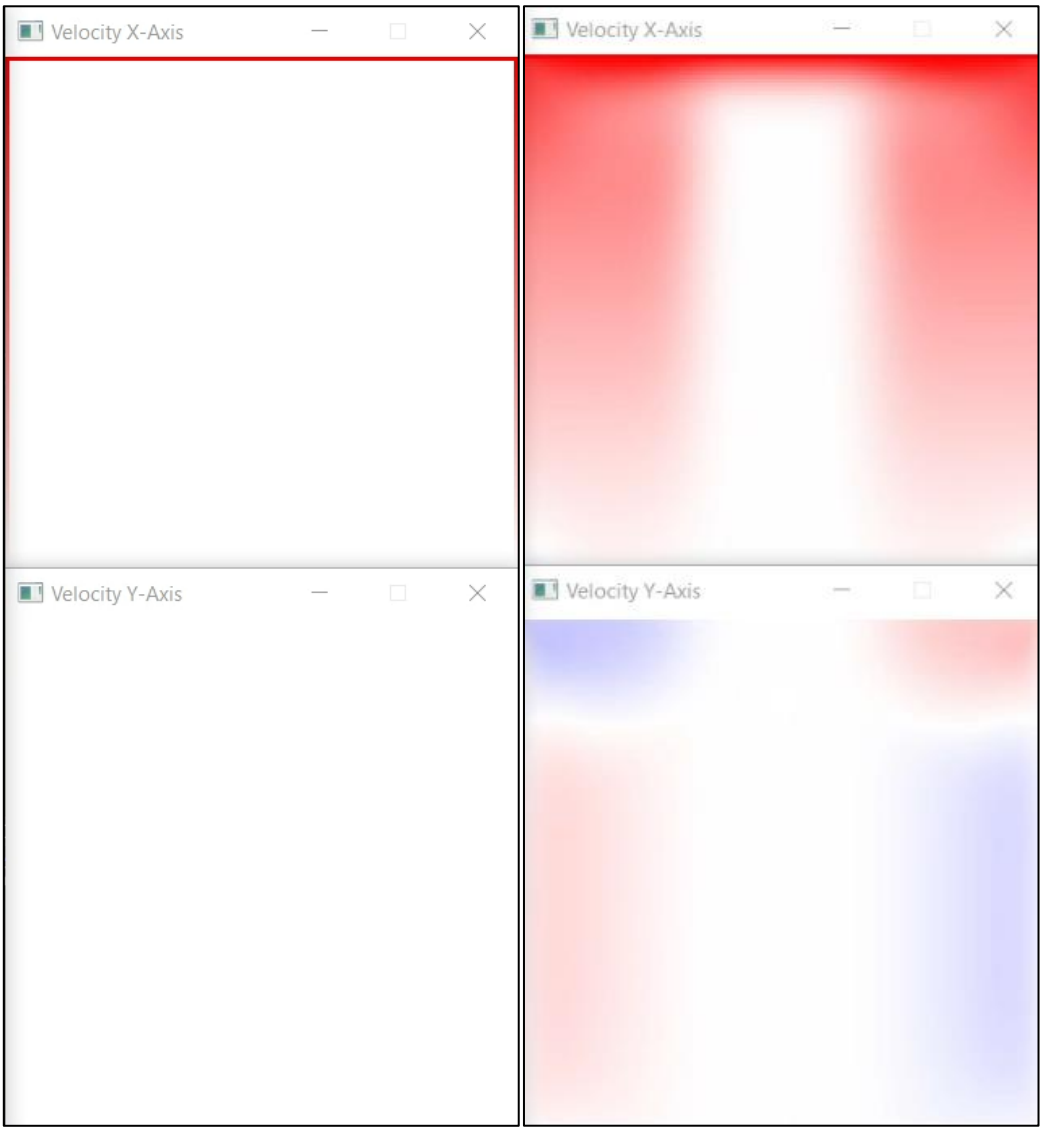
```
void Automaton::apply_bc2(int x, int y)
{
	bool top = (y == 0);
	bool bottom = (y == height - 1);
	bool left = (x == 0);
	bool right = (x == width - 1);

	// Return for inner cells
	if (!top && !bottom && !left && !right)
	{
		return;
	}

	int cell_id = grid.get_id(x, y);

	/* Bottom Boundary - Bounce Back */
	if (bottom)
	{
		for (int dir = 0; dir < Grid::direction_num; dir++)
		{
			int opp = grid.opposite_directions[dir];
			grid.f_in[opp][cell_id] = grid.f_in[dir][cell_id];
		}

		return;
	}

	/* Left Boundary - Open with Applied Speed */
	else if (left)
	{
		// Linear speed change on Vertical Axis from top (y = 0) to bottom (y = height - 1)

		// Normalize
		double normalized = (double)y / (double)(height - 1);
		normalized = 1.0 - normalized;

		// Values to apply
		double Ux = 0.02f * normalized;
		double Uy = 0.f;
		double rho = 1.f;  // Assume density of 1 at the inflow

		// Apply
		grid.density[cell_id] = rho;
		grid.velocity_x[cell_id] = Ux;
		grid.velocity_y[cell_id] = Uy;
	}

	/* Top Boundary - Symmetric */
	else if (top)
	{
		grid.f_in[4][cell_id] = grid.f_in[3][cell_id];	// 4a
		grid.f_in[8][cell_id] = grid.f_in[5][cell_id];	// 5a
		grid.f_in[7][cell_id] = grid.f_in[6][cell_id];	// 6a

		grid.velocity_y[cell_id] = 0.0;

		return;
	}

	/* Right Boundary - Open with Applied Density = 1.0 */
	else if (right)
	{
		// Apply density
		grid.density[cell_id] = 1.0;

		// 12a
		double u_x = (grid.f_in[0][cell_id] + grid.f_in[3][cell_id] + grid.f_in[4][cell_id]
			+ 2 * (grid.f_in[1][cell_id] + grid.f_in[5][cell_id] + grid.f_in[8][cell_id]));
		u_x /= grid.density[cell_id];
		u_x -= 1;

		grid.velocity_x[cell_id] = u_x;
		grid.velocity_y[cell_id] = 0.f;
	}

	/* Update Input Functions */
	for (int j = 0; j < grid.direction_num; j++)
	{
		double ci_dot_u = grid.directions_x[j] * grid.velocity_x[cell_id] +
			grid.directions_y[j] * grid.velocity_y[cell_id];

		double u_square = grid.velocity_x[cell_id] * grid.velocity_x[cell_id] +
			grid.velocity_y[cell_id] * grid.velocity_y[cell_id];

		// Equlibrium function
		double f_eq = grid.weights[j] * grid.density[cell_id] *
			(1.0 + 3.0 * ci_dot_u + 4.5 * ci_dot_u * ci_dot_u - 1.5 * u_square);

		// Initialize f_in as equlibrium function
		grid.f_in[j][cell_id] = f_eq;
	}
}
```

# Przeprowadzone Symulacje

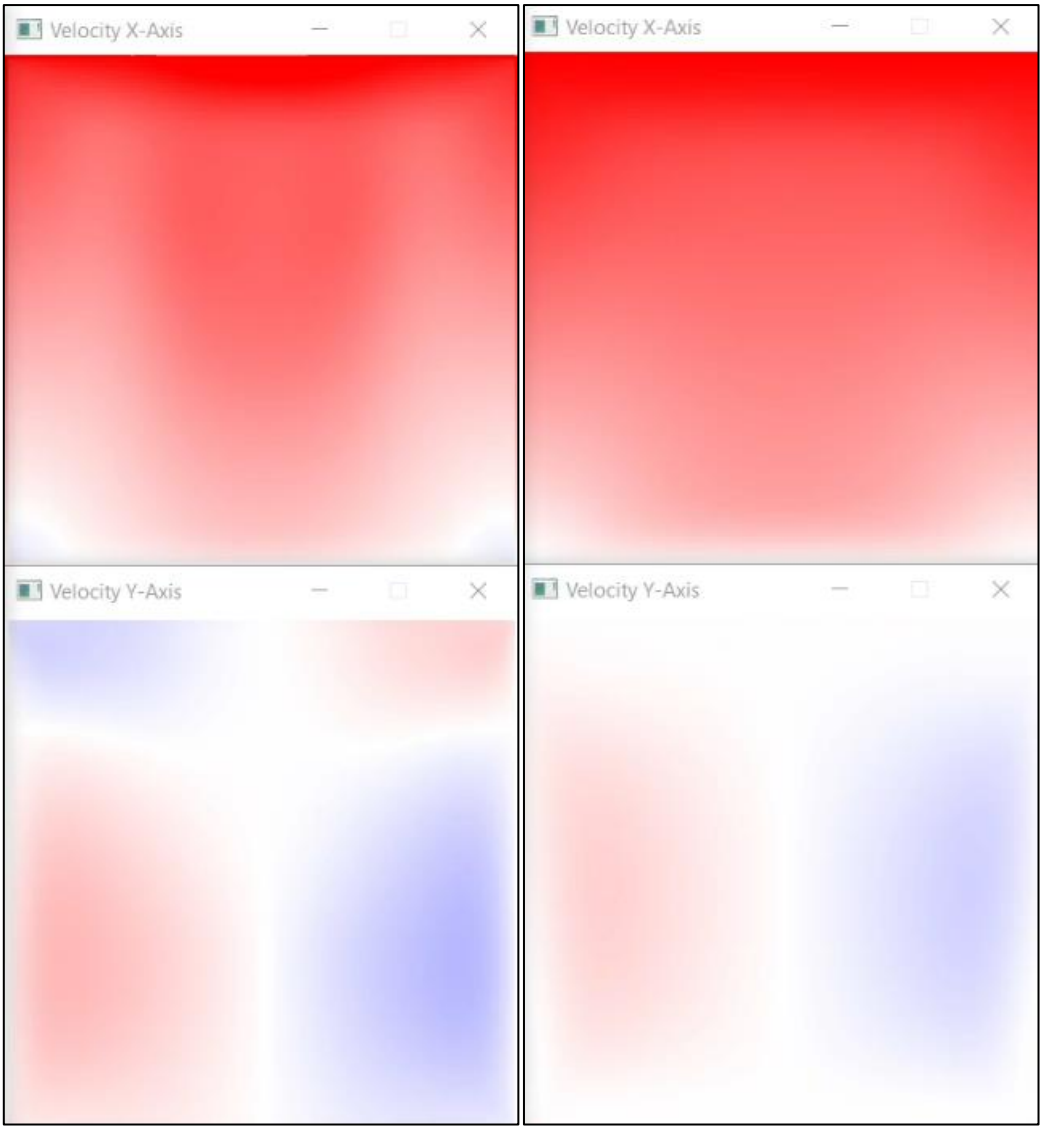
## Warunki Początkowe

- Obszar siatki: 128x128
- Gęstość: 1.0
- Prędkość: 0.0 (nie odnosi się do brzegów)

## Wariant Pierwszy

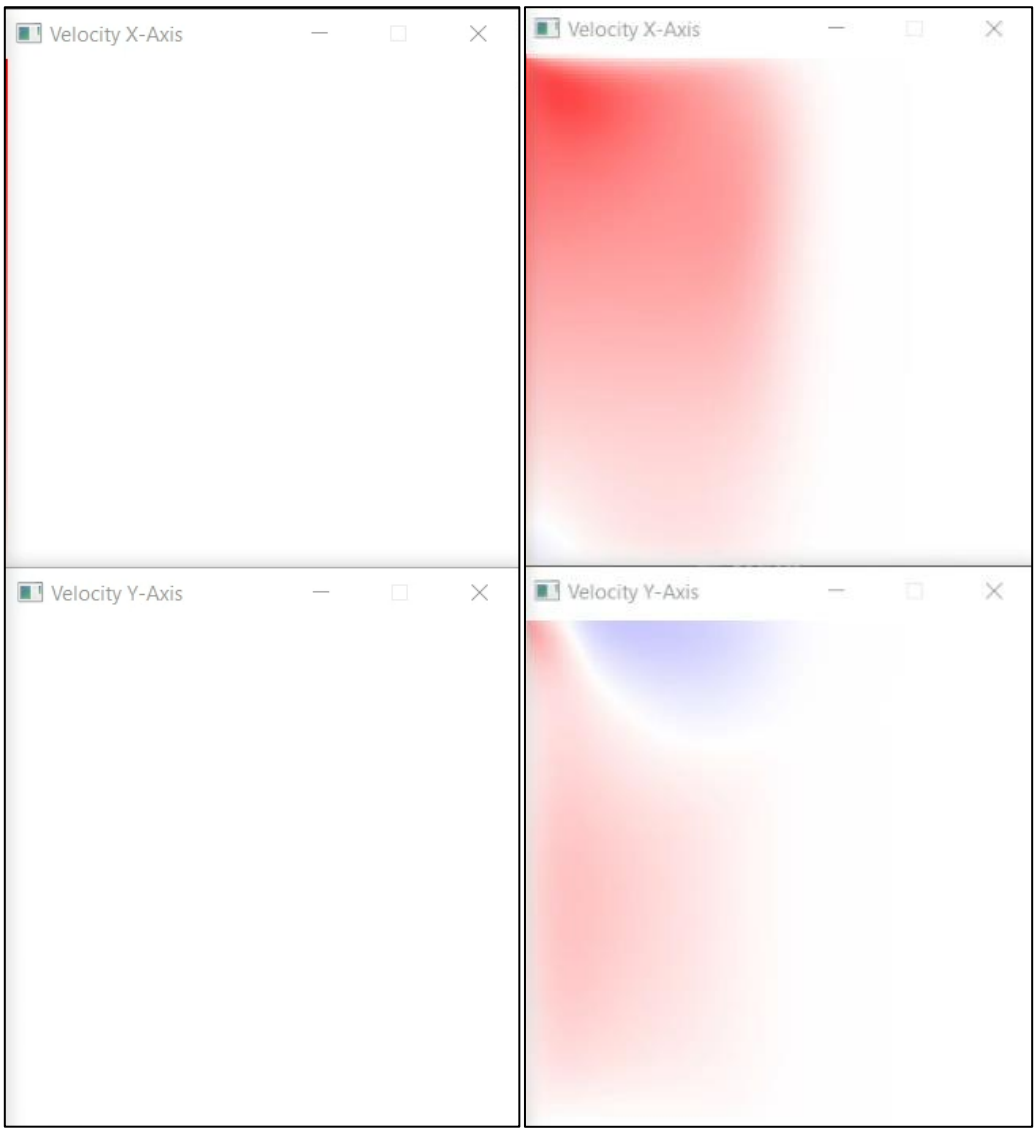



*Stan początkowy, iteracja 0 po lewej. Iteracja 68 po prawej.*

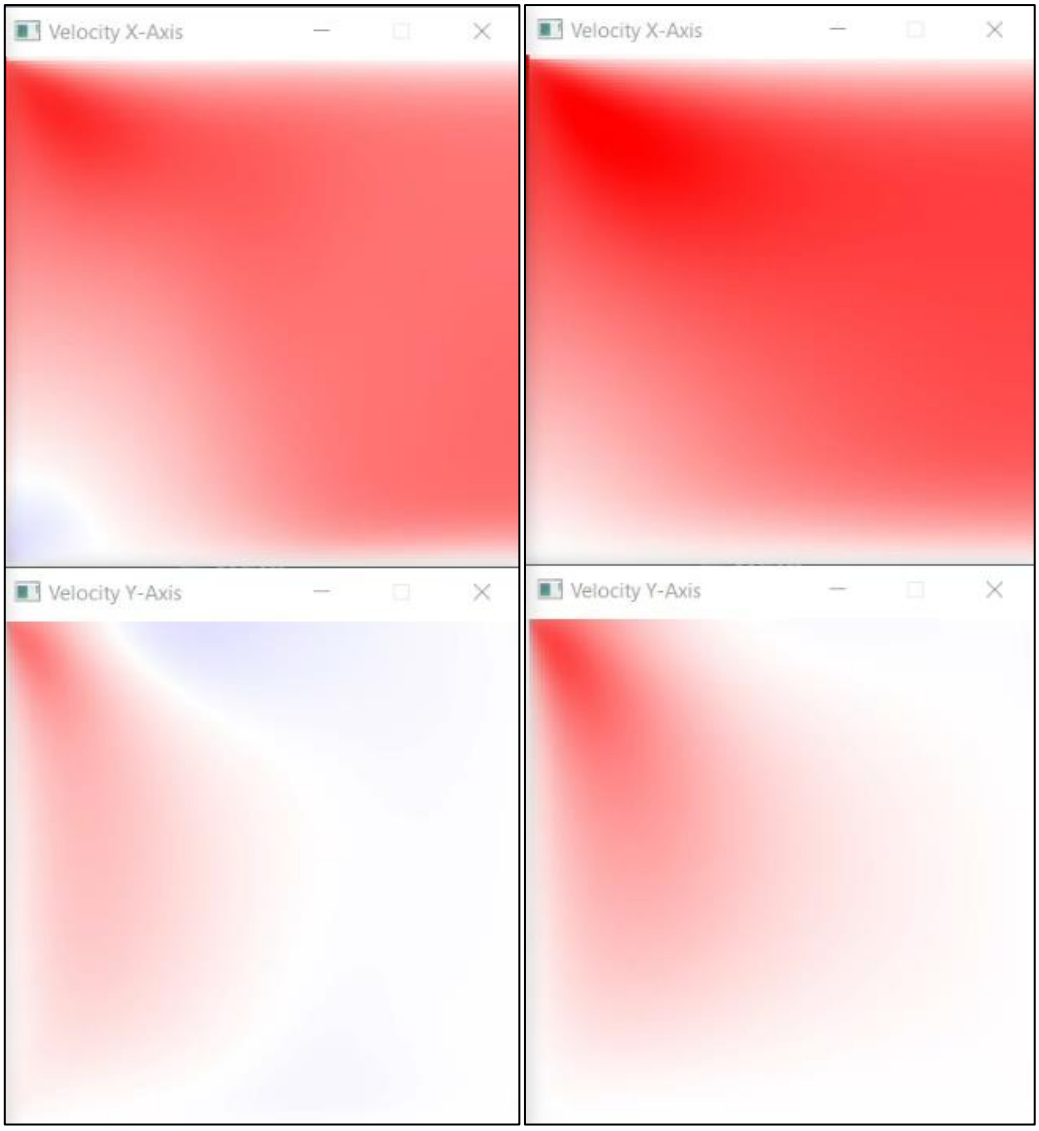



*Iteracja 157 po lewej. Stan ustalony (iteracja 278) po prawej.*

## Wariant Drugi




*Stan początkowy, iteracja 0 po lewej. Iteracja 106 po prawej.*




*Iteracja 256 po lewej. Stan ustalony (iteracja 574) po prawej.*

Nagranie

## Podsumowanie

Pierwszy wariant wydaje się być zgodny z materiałem referencyjnym przekazanym przez prowadzącego. Wyniki symulacji odzwierciedlają oczekiwane zachowanie modelu.

W przypadku drugiego wariantu trudno jednoznacznie stwierdzić, czy uzyskany rezultat jest w pełni poprawny. Pomimo dużej ilości poświęconego czasu i wielokrotnych zmianach w kodzie źródłowym oraz wielu testach nie udało się dokładnie odtworzyć rezultatu z materiału referencyjnego w okolicach górnego brzegu.