



ESCOLA TÈCNICA SUPERIOR D'ENGINYERIES INDUSTRIAL I  
AERONÀUTICA DE TERRASSA

PROYECTO FINAL DE CARRERA

**ESTUDIO DE FENÓMENOS DE TRANSFERENCIA  
DE CALOR Y DINÁMICA DE FLUIDOS MEDIANTE  
LOS MÉTODOS DE LATTICE BOLTZMANN Y  
VOLÚMENES FINITOS**

---

AUTOR: TIGRAN SARGSYAN

DIRECTOR: FRANCESC XAVIER TRIAS MIQUEL, ASENSI OLIVA  
LLENA

INGENIERÍA AERONÁUTICA

CURSO 2011-2012

## **Resumen**

El trabajo se divide en tres partes fundamentales. En la primera parte (capítulo 1) se tratan las ecuaciones que rigen la dinámica de fluidos, que son las de Navier-Stokes. Se presentan las adimensionalizaciones de las ecuaciones de Navier-Stokes para problemas de convección natural y convección forzada, identificando en cada caso los grupos adimensionales que rigen el problema. La segunda parte (capítulo 2 y 3) trata sobre los métodos numéricos LBM (Lattice Boltzmann Method) y FVM (Finite Volume Method). En cada caso se presentan el fundamento teórico sobre el que se basa y también su implementación a nivel esquemático. Por último, en la tercera parte (capítulo 4) se presenta la aplicación práctica (numérica) de LBM y FVM en problemas de referencia (los cuales sirven para validar nuevas herramientas de simulación). Se discuten los resultados obtenidos de ambos métodos y se comparan entre ellos. En los últimos capítulos se presentan las conclusiones (capítulo 5), el impacto medioambiental (capítulo 6), el presupuesto (capítulo 7) y también diferentes temas que son importantes pero han quedado fuera de alcance de este proyecto (capítulo 8). En los anexos se presentan los códigos fuentes de los programas.

# Índice general

0.1. Objetivo . . . . .	7
0.2. Justificación . . . . .	7
0.3. Alcance . . . . .	7
<b>1. Las ecuaciones de Navier Stokes</b>	<b>9</b>
1.1. Ecuaciones integrales de conservación . . . . .	9
1.2. Forma diferencial de las ecuaciones de conservación . . . . .	15
1.2.1. Convección forzada . . . . .	19
1.2.2. Convección natural . . . . .	21
<b>2. Lattice Boltzmann Method (LBM)</b>	<b>24</b>
2.1. La Ecuación Continua de Boltzmann . . . . .	24
2.2. Modelos de LBM . . . . .	35
2.2.1. Single Relaxation Time (SRT) . . . . .	35
2.2.2. Multiple Relaxation Time (MRT) . . . . .	44
2.3. Aspectos de implementación . . . . .	46
2.3.1. Tratamiento de las condiciones de contorno . . . . .	51
2.3.2. Evaluación de fuerzas . . . . .	58
2.3.3. Adimensionalización . . . . .	59
2.3.4. Aspectos de implementación paralela . . . . .	59
<b>3. Finite Volume Method (FVM)</b>	<b>61</b>
<b>4. Casos prácticos</b>	<b>78</b>
4.1. Lid Driven Cavity . . . . .	78
4.2. Differentially Heated Cavity . . . . .	88
<b>5. Conclusiones</b>	<b>99</b>

6. Impacto medioambiental	100
7. Presupuesto	101
8. Futuros estudios	102
A. Código fuente de LBM	106
B. Código fuente de FVM	133
C. Código fuente de Driven Cavity	144
D. Código fuente de Differentially Heated Cavity	153

# Índice de figuras

1.1.	Hipótesis del medio continuo . . . . .	9
1.2.	Sistema aislado . . . . .	10
1.3.	Sistema cerrado . . . . .	12
1.4.	Sistema abierto . . . . .	13
1.5.	Flujos de masa para un volumen de control 2D . . . . .	16
1.6.	Flujos de momento lineal en la dirección x para un volumen de control 2D y las tensiones . . . . .	18
2.1.	Espacio de fases . . . . .	25
2.2.	Modelo D2Q9 . . . . .	38
2.3.	Ejemplo de discretización D2Q9 . . . . .	47
2.4.	Método push de propagación . . . . .	48
2.5.	Método pull de propagación . . . . .	48
2.6.	Método de propagación utilizando una única matriz . . . . .	49
2.7.	El estado postpropagación . . . . .	49
2.8.	Ejemplo de un contorno horizontal . . . . .	52
2.9.	Ejemplo de un contorno esquina . . . . .	54
3.1.	Proyección del término convectivo/difusivo mediante el gradiente de presiones . . . . .	63
3.2.	Ejemplo de una malla rectangular uniforme . . . . .	65
3.3.	Ejemplo de una malla rectangular no uniforme . . . . .	66
3.4.	Definición de la malla . . . . .	66
3.5.	Ejemplo de colocación de las variables . . . . .	67
3.6.	Función de proyección . . . . .	69
3.7.	Definición del volumen de control para evaluar la ecuación de conti- nuidad . . . . .	69

3.8. Solver Gauss-Sheidel . . . . .	71
3.9. Solver Conjugate Gradient . . . . .	71
3.10. Definición del volumen de control para evaluar la componente $x$ de la ecuación de momentum . . . . .	72
3.11. Definición del volumen de control para evaluar la componente $y$ de la ecuación de momentum . . . . .	73
3.12. Algoritmo de cálculo para convección forzada . . . . .	75
3.13. Algoritmo de cálculo para convección natural . . . . .	76
 4.1. Driven Cavity . . . . .	78
4.2. Tiempo de computación total, norma $L_2$ del error relativo de la componente horizontal y vertical de la velocidad en función del error de solver y error en estacionario para $N = 40$ y $Re = 100$ . . . . .	81
4.3. Frecuencia de relajación adimensional para $Re = 100$ y $Re = 1000$ . .	82
4.4. Tiempo de computación total, norma $L_2$ del error relativo de la componente horizontal y vertical de la velocidad en función de la frecuencia de relajación adimensional para $N = 40$ , $Re = 100$ con $\varepsilon_{est} = 10^{-3}$ y $\varepsilon_{est} = 10^{-6}$ . . . . .	83
4.5. Comparación de LBM y FVM para diferentes tamaños de discretización	85
4.6. Comparación de LBM y FVM para diferentes tamaños de discretización	86
4.7. Comparación de LBM y FVM para diferentes tamaños de discretización	86
4.8. Comparación de LBM (línea continua) y FVM (línea discontinua) para diferentes tamaños de discretización . . . . .	87
4.9. Heated Cavity . . . . .	88
4.10. Frecuencia de relajación adimensional para $Ra = 10^3$ , $Ra = 10^4$ , $Ra = 10^5$ y $Ra = 10^6$ . . . . .	90
4.11. Perfiles de velocidad horizontal y vertical para diferentes números de Rayleigh . . . . .	91
4.12. Perfil de temperatura para diferentes números de Rayleigh . . . . .	91
4.13. Comparación de LBM (línea continua) y FVM (línea discontinua) para el contorno de la velocidad horizontal para diferentes números de Rayleigh . . . . .	92
4.14. Comparación de LBM (línea continua) y FVM (línea discontinua) para el contorno de la velocidad vertical para diferentes números de Rayleigh	97

4.15. Comparación de LBM (línea continua) y FVM (línea discontinua) para  
la función de corriente para diferentes números de Rayleigh . . . . . 98

# Índice de tablas

2.1.	Cuadratura de Hermite de orden 3 . . . . .	37
2.2.	Velocidades discretas D2Q9 . . . . .	38
2.3.	Magnitudes características en unidades lattice . . . . .	59
4.1.	Combinaciones del error del solver y del error estacionario para $\text{Re} = 100$ y $N = 40$ . . . . .	80
4.2.	Efecto de variación de la frecuencia de relajación adimensional y del error estacionario para $\text{Re} = 100$ y $N = 40$ . . . . .	82
4.3.	Comparación de LBM y FVM para diferentes tamaños de discretización	84
4.4.	Comparación de LBM y FVM para $\text{Ra} = 10^3$ . . . . .	93
4.5.	Comparación de LBM y FVM para $\text{Ra} = 10^4$ . . . . .	94
4.6.	Comparación de LBM y FVM para $\text{Ra} = 10^5$ . . . . .	95
4.7.	Comparación de LBM y FVM para $\text{Ra} = 10^6$ . . . . .	96

## **0.1. Objetivo**

El objetivo de este trabajo es investigar, implementar y comparar mediante casos de referencia, dos métodos de simulación de dinámica de fluidos: Finite Volume Method (FVM) y Lattice Boltzmann Method (LBM).

## **0.2. Justificación**

En la actualidad, las simulaciones de dinámica de fluidos son imprescindibles en muchas industrias. Por ejemplo, en la industria aeronáutica se utilizan las herramientas de simulación de dinámica de fluidos para determinar los coeficientes de fuerzas y momentos de los modelos de diseño de las aeronaves con el fin de hacer que éstas sean más eficientes enérgicamente. En la industria de las energías renovables se utilizan para diseñar los álabes de las eólicas y hacer que éstos sean eficientes. En la industria automotora se utilizan para hacer que los coches tengan coeficientes de resistencia pequeños y, por tanto, consuman menos combustible.

La herramientas de simulación pueden estar basadas en diferentes métodos. El más extendido en el ámbito de la dinámica de fluidos es el método de volúmenes finitos (FVM en siglas inglesas). Sin embargo, en el ámbito científico hay investigaciones continuas con el fin de descubrir nuevos métodos de simulación o mejorar los existentes.

El método de volúmenes finitos es relativamente antiguo comparado con el método de Lattice Boltzmann (LBM en siglas inglesas). En los últimos años ha habido mucho movimiento en la comunidad científica en el desarrollo teórico del LBM.

## **0.3. Alcance**

Para abordar el trabajo:

- se presentarán las ecuaciones de Navier Stokes y sus adimensionalizaciones para problemas de convección natural y convección forzada;
- se presentará la teoría sobre la cual se basa LBM;
- se presentará la teoría sobre la cual se basa FVM;
- se implementarán y se compararán LBM y FVM para el caso de Driven Cavity para diferentes números de Reynolds;

- y se implementarán y se compararán LBM y FVM para el caso de Heated Cavity para diferentes números de Rayleigh.

# Capítulo 1

## Las ecuaciones de Navier Stokes

### 1.1. Ecuaciones integrales de conservación

En primer lugar, se introduce un concepto muy importante llamado hipótesis del medio continuo. La velocidad en un punto del espacio es indefinida en un medio molecular, ya que sería cero en todo el tiempo excepto cuando una molécula ocupa dicho punto, y entonces sería la velocidad de la molécula y no la velocidad media de la vecindad de las moléculas en el punto. Otro ejemplo es la densidad de un fluido. Por definición, la densidad es igual a un incremento de masa dividido por un incremento del volumen. Si el volumen escogido es muy pequeño (de escala de molécula), la densidad oscilará. Existe un volumen mínimo a partir del cual la propiedad deja de oscilar y se vuelve continuo. Esta oscilación de la densidad se representa en la Figura (1.1). De manera similar se puede hablar de la presión. En un recipiente que

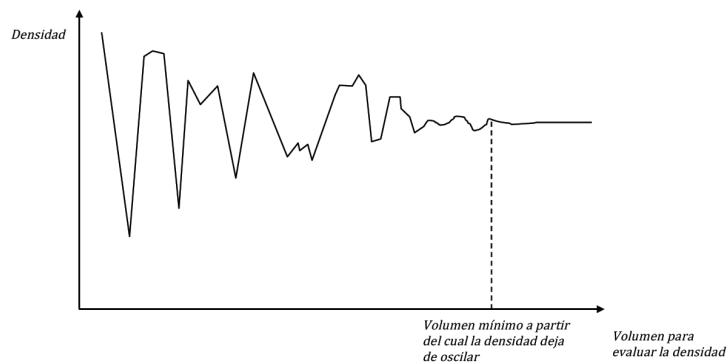


Figura 1.1: A partir de un cierto valor del volumen del elemento escogido, para evaluar la densidad, la densidad deja de oscilar.

contiene gas en equilibrio, el fluido ejerce presión en las paredes del recipiente, lo que a nivel microscópico equivale a la fuerza de choque de las moléculas con la pared del recipiente. La presión se define como la fuerza normal dividida por la superficie. Si la superficie escogida es muy pequeña, el número de moléculas que chocan con la pared del recipiente es pequeño y por tanto se contemplará el choque individual de las moléculas haciendo que la presión sea una magnitud oscilante. Sin embargo, si se escoge una superficie mayor, la presión se vuelve más continua ya que el número de moléculas que impactan en dicha superficie es elevada.

Por motivos explicados anteriormente, la estructura molecular se reemplaza por un medio hipotético llamado medio continuo. Esto supone que existen diferenciales de volumen que engloban una vecindad de moléculas. El diferencial de volumen tiene las siguientes características:

- es lo suficientemente grande como para albergar un número enorme de moléculas de forma que las fluctuaciones en las propiedades se anulen entre sí;
- es lo suficientemente pequeño como para que la propiedad pueda ser considerada local.

### Sistema aislado

Un sistema aislado es tal que no interacciona con el exterior, lo que implica que el momento lineal, el momento angular y la energía permanecen constantes a lo largo del tiempo. Sea un volumen material en el espacio que tiene un volumen

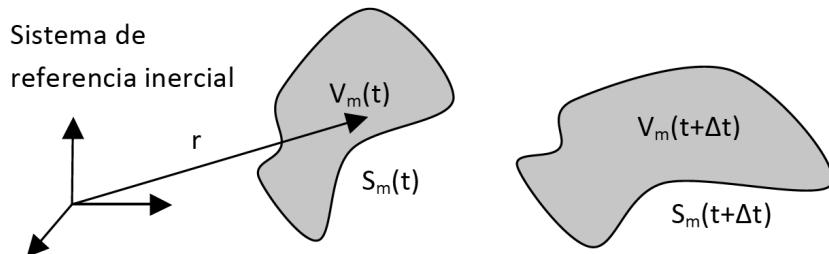


Figura 1.2: Representación de un sistema aislado en el instante de tiempo  $t$  y  $t + \Delta t$ .

$V_m(t)$  y un contorno  $S_m(t)$ . El material dentro del volumen puede estar dotado de movimiento y, por tanto el volumen como el contorno pueden ir modificándose a lo largo del tiempo. La propiedad que cumple dicho volumen material es que la masa

que contiene se mantiene constante. Las ecuaciones que describen el comportamiento del volumen material son las siguientes:

$$\frac{D}{Dt} \int_{V_m(t)} \rho dV = 0 \quad \text{Conservación de la masa} \quad (1.1a)$$

$$\frac{D}{Dt} \int_{V_m(t)} \mathbf{u} \rho dV = 0 \quad \text{Conservación del momento lineal} \quad (1.1b)$$

$$\frac{D}{Dt} \int_{V_m(t)} \mathbf{r} \times \mathbf{u} \rho dV = 0 \quad \text{Conservación del momento lineal} \quad (1.1c)$$

$$\frac{D}{Dt} \int_{V_m(t)} (\mathbf{u} + e_c) \rho dV = 0 \quad \text{Conservación de la energía} \quad (1.1d)$$

$$\frac{D}{Dt} \int_{V_m(t)} s \rho dV = \dot{S}_{gen} \geq 0 \quad \text{Segundo principio de la termodinámica} \quad (1.1e)$$

La notación  $\frac{D}{Dt}$  indica que se trata de una derivada material (sustancial) y por tanto se evalúa en un sistema Langrangiano. El planteamiento Lagrangiano fija una cierta cantidad de masa y observa qué pasa con dicho material (planteamiento de volumen material). En la ecuación de la conservación de la energía, la energía específica se separa en energía interna específica  $\mathbf{u}$  y en energía cinética específica  $e_c$  en  $[J/kg]$ . El segundo principio de la termodinámica, efectivamente, no es una ecuación de conservación.  $\dot{S}_{gen}$  es la entropía generada por unidad de tiempo, y conforme el segundo principio de la termodinámica, dicha propiedad es positiva para un proceso real y para un hipotético caso llamado proceso reversible, dicha magnitud vale cero.

### Sistema cerrado

Un sistema cerrado es tal que no intercambia masa con el exterior pero sí energía mediante flujo de calor por radiación, conducción o bien, mediante la aplicación de una fuerza distribuida sobre el contorno o en el interior del mismo material. El volumen material puede interaccionar con el exterior mediante los siguientes mecanismos:

- $\mathbf{f}_{(\mathbf{n})}$  es el vector de tensiones, fuerza por unidad de superficie actuando en el contorno del volumen material. Dicha fuerza dependerá del punto del contorno y del vector normal al contorno  $\mathbf{n}$ ;
- $\mathbf{b}$  es la fuerza interna másica, fuerza por unidad de volumen actuando dentro del volumen material. En cada punto del volumen puede tener un valor diferente. Un ejemplo de la fuerza interna es la de la gravedad que tendrá un único componente (según el criterio de selección del sistema de referencia).

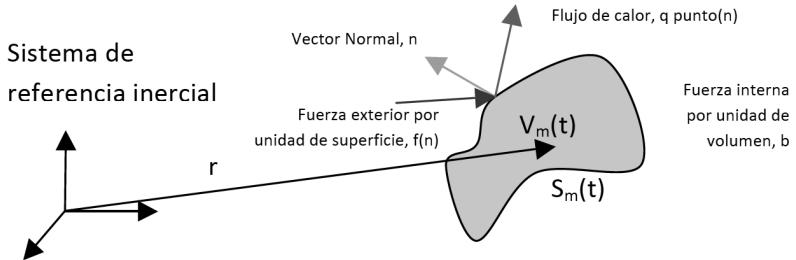


Figura 1.3: Representación de un sistema cerrado en el instante de tiempo  $t$  y sus medios de interacción.

- $\dot{q}$  es el flujo de calor por radiación o conducción a través del contorno del volumen material.

Las ecuaciones de conservación para un sistema cerrado son:

$$\frac{D}{Dt} \int_{V_m(t)} \rho dV = 0 \quad (1.2a)$$

$$\frac{D}{Dt} \int_{V_m(t)} \mathbf{u} \rho dV = \int_{S_m(t)} \mathbf{f}_{(\mathbf{n})} dS + \int_{V_m(t)} \mathbf{b} \rho dV \quad (1.2b)$$

$$\frac{D}{Dt} \int_{V_m(t)} \mathbf{r} \times \mathbf{u} \rho dV = \int_{S_m(t)} \mathbf{r} \times \mathbf{f}_{(\mathbf{n})} dS + \int_{V_m(t)} \mathbf{r} \times \mathbf{b} \quad (1.2c)$$

$$\frac{D}{Dt} \int_{V_m(t)} (\mathbf{u} + \mathbf{e}_c) \rho dV = - \int_{S_m(t)} \mathbf{q} \cdot \mathbf{n} dS + \int_{S_m(t)} \mathbf{u} \cdot \mathbf{f}_{(\mathbf{n})} dS + \int_{V_m(t)} \mathbf{u} \cdot \mathbf{b} \rho dV \quad (1.2d)$$

$$\frac{D}{Dt} \int_{V_m(t)} s \rho dV = - \int_{S_m(t)} \frac{\dot{q} \cdot \mathbf{n}}{T} dS + \dot{S}_{gen} \geq 0 \quad (1.2e)$$

En la ecuación de energía, el primer término representa el flujo de energía debido al flujo de calor por conducción / radiación. Cuando el producto escalar es positivo, significa que el flujo sale del volumen de control y, por tanto, la energía del volumen material disminuye y debe haber un signo negativo para indicar dicha disminución. El segundo término representa el trabajo por unidad de tiempo que se genera por el hecho de tener una fuerza aplicada en un punto y que dicho punto tenga una cierta velocidad. La potencia es positiva si la componente paralela de la fuerza sobre la velocidad tiene el mismo sentido que la velocidad. El tercer término es la potencia generada por las fuerzas másicas internas. Por ejemplo, en el caso de considerar la fuerza de gravedad, cuando el volumen material cae, la fuerza de gravedad produce una potencia positiva y por tanto la energía del volumen material va aumentando

con el tiempo. Si hay un intercambio de calor, habrá también un intercambio de entropía. No hay intercambio de entropía por trabajo de fuerzas externas. Con el segundo principio de la termodinámica, el término de la entropía generada sigue siendo positivo, es decir la entropía generada puede aumentar o, en el caso de proceso reversible, mantenerse constante, pero nunca puede disminuir.

### Sistema abierto

Todas las ecuaciones de mecánica y termodinámica deducidas anteriormente se refieren a volumen material. Es necesario deducirlas para un sistema abierto donde sí que pueda haber intercambio de masa. Un volumen de control es una zona del espacio arbitraria que tiene una frontera y un volumen que pueden ir modificándose de manera deseada. A diferencia del volumen material, el volumen de control no está asociado a ninguna masa de material fija, sino a una cierta zona del espacio. Dicho planteamiento es Euleriano, es decir, se fija una zona del espacio y se observa qué pasa en dicha zona. Para poder continuar es necesario introducir el Teorema de Transporte

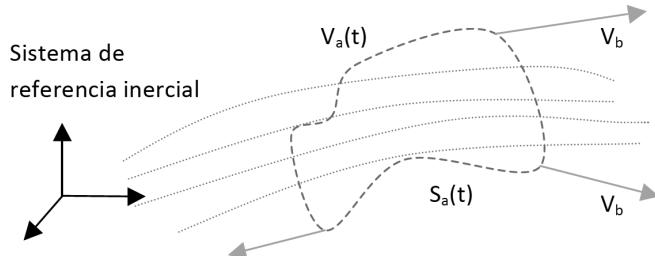


Figura 1.4: Representación de un sistema abierto en el instante de tiempo  $t$  en la que se ve cómo la materia atraviesa a través de su frontera.

de Reynolds. Se considera un volumen material y un volumen de control que en un instante genérico  $t$  coinciden. En un instante posterior  $t + \Delta t$ , el volumen material se mueve e incluso cambia su volumen debido a que el fluido que contiene puede ser compresible, mientras que el volumen de control también puede ir modificándose con una velocidad  $u_b$  (definida en todos los puntos del contorno del volumen de control). Se define una magnitud extensiva  $\Phi$ ;  $\phi$  es la misma magnitud por unidad de masa del fluido (dicha magnitud puede ser masa, momento lineal, energía, etc.). La relación

entre las dos magnitudes mencionadas anteriormente es la siguiente:

$$\Phi = \int_{V(t)} \phi \rho dV \quad (1.3)$$

La integral anterior se evalúa en el volumen de control en un cierto instante del tiempo  $t$ . A continuación se definen las siguientes magnitudes asociadas al volumen material y al volumen control, siendo la única diferencia entre ellas el instante y la región de integración:

$$\Phi_m = \int_{V_m(t)} \phi \rho dV \quad (1.4a)$$

$$\Phi_m^+ = \int_{V_m(t+\Delta t)} \phi \rho dV \quad (1.4b)$$

$$\Phi_a = \int_{V_a(t)} \phi \rho dV \quad (1.4c)$$

$$\Phi_a^+ = \int_{V_m(t+\Delta t)} \phi \rho dV \quad (1.4d)$$

En el instante inicial  $t$ , el volumen de control coincide con el volumen material ( $\Phi_m = \Phi_a$ ). La variación de la magnitud durante el  $\Delta t$  será:

$$\Delta\Phi_m = \Phi_m^+ - \Phi_m \quad (1.5a)$$

$$\Delta\Phi_a = \Phi_a^+ - \Phi_a \quad (1.5b)$$

A continuación se define  $\Phi_s$ , la cantidad que abandona el volumen de control, y  $\Phi_e$ , la cantidad que ingresa en el volumen de control durante el intervalo  $\Delta t$ . Teniendo estas definiciones en cuenta, se puede afirmar que:

$$\Phi_m^+ = \Phi_a^+ + \Phi_s - \Phi_e \quad (1.6)$$

Restando de los dos lados de la Ecuación (1.6)  $\Phi_m$  y  $\Phi_a$  y teniendo en cuenta que  $\Phi_m = \Phi_a$  (dado que en el instante inicial el volumen de control coincide con el volumen material), se obtiene:

$$\Delta\Phi_m = \Delta\Phi_a + \Phi_s - \Phi_e \quad (1.7)$$

A continuación se divide la ecuación anterior por  $\Delta t$  y se evalúa el límite cuando el incremento del tiempo tiende a cero.

$$\lim_{\Delta t \rightarrow 0} \frac{\Delta\Phi_m}{\Delta t} = \lim_{\Delta t \rightarrow 0} \frac{\Delta\Phi_a}{\Delta t} + \lim_{\Delta t \rightarrow 0} \frac{\Phi_s - \Phi_e}{\Delta t} \quad (1.8)$$

El primer término corresponde a la derivada Lagrangiana ya que la derivada se evalúa en el volumen material. El segundo término representa la derivada Euleriana ya que se evalúa en el volumen de control. El último término es el flujo de la cantidad  $\Phi$  a través de la superficie del volumen de control. Se obtiene la siguiente ecuación integral <sup>1</sup>:

$$\frac{D}{Dt} \int_{V_m(t)} \phi \rho dV = \frac{d}{dt} \int_{V_a(t)} \phi \rho dV + \int_{S_a(t)} \phi \rho (\mathbf{u} - \mathbf{u}_b) \cdot \mathbf{n} dS \quad (1.9)$$

A continuación, particularizando la magnitud  $\phi$  a 1,  $\mathbf{u}$ ,  $\mathbf{u} + e_c$  y  $s$ , se obtienen las ecuaciones integrales de conservación de masa, momento lineal, energía y el segundo principio de la termodinámica para un sistema abierto, respectivamente.

$$\frac{d}{dt} \int_{V_a(t)} \rho dV + \int_{S_a(t)} \rho (\mathbf{u} - \mathbf{u}_b) \cdot \mathbf{n} dS = 0 \quad (1.10)$$

$$\frac{d}{dt} \int_{V_a(t)} \rho \mathbf{u} dV + \int_{S_a(t)} \rho \mathbf{u} (\mathbf{u} - \mathbf{u}_b) \cdot \mathbf{n} dS = \int_{S_a(t)} \rho \mathbf{f}_{(\mathbf{n})} dS + \int_{V_a(t)} \rho \mathbf{b} dV \quad (1.11)$$

$$\begin{aligned} \frac{d}{dt} \int_{V_a(t)} \rho (\mathbf{u} + e_c) dV + \int_{S_a(t)} \rho (\mathbf{u} + e_c) (\mathbf{u} - \mathbf{u}_b) \cdot \mathbf{n} dS = \\ = - \int_{S_a(t)} \rho \dot{\mathbf{q}} \cdot \mathbf{n} dS + \int_{S_a(t)} \mathbf{u} \cdot \mathbf{f}_{(\mathbf{n})} dS + \int_{V_a(t)} \rho \mathbf{u} \cdot \mathbf{b} dV \end{aligned} \quad (1.12)$$

$$\frac{d}{dt} \int_{V_a(t)} \rho s dV + \int_{S_a(t)} \rho s (\mathbf{u} - \mathbf{u}_b) \cdot \mathbf{n} dS = - \int_{S_m(t)} \frac{\dot{\mathbf{q}} \cdot \mathbf{n}}{T} dS + \dot{S}_{gen} \quad (1.13)$$

## 1.2. Forma diferencial de las ecuaciones de conservación

A partir de las ecuaciones de conservación en forma integral se puede pasar a forma diferencial utilizando teoremas matemáticos (teorema de divergencia, etc.). Pero su deducción resulta ser poco intuitiva, así que se hará de una forma diferente aunque partiendo de la formulación integral. Las deducciones se harán para el caso bidimensional aunque su extensión a tres dimensiones es inmediata.

---

<sup>1</sup> La notación  $\frac{D}{Dt}$  hace referencia a que la derivada se evalúa en un volumen material, y se llama derivada material o sustancial (asociada a un volumen material), mientras que la notación  $\frac{d}{dt}$  hace referencia a que la derivada se evalúa en un volumen de control asociado a un sistema abierto, y se llama derivada total (asociada a un volumen de control). Por último, la notación de derivada parcial  $\frac{\partial}{\partial t}$  indica que la derivada se evalúa sobre un volumen de control estático.

## Conservación de la masa (continuidad)

Se escoge un volumen de control cuadrado estático centrado en una cierta zona del espacio tal y como se muestra en la Figura (1.5). A continuación se aplica la

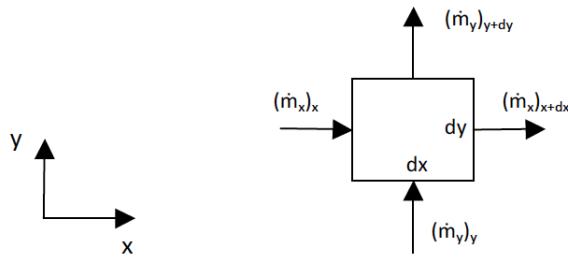


Figura 1.5: Representación de un volumen de control diferencial bidimensional con los flujos de masa.

ecuación de conservación de masa integral (1.10). Al ser el volumen de control estático  $\mathbf{u}_b = \mathbf{0}$ , la derivada total se convierte en derivada parcial. Por otro lado, al ser el volumen de control diferencial, hacer la integral es lo mismo que evaluar su valor. Se introducen los términos de flujos en las caras de entrada  $(\dot{m}_x)_x$  y  $(\dot{m}_y)_y$  y se aproximan linealmente los flujos en las caras de salida  $(\dot{m}_x)_{x+dx} = (\dot{m}_x)_x + \frac{\partial \dot{m}_x}{\partial x} dx$  y  $(\dot{m}_y)_{y+dy} = (\dot{m}_y)_y + \frac{\partial \dot{m}_y}{\partial y} dy$ . Por último, teniendo en cuenta que  $\dot{m}_x = \rho u dy dz$ ,  $\dot{m}_y = \rho v dx dz$  y  $dV = dx dy dz$ , se obtiene la ecuación diferencial de conservación de masa.

$$\frac{\partial \rho}{\partial t} + \frac{\partial (\rho u)}{\partial x} + \frac{\partial (\rho v)}{\partial y} = 0 \quad (1.14)$$

siendo  $u$  y  $v$  las dos componentes de la velocidad en el sistema de referencia inercial.

Utilizando la notación vectorial y extendiendo a un caso tridimensional, la expresión anterior se convierte en:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \quad (1.15)$$

Si la densidad no es homogénea en el espacio pero sí constante en los mismos puntos, entonces el término transitorio desaparecerá, pero al ser no homogéneo, aunque no haya variación en el valor de la densidad en los mismos puntos, sí que puede haber una variación de la densidad en el espacio y por tanto no se puede sacar la densidad del operador de divergencia. Si el campo fluido es incompresible, la densidad no variará en función del tiempo y tampoco en el espacio, y por tanto la Ecuación (1.15) se convierte en:

$$\nabla \cdot \mathbf{u} = 0 \quad (1.16)$$

## Conservación del momento lineal

Se parte de la ecuación de conservación del momento lineal (1.11) particularizada para la dirección  $x$ . El término de fuerzas superficiales se puede representar como  $\mathbf{f}_{(n)} = \mathbf{n} \cdot \boldsymbol{\sigma}$ , siendo  $\boldsymbol{\sigma}$  el tensor de tensiones que se puede separar en una parte isotrópica representando los esfuerzos normales  $-p\mathbf{I}$  (siendo  $\mathbf{I}$  la matriz identidad y  $p$  la presión hidrostática) y en una parte anisotrópica, con traza nula, representando los esfuerzos tangenciales o viscosos  $\boldsymbol{\tau}$ . De esta manera:

$$\boldsymbol{\sigma} = -p\mathbf{I} + \boldsymbol{\tau} \quad (1.17)$$

Si el fluido es Newtoniano, existe una expresión para el tensor de esfuerzos tangenciales para un problema de dimensión  $D$  (2 en el caso bidimensional y 3 en el caso tridimensional):

$$\boldsymbol{\tau} = \mu \left( \nabla \mathbf{u} + (\nabla \mathbf{u})^T \right) - \mu \frac{2}{D} (\nabla \cdot \mathbf{u}) \mathbf{I} + \mu_v (\nabla \cdot \mathbf{u}) \mathbf{I} \quad (1.18)$$

siendo  $\mu$  la viscosidad dinámica (o el primer coeficiente de viscosidad),  $\lambda$  el segundo coeficiente de viscosidad y  $\mu_v$  el coeficiente de viscosidad volumétrica definida como  $\mu_v = \lambda + \frac{2}{3}\mu$ . La divergencia del tensor de esfuerzos tangenciales será:

$$\nabla \cdot \boldsymbol{\tau} = \mu \Delta \mathbf{u} + \mu \frac{D-2}{D} \nabla (\nabla \cdot \mathbf{u}) + \mu_v \nabla (\nabla \cdot \mathbf{u}) \quad (1.19)$$

Según la hipótesis de Stokes  $\mu_v = 0$  y por tanto:

$$\nabla \cdot \boldsymbol{\tau} = \mu \Delta \mathbf{u} + \mu \frac{D-2}{D} \nabla (\nabla \cdot \mathbf{u}) \quad (1.20)$$

Se observa que el segundo término de la izquierda se anula cuando el flujo es incompresible o bien cuando se trata de un problema bidimensional.

Una vez discutido el término de fuerzas superficiales se utiliza el mismo procedimiento de evaluación de flujos (en este caso el flujo de momento lineal en la dirección  $x$  representados en la Figura (1.6)), y se obtiene la ecuación del momento lineal para la dirección  $x$ .

$$\frac{\partial(\rho u)}{\partial t} + \frac{\partial(\rho uu)}{\partial x} + \frac{\partial(\rho uv)}{\partial y} = -\frac{\partial p}{\partial x} + \frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + \rho b_x \quad (1.21)$$

Utilizando la notación vectorial y extendiendo a un caso tridimensional la expresión anterior se convierte en:

$$\frac{\partial(\rho \mathbf{u})}{\partial t} + \nabla \cdot (\rho \mathbf{u} \mathbf{u}) = -\nabla p + \nabla \cdot \boldsymbol{\tau} + \rho \mathbf{b} \quad (1.22)$$

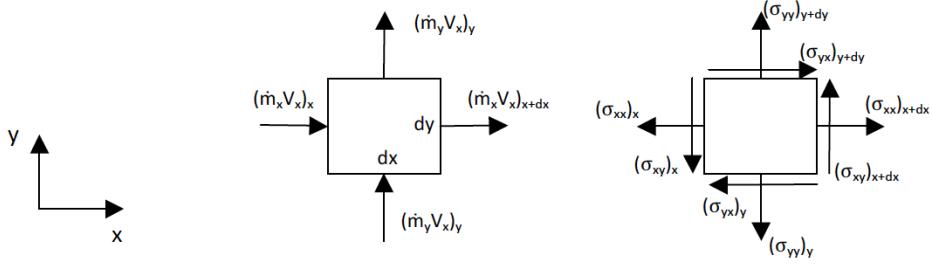


Figura 1.6: Representación de un volumen de control diferencial bidimensional con los flujos de momento lineal en la dirección  $x$ , junto a las tensiones que se ejercen en las caras del volumen de control.

Si la variación de la viscosidad es pequeña en el dominio del espacio, entonces combinando la Ecuación (1.22) con (1.19), se obtiene:

$$\frac{\partial(\rho\mathbf{u})}{\partial t} + \nabla \cdot (\rho\mathbf{u}\mathbf{u}) = -\nabla p + \mu\Delta\mathbf{u} + \frac{D-2}{D}\mu\nabla(\nabla \cdot \mathbf{u}) + \rho\mathbf{b} \quad (1.23)$$

Por otro lado, si el campo fluido es incompresible, la Ecuación (1.23) se convierte en:

$$\frac{\partial\mathbf{u}}{\partial t} + \nabla \cdot (\mathbf{u}\mathbf{u}) = -\frac{1}{\rho}\nabla p + \nu\Delta\mathbf{u} + \mathbf{b} \quad (1.24)$$

siendo  $\nu$  la viscosidad cinemática definida como  $\nu = \frac{\mu}{\rho}$ . Por último, si se desprecia el término de las fuerzas viscosas se obtiene la ecuación de Euler:

$$\frac{\partial(\rho\mathbf{u})}{\partial t} + \nabla \cdot (\rho\mathbf{u}\mathbf{u}) = -\nabla p + \rho\mathbf{b} \quad (1.25)$$

### Conservación de la energía

De manera análoga, se puede obtener la ecuación de conservación de energía diferencial a partir de la forma integral.

$$\begin{aligned} \frac{\partial(\rho\mathbf{u})}{\partial t} + \frac{\partial(\rho u\mathbf{u})}{\partial x} + \frac{\partial(\rho v\mathbf{u})}{\partial y} &= -\frac{\dot{q}_x^C}{\partial x} - \frac{\dot{q}_y^C}{\partial y} - \frac{\dot{q}_x^R}{\partial x} - \frac{\dot{q}_y^R}{\partial y} - p\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) + \\ &+ \tau_{xx}\frac{\partial u}{\partial x} + \tau_{xy}\frac{\partial v}{\partial x} + \tau_{yx}\frac{\partial u}{\partial y} + \tau_{yy}\frac{\partial v}{\partial y} \end{aligned} \quad (1.26)$$

siendo  $\dot{q}^C$  y  $\dot{q}^R$  los flujos de calor por conducción y radiación, respectivamente. Suponiendo que se trata de un gas semiperfecto ( $d\mathbf{u} = c_v dT$ ) e introduciendo la definición de la conductividad térmica  $k = -\frac{\dot{q}^C}{\nabla T}$  con unidades de  $[W/(mK)]$  se obtiene:

$$\rho c_v \frac{\partial T}{\partial t} + \rho c_v \mathbf{u} \cdot \nabla T = \nabla \cdot (k \nabla T) - \nabla \cdot \dot{\mathbf{q}}^R - p \nabla \cdot \mathbf{u} + \boldsymbol{\tau} : \nabla \mathbf{u} \quad (1.27)$$

Normalmente se desprecia el último término de la derecha de la Ecuación (1.27), que representa la disipación de energía debida a los efectos viscosos. Si el medio es transparente a la radiación, también se puede despreciar el término de transferencia de calor por radiación.

### 1.2.1. Convección forzada

Se habla de convección forzada para referirse al mecanismo de transferencia de energía y momentum producido por una fuerza externa (ventilador, bomba de succión, etc.). Dicho de otra forma, el movimiento del fluido en las condiciones de contorno vienen impuesto por una fuerza externa de velocidad o presión, y son las responsables de la generación del movimiento dentro del dominio del problema. Para la formulación matemática se va a suponer que:

- el medio es transparente a la radiación;
- la conductividad térmica es constante;
- la viscosidad dinámica es constante;
- la variación de la densidad  $\rho_{ref}$  es suficientemente pequeña como para que se pueda considerar el flujo incompresible;
- y que la única fuerza fuerza volumétrica que actúa es la gravitacional  $\mathbf{b} = g\mathbf{e}_y$ .

Teniendo todas las suposiciones anteriores en cuenta, las Ecuaciones (1.15), (1.22) y (1.22) se convierten en:

$$\nabla \cdot \mathbf{u} = 0 \quad (1.28a)$$

$$\rho_{ref} \frac{\partial \mathbf{u}}{\partial t} + \rho_{ref} (\mathbf{u} \cdot \nabla) \mathbf{u} = -\nabla p + \mu \Delta \mathbf{u} + \rho_{ref} \mathbf{g} \quad (1.28b)$$

$$\frac{\partial T}{\partial t} + (\mathbf{u} \cdot \nabla) T = a \Delta T \quad (1.28c)$$

siendo  $a = \frac{k}{\rho_{ref} c_p}$  la difusividad térmica [ $m^2/s$ ], que es una relación entre la capacidad de conducción y acumulación del calor (por ejemplo si un material o fluido tiene una difusividad térmica alta significa que transfiere el calor por conducción rápidamente).

Con el fin de disminuir la cantidad de parámetros (densidad, viscosidad, etc.) y por otro lado revelar la importancia de los diferentes términos de las ecuaciones que rigen el movimiento, se adimensionalizan las Ecuaciones (1.31). Para hacerlo se

definen las variables adimensionales:  $t^* = \frac{t}{t_0}$ , siendo  $t_0$  un tiempo característico del problema,  $\mathbf{u}^* = \frac{\mathbf{u}}{U_0}$ , siendo  $U_0$  una velocidad característica del problema,  $\mathbf{r}^* = \frac{\mathbf{r}}{L_0}$  siendo  $L_0$  una longitud característica del problema,  $p^* = \frac{p-p_0}{p_1-p_0}$ , siendo  $p_0$  y  $p_1$  dos presiones características del problema y por último  $T^* = \frac{T-T_0}{T_1-T_0}$ , siendo  $T_0$  y  $T_1$  dos temperaturas características del problema. Introduciendo las definiciones anteriores en las Ecuaciones (1.31) se obtienen:

$$\nabla^* \cdot \mathbf{u}^* = 0 \quad (1.29a)$$

$$\frac{L_0}{U_0 t_0} \frac{\partial \mathbf{u}^*}{\partial t^*} + (\mathbf{u}^* \cdot \nabla^*) \mathbf{u}^* = -\frac{p_1 - p_0}{\rho U_0^2} \nabla^* p^* + \frac{\mu}{U_0 L_0 \rho_0} \Delta^* \mathbf{u}^* - \frac{g L_0}{U_0^2} \mathbf{e}_y \quad (1.29b)$$

$$\frac{L_0}{U_0 t_0} \frac{\partial T^*}{\partial t^*} + (\mathbf{u}^* \cdot \nabla^*) T^* = \frac{a}{U_0 L_0} \Delta^* T^* \quad (1.29c)$$

El término  $\frac{L_0}{U_0 t_0}$  es el número de Strouhal  $St = \frac{L_0}{U_0 t_0}$ , y es la relación entre el tiempo de residencia (el invertido por una partícula en recorrer la longitud característica  $L_0$ , con la velocidad característica  $U_0$ ) y el tiempo característico (que podría ser, por ejemplo, el periodo de oscilación de la fuerza de sustentación que opone un cilindro dentro de un conducto). En los movimientos oscilatorios de alta frecuencia  $St \gg 1$ , el movimiento resulta ser isentrópico y con un balance entre la fuerza de inercia debida a la aceleración local y las fuerzas de presión. Los efectos viscosos quedan confinados en la llamada capa de Stokes, muy delgada frente a la longitud característica  $L_0$ . El caso límite opuesto  $St \ll 1$ , es el típico del movimiento alrededor de aviones, el cual puede tratarse como casiestacionario, si se utilizan ejes ligados al avión.

El término  $\frac{g L_0}{U_0^2}$  es la inversa al cuadrado del número de Froude  $Fr = \frac{U_0}{\sqrt{gL_0}}$  y es la relación entre la energía cinética y la gravitatoria. En el movimiento del aire alrededor de aviones y automóviles el número de Froude es  $Fr \gg 1$  y la fuerza gravitatoria es despreciable, lo que no es el caso de la hidrodinámica de buques, en la que  $Fr$  es del orden de la unidad.

El término  $\frac{\mu}{U_0 L_0 \rho_0}$  es la inversa del número de Reynolds  $Re = \frac{U_0 L_0 \rho_0}{\mu}$ , que mide la relación entre las fuerzas de inercia convectivas y las viscosas. Sirve para caracterizar los flujos laminares (Re pequeño) y turbulentos (Re grande).

El término  $\frac{p_1 - p_0}{\rho U_0^2}$  es el número de Euler  $Eu = \frac{p_1 - p_0}{\rho U_0^2}$ , y representa la relación entre la energía de presión y la energía cinética. Normalmente se utiliza para caracterizar la pérdida de carga en las tuberías o en los conductos. Cuando la diferencia de presiones está asociada a la presión de vapor, se habla del número de Cavitación  $Ca = \frac{p_0 - p_v}{\rho U_0^2}$ .

El término  $\frac{a}{U_0 L_0}$  es la inversa del número de Peclet  $\text{Pe} = \frac{U_0 L_0}{a}$  que, a su vez, es el producto de los números de Reynolds y Prandtl,  $\text{Pe} = \text{Pr} \cdot \text{Re}$ . El número de Prandtl,  $\text{Pr} = \frac{\nu}{a}$ , es la relación entre las difusividades viscosa y térmica. Si el número de Prandtl es pequeño, el espesor de la capa límite térmica es mayor que el de la capa límite de momentum.

Identificando los números adimensionales en las Ecuaciones (1.29) se obtienen:

$$\nabla^* \cdot \mathbf{u}^* = 0 \quad (1.30\text{a})$$

$$\text{St} \frac{\partial \mathbf{u}^*}{\partial t^*} + (\mathbf{u}^* \cdot \nabla^*) \mathbf{u}^* = -\text{Eu} \nabla^* p^* + \frac{1}{\text{Re}} \Delta^* \mathbf{u}^* - \frac{1}{\text{Fr}^2} \mathbf{e}_y \quad (1.30\text{b})$$

$$\text{St} \frac{\partial T^*}{\partial t^*} + (\mathbf{u}^* \cdot \nabla^*) T^* = \frac{1}{\text{Pe}} \Delta^* T^* \quad (1.30\text{c})$$

Si el problema a tratar carece de un tiempo característico, se puede eliminar el número de Strouhal definiendo el tiempo característico como  $t_0 = L_0/U_0$ . De manera análoga, si el problema carece de una diferencia de presiones característico, se puede eliminar el número de Euler adimensionalizando la presión mediante la expresión  $p_0 = \rho_0 U_0^2$ . Si se eliminan los números de Euler y Strouhal, se puede ver que los dos principales parámetros en convección forzada son los números de Reynolds y Peclet (o Prandtl).

### 1.2.2. Convección natural

Se habla de convección natural para referirse al mecanismo de transferencia de energía y momentum producido por la diferencia de densidad en el fluido que, a su vez, es causada debido al gradiente de temperatura impuesto. El fluido recibe calor y, debido a que aumenta su temperatura, la densidad baja y se eleva respecto a la masa de fluido con densidad mayor. Al elevarse la masa, el fluido de mayor densidad ocupa el lugar de la masa de fluido de menor densidad. De esta forma, se forma el flujo de convección natural. La fuerza que hace posible esto es la fuerza de flotación. Es normal pensar que, al haber una variación en la densidad, se tendrían que utilizar las ecuaciones de Navier-Stokes compresibles. Sin embargo, esto complicaría mucho la solución numérica ya que las ecuaciones se vuelven más complicadas y además se establece un acoplamiento entre la ecuación de energía y la de momentum. Para facilitar la resolución se introduce la aproximación de Boussinesq. Si la variación de la densidad no es muy grande, se pueden utilizar las mismas ecuaciones de Navier-Stokes incompresibles pero haciendo que la densidad que aparece junto al término de las fuerzas volumétricas (gravitatorias) sea función de la temperatura.

Esta aproximación puede introducir errores de orden del 1% si la diferencia entre la temperatura máxima y mínima ( $T_1 - T_0$ ) está por debajo de 2°C para agua y por debajo de 15°C para el aire [18]. Para encontrar la expresión de la variación de la densidad en función de la temperatura se define el coeficiente de expansión volumétrica a presión constante:

$$\beta = \rho \left( \frac{\partial \frac{1}{\rho}}{\partial T} \right)_{p=cte.} \quad (1.31)$$

Suponiendo que las desviaciones de la densidad y de la temperatura respecto a los valores de referencia son pequeñas, se puede aproximar la Ecuación (1.31) de la siguiente manera:

$$\beta \approx \rho \left( \frac{\frac{1}{\rho} - \frac{1}{\rho_0}}{T - T_0} \right) \implies \rho = \rho_0 (1 - \beta (T - T_0)) \quad (1.32)$$

Teniendo en cuenta la Ecuación (1.32) y definiendo el tiempo y presión característicos como  $t_0 = L_0/U_0$  y  $p_0 = \rho_0 U_0^2$ , respectivamente, se obtienen las siguientes ecuaciones adimensionales:

$$\nabla^* \cdot \mathbf{u}^* = 0 \quad (1.33a)$$

$$\frac{\partial \mathbf{u}^*}{\partial t^*} + (\mathbf{u}^* \cdot \nabla^*) \mathbf{u}^* = -\nabla^* p^* + \frac{\mu}{U_0 L_0 \rho_0} \Delta^* \mathbf{u}^* + \frac{\beta g L_0 (T - T_0)}{U_0^2} T^* \mathbf{e}_y \quad (1.33b)$$

$$\frac{\partial T^*}{\partial t^*} + (\mathbf{u}^* \cdot \nabla^*) T^* = \frac{a}{U_0 L_0} \Delta^* T^* \quad (1.33c)$$

La única novedad en la ecuación anterior es el último término de la derecha de la ecuación de momentum. Se puede ver fácilmente que el término  $\frac{\beta g L_0 (T - T_0)}{U_0^2}$  es el número de Grashof dividido por el cuadrado del número de Reynolds. El número de Grashof,  $Gr = \frac{\beta g L_0^3 (T_1 - T_0)}{\nu^2}$ , representa la relación entre la fuerza de flotación y la fuerza producida por la viscosidad. Para acabar, el número de Rayleigh se define como el producto del número de Grashof y el número de Prandtl  $Ra = Pr \cdot Gr$ .

Es evidente que en convección natural no habrá una velocidad de referencia (característico del problema)  $U_0$  y por este motivo se define la velocidad característica como  $U_0 = a/L_0$ , teniendo unidades de velocidad. Teniendo esto en cuenta, el número de Reynolds desaparece de la ecuación de momentum y el número de Peclet

también desparece de la ecuación de energía. Se obtienen las siguientes ecuaciones:

$$\nabla^* \cdot \mathbf{u}^* = 0 \quad (1.34a)$$

$$\frac{\partial \mathbf{u}^*}{\partial t^*} + (\mathbf{u}^* \cdot \nabla^*) \mathbf{u}^* = -\nabla^* p^* + \text{Pr} \Delta^* \mathbf{u}^* + \text{Ra} \cdot \text{Pr} T^* \mathbf{e}_y \quad (1.34b)$$

$$\frac{\partial T^*}{\partial t^*} + (\mathbf{u}^* \cdot \nabla^*) T^* = \Delta^* T^* \quad (1.34c)$$

Se puede ver que los dos principales parámetros en convección natural son los números de Prandtl y Rayleigh.

## Capítulo 2

# Lattice Boltzmann Method (LBM)

### 2.1. La Ecuación Continua de Boltzmann

Se puede caracterizar el estado de una partícula en el espacio mediante su vector posición  $\mathbf{r}$  y su vector momentum o velocidad  $\xi$ .

A continuación se introduce el espacio de fases  $\mu$ , que se define como un espacio 6-dimensional que está formado por las coordenadas de vector posición y vector velocidad, es decir,  $(\mathbf{r}, \xi)$ . Un punto en dicho espacio representa el estado de una partícula en un cierto instante de tiempo. Por tanto, el estado del sistema físico formado por  $N$  partículas en un cierto instante de tiempo quedará caracterizado por  $N$  puntos en el espacio de fases. Las partículas interactúan entre ellas y, por tanto, a medida que avanza el tiempo, los  $N$  puntos en el espacio de fases irán moviéndose.

La idea principal de Boltzmann era que, en general, no es imprescindible conocer el movimiento de cada partícula en detalle. Más bien, es más útil el concepto de la función de distribución  $f(\mathbf{r}, \xi, t)$ . Dicha función se define de tal manera que la cantidad  $f(\mathbf{r}, \xi, t)d^3rd^3\xi$  (siendo  $d^3r$  un volumen en el espacio físico y  $d^3\xi$  un volumen en espacio de velocidades<sup>1</sup>) representa la cantidad de partículas existentes en el volumen 6-dimensional del espacio de fases  $d^3rd^3\xi$  centrado en el punto  $(\mathbf{r}, \xi)$  en el instante  $t$ . En la Figura (2.1) se representa esquemáticamente un espacio 2-dimensional de fases y un volumen en dicho espacio. Por tanto, la cantidad  $f(\mathbf{r}, \xi, t)d^3rd^3\xi$  por definición indica la cantidad de partículas que tienen su estado (posición y velocidad) comprendido en el volumen dibujado en la Figura (2.1).

El objetivo principal es encontrar la dinámica que rige la función de distribución.

---

<sup>1</sup>Si  $\mathbf{r} = (x, y, z)$  entonces  $d^3r = dx dy dz$ , mientras que si  $\xi = (\xi_x, \xi_y, \xi_z)$  entonces  $d^3\xi = d\xi_x d\xi_y d\xi_z$

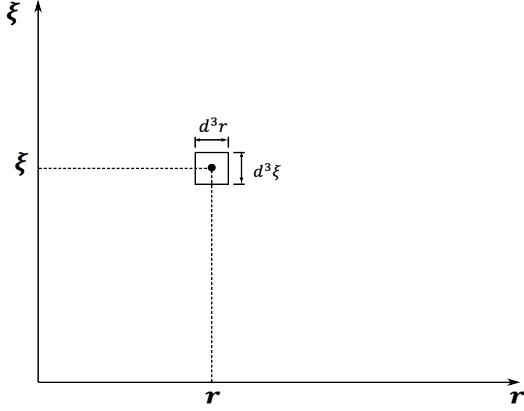


Figura 2.1: Representación del espacio de fases y un volumen de dicho espacio para un cierto instante del tiempo. La gráfica es ilustrativa y no representa la realidad ya que se tendría que dibujar en un espacio 6-dimensional.

Evidentemente la función de distribución cambia con el tiempo ya que las partículas entran y salen de los volúmenes en el espacio  $\mu$ . Si se supone la ausencia de colisiones, una moléculas en instante  $t$  con coordenadas en el espacio  $\mu(\boldsymbol{r}, \boldsymbol{\xi})_t$  se moverá a otro punto en el espacio  $\mu(\boldsymbol{r}', \boldsymbol{\xi}')_{t+\delta t}$  en el instante  $t + \delta t$ , siendo:

$$\boldsymbol{r}' = \boldsymbol{r} + \boldsymbol{\xi}\delta t \quad (2.1a)$$

$$\boldsymbol{\xi}' = \boldsymbol{\xi} + \mathbf{F} \frac{\delta t}{m} \quad (2.1b)$$

siendo  $F$  las fuerzas externas que actúan sobre la partícula y  $m$  su masa. El intervalo  $\delta t$  se define de tal manera que es mayor que el Tiempo de Colisión Medio (la mayoría de las colisiones duran  $\delta t$ ). Sin embargo,  $\delta t$  debe ser más pequeño que el Tiempo de Recorrido Medio (el tiempo medio que tarda una partícula que acaba de tener una colisión en colisionar con otra partícula).

Con la ausencia de colisiones se puede afirmar que la cantidad de partículas en el volumen  $d^3r d^3\xi$  en el instante  $t$  será igual a la cantidad de partículas en el volumen  $d^3r' d^3\xi'$  para el instante  $t + \delta t$ . Escrito dicha igualdad de forma matemática resulta:

$$f(\boldsymbol{r}, \boldsymbol{\xi}, t) d^3r d^3\xi = f\left(\boldsymbol{r} + \boldsymbol{\xi}\delta t, \boldsymbol{\xi} + \mathbf{F} \frac{\delta t}{m}, t\right) d^3r' d^3\xi' \quad (2.2)$$

En el caso de la presencia de colisiones, la cantidad:

$$f(\boldsymbol{r}, \boldsymbol{\xi}, t) d^3r d^3\xi - f\left(\boldsymbol{r} + \boldsymbol{\xi}\delta t, \boldsymbol{\xi} + \mathbf{F} \frac{\delta t}{m}, t\right) d^3r' d^3\xi' = C \quad (2.3)$$

representa la cantidad de partículas que entran en el volumen  $d^3rd^3\xi$  menos la cantidad de partículas que salen del volumen  $d^3rd^3\xi$  causadas por las colisiones durante el intervalo de tiempo  $\delta t$ . A continuación, se define el término de colisión  $\left[\frac{\partial f}{\partial t}\right]_c$  de la siguiente manera:

$$\left[\frac{\partial f}{\partial t}\right]_c d^3rd^3\xi = C \quad (2.4)$$

Se puede relacionar el volumen  $d^3rd^3\xi$  con el volumen  $d^3r'd^3\xi'$  calculando el determinante de la matriz Jacobiana  $J$  de la transformación:

$$d^3r'd^3\xi' = |J|d^3rd^3\xi \quad (2.5)$$

Para calcular el determinante se supone que la fuerza exterior  $F$  puede ser función del vector de posición y del vector de velocidad microscópica. De esta manera la matriz Jacobiana resulta ser:

$$|J| = \frac{\partial(\mathbf{r}', \boldsymbol{\xi}')}{(\mathbf{r}, \boldsymbol{\xi})} = \begin{pmatrix} 1 & 0 & 0 & \delta t & 0 & 0 \\ 0 & 1 & 0 & 0 & \delta t & 0 \\ 0 & 0 & 1 & 0 & 0 & \delta t \\ \frac{\partial F_x}{\partial r_x} \frac{\delta t}{m} & \frac{\partial F_x}{\partial r_y} \frac{\delta t}{m} & \frac{\partial F_x}{\partial r_z} \frac{\delta t}{m} & 1 + \frac{\partial F_x}{\partial \xi_x} \frac{\delta t}{m} & \frac{\partial F_x}{\partial \xi_y} \frac{\delta t}{m} & \frac{\partial F_x}{\partial \xi_z} \frac{\delta t}{m} \\ \frac{\partial F_y}{\partial r_x} \frac{\delta t}{m} & \frac{\partial F_y}{\partial r_y} \frac{\delta t}{m} & \frac{\partial F_y}{\partial r_z} \frac{\delta t}{m} & \frac{\partial F_y}{\partial \xi_x} \frac{\delta t}{m} & 1 + \frac{\partial F_y}{\partial \xi_y} \frac{\delta t}{m} & \frac{\partial F_y}{\partial \xi_z} \frac{\delta t}{m} \\ \frac{\partial F_z}{\partial r_x} \frac{\delta t}{m} & \frac{\partial F_z}{\partial r_y} \frac{\delta t}{m} & \frac{\partial F_z}{\partial r_z} \frac{\delta t}{m} & \frac{\partial F_z}{\partial \xi_x} \frac{\delta t}{m} & \frac{\partial F_z}{\partial \xi_y} \frac{\delta t}{m} & 1 + \frac{\partial F_z}{\partial \xi_z} \frac{\delta t}{m} \end{pmatrix} \quad (2.6)$$

Calculando el determinante de la matriz  $J$  y quedándose con los términos de primer orden en  $\delta t$  se obtiene el siguiente resultado:

$$|J| = 1 + \frac{\partial}{\partial \boldsymbol{\xi}} \cdot \mathbf{F} \frac{\delta t}{m} + O(\delta t^2) \quad (2.7)$$

En la Ecuación (2.7), la expresión  $\frac{\partial}{\partial \boldsymbol{\xi}}$  representa el gradiente respecto la velocidad. También se introduce la notación  $\frac{\partial}{\partial \mathbf{r}}$ , que representa el gradiente respecto la posición<sup>2</sup>.

A continuación se expande el término  $f(\mathbf{r}, \boldsymbol{\xi}, t)d^3rd^3\xi = f(\mathbf{r} + \boldsymbol{\xi}\delta t, \boldsymbol{\xi} + \mathbf{F}\frac{\delta t}{m}, t)$  en series de Taylor hasta el primer orden en  $\delta t$ :

$$f\left(\mathbf{r} + \boldsymbol{\xi}\delta t, \boldsymbol{\xi} + \mathbf{F}\frac{\delta t}{m}, t\right) = f(\mathbf{r}, \boldsymbol{\xi}, t) + \boldsymbol{\xi}\delta t \cdot \frac{\partial f}{\partial \mathbf{r}}\Big|_{(\mathbf{r}, \boldsymbol{\xi}, t)} + \mathbf{F}\frac{\delta t}{m} \cdot \frac{\partial f}{\partial \boldsymbol{\xi}}\Big|_{(\mathbf{r}, \boldsymbol{\xi}, t)} + \delta t \frac{\partial f}{\partial t}\Big|_{(\mathbf{r}, \boldsymbol{\xi}, t)} + O(\delta t^2) \quad (2.8)$$

---

<sup>2</sup>  $\frac{\partial}{\partial \mathbf{r}} \equiv \nabla_{\mathbf{r}} = \left( \frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right)$  y  $\frac{\partial}{\partial \boldsymbol{\xi}} \equiv \nabla_{\boldsymbol{\xi}} = \left( \frac{\partial}{\partial \xi_x}, \frac{\partial}{\partial \xi_y}, \frac{\partial}{\partial \xi_z} \right)$

Introduciendo las Ecuaciones (2.4,2.5,2.7) dentro de la Ecuación (2.3), dividiendo por  $\delta t$  y evaluando el límite cuando  $\delta t \rightarrow 0$ , se obtiene la conocida ecuación de Boltzmann:

$$\frac{\partial f(\mathbf{r}, \boldsymbol{\xi}, t)}{\partial t} + \boldsymbol{\xi} \cdot \nabla_{\mathbf{r}} f(\mathbf{r}, \boldsymbol{\xi}, t) + \frac{1}{m} \nabla_{\boldsymbol{\xi}} \cdot (\mathbf{F} f(\mathbf{r}, \boldsymbol{\xi}, t)) = \left[ \frac{\partial f}{\partial t} \right]_c \quad (2.9)$$

En el caso en el que la fuerza externa  $\mathbf{F}$  no sea función de la velocidad de las partículas, como por ejemplo en el caso de la fuerza de gravitación, la Ecuación (2.9) se simplifica obteniendo la siguiente ecuación:

$$\left[ \frac{\partial}{\partial t} + \boldsymbol{\xi} \cdot \nabla_{\mathbf{r}} + \frac{\mathbf{F}}{m} \cdot \nabla_{\boldsymbol{\xi}} \right] f(\mathbf{r}, \boldsymbol{\xi}, t) = \left[ \frac{\partial f}{\partial t} \right]_c \quad (2.10)$$

Se trata de una ecuación integrodiferencial donde el término de colisión, como se verá más adelante, es una ecuación integral. Otra característica importante es la linealidad del término de convección que, a diferencia de las Ecuaciones de Navier Stokes, es lineal.

En un intervalo de tiempo  $\delta t$  pueden ocurrir colisiones entre las partículas originalmente en  $d^3 r d^3 \xi$ , que arrojan moléculas fuera del volumen. Pero también pueden haber colisiones en los volúmenes vecinos que pueden mandar partículas al interior del  $d^3 r d^3 \xi$ . De esta manera se separa el término de colisiones en dos contribuciones:

$$Q \equiv \left[ \frac{\partial f}{\partial t} \right]_c = \left[ \frac{\partial f}{\partial t} \right]_c^+ - \left[ \frac{\partial f}{\partial t} \right]_c^- \quad (2.11)$$

siendo  $\left[ \frac{\partial f}{\partial t} \right]_c^+$  la parte correspondiente a la ganancia de partículas y  $\left[ \frac{\partial f}{\partial t} \right]_c^-$  la parte correspondiente a la pérdida de partículas.

El hecho de que tenga lugar una colisión entre dos partículas significa que ambas deben ocupar una cierta configuración en el espacio simultáneamente en el instante de tiempo  $t$ . Por ejemplo, deben ocupar posiciones  $\mathbf{r}_1$  y  $\mathbf{r}_2$ , y deben tener velocidades  $\boldsymbol{\xi}_1$  y  $\boldsymbol{\xi}_2$ . Para que ocurra una colisión la distancia entre ambas debe ser menor o igual al *alcance del potencial de interacción* y las velocidades también deben tener direcciones apropiadas. El número total de partículas con las características descritas anteriormente proviene de la función de distribución binaria,  $f_2(\mathbf{r}_1, \mathbf{r}_2, \boldsymbol{\xi}_1, \boldsymbol{\xi}_2, t)$ . Para poder avanzar, se introduce lo que se llama la hipótesis del *caos molecular* o *Stosszahlansatz* (hipótesis sobre el número de colisiones) [1, 2]. La hipótesis supone que la presencia de ambas partículas, en las condiciones apropiadas para la colisión, solamente depende de la posición y que es el producto de dos eventos totalmente

independientes. Al suponer esto, se suprimen todas las correlaciones entre las moléculas. En otras palabras, las moléculas que intervienen en una colisión no han colisionado nunca ni lo harán en el futuro. Escrita la hipótesis del *caos molecular* matemáticamente resulta:

$$f_2(\mathbf{r}_1, \mathbf{r}_2, \boldsymbol{\xi}_1, \boldsymbol{\xi}_2, t) = f(\mathbf{r}, \boldsymbol{\xi}_1, t)f(\mathbf{r}, \boldsymbol{\xi}_2, t) \quad (2.12)$$

En [3, 4, 5] se puede encontrar la deducción del término de colisiones:

$$Q(f, f) = \int [f(\mathbf{r}, \boldsymbol{\xi}'_1, t)f(\mathbf{r}, \boldsymbol{\xi}'_2, t) - f(\mathbf{r}, \boldsymbol{\xi}_1, t)f(\mathbf{r}, \boldsymbol{\xi}_2, t)] \sigma(\Omega) \xi_r d\Omega d^3 \boldsymbol{\xi}_2 \quad (2.13)$$

siendo  $\boldsymbol{\xi}_1, \boldsymbol{\xi}_2$  las velocidades de las partículas antes de la colisión,  $\boldsymbol{\xi}'_1, \boldsymbol{\xi}'_2$  las velocidades después de la colisión,  $\sigma(\Omega)$  la sección eficaz de colisión,  $\Omega$  el ángulo sólido y  $\xi_r$  el módulo de la velocidad relativa antes de la colisión. El hecho de escribir  $Q(f, f)$  en vez de  $Q(f)$  sirve para remarcar que el término de colisión no es lineal (segundo grado de no linealidad).

Se supone que cuando se deja evolucionar infinitamente el sistema, éste alcanza el estado de equilibrio caracterizado por la función de distribución de equilibrio  $g$  definida como un estado en el que el término de colisiones se anula. Esto significa que la pérdida de partículas es igual a la ganancia de partículas durante las colisiones, y no significa precisamente que no haya colisiones. De esta manera, si  $Q(g, g) = 0$  implica que:

$$g(\mathbf{r}, \boldsymbol{\xi}'_1)g(\mathbf{r}, \boldsymbol{\xi}'_2) = g(\mathbf{r}, \boldsymbol{\xi}_1)g(\mathbf{r}, \boldsymbol{\xi}_2) \quad (2.14)$$

A continuación se evalúa el logaritmo en ambos lados de la Ecuación (2.14) y se obtiene:

$$\ln(g(\mathbf{r}, \boldsymbol{\xi}'_1)) + \ln(g(\mathbf{r}, \boldsymbol{\xi}'_2)) = \ln(g(\mathbf{r}, \boldsymbol{\xi}_1)) + \ln(g(\mathbf{r}, \boldsymbol{\xi}_2)) \quad (2.15)$$

En [5], la función  $\ln(g)$  que cumple la Ecuación (2.15) recibe el nombre de invariante respecto la suma y debe ser de la siguiente forma:

$$\ln(g) = A + \mathbf{B} \cdot \boldsymbol{\xi} + C |\boldsymbol{\xi}|^2 \quad (2.16)$$

siendo  $A$  y  $C$  constantes y  $\mathbf{B}$  un vector de la misma dimensión que el vector de velocidad. Para demostrar la condición de suficiencia de la Ecuación (2.16) para cumplir la Ecuación (2.15), únicamente hace falta tener en cuenta que el proceso de colisión es elástico y por tanto, tanto el momentum como la energía cinética se conservan ( $\boldsymbol{\xi}_1 + \boldsymbol{\xi}_2 = \boldsymbol{\xi}'_1 + \boldsymbol{\xi}'_2$  y  $|\boldsymbol{\xi}_1|^2 + |\boldsymbol{\xi}_2|^2 = |\boldsymbol{\xi}'_1|^2 + |\boldsymbol{\xi}'_2|^2$ , donde se ha supuesto

que todas las partículas tienen la misma masa). Demostrar la condición necesaria ya no es tan trivial y se puede encontrar en [5].

A continuación se definen  $\alpha$ ,  $\beta$  y  $\gamma$  de la siguiente manera:  $A = \ln(\alpha) - \beta |\gamma|^2$ ,  $\mathbf{B} = 2\beta\gamma$  y  $C = -\beta$ . Introduciendo las definiciones anteriores en la Ecuación (2.16) se obtiene la función de distribución de equilibrio local:

$$g = \alpha \exp(-\beta |\boldsymbol{\xi} - \boldsymbol{\gamma}|^2) \quad (2.17)$$

Para determinar las constantes  $\alpha$ ,  $\beta$  y  $\gamma$  se aplican las leyes de conservación, las cuales serán introducidas más adelante.

Según el trabajo de Bhatnager, Gross & Krook (BGK) 1954 [7], el término de colisión se interpreta como un proceso de relajación hasta el estado de equilibrio local  $g$ . Dicho de otra forma, la variación de la función de distribución  $f$  debida a la colisión es proporcional a la diferencia del estado de equilibrio local y el estado actual del sistema. El parámetro de proporcionalidad se introduce mediante el tiempo de relajación  $\lambda$ , que es una función compleja de la función de distribución (también se habla del tiempo de relajación adimensional y de la frecuencia de relajación adimensional  $\tau = \frac{\lambda}{\delta t}$  y  $\omega = \frac{\delta t}{\lambda}$ , respectivamente). Por tanto, el término de colisión se escribe como:

$$Q(f) \equiv \left[ \frac{\partial f}{\partial t} \right]_c = -\frac{1}{\lambda}(f - g) \quad (2.18)$$

A continuación se va a deducir una propiedad muy importante que debe cumplir el término de colisión. Para ello se multiplica el término de colisión  $Q(f, f)$  por una función escalar arbitraria  $\psi(\mathbf{r}, \boldsymbol{\xi}, t)$  y se integra en todo el rango de velocidades microscópicas, y tal y como se demuestra en [5], da el siguiente resultado:

$$\iiint_{-\infty}^{\infty} \psi(\mathbf{r}, \boldsymbol{\xi}, t) Q(f, f) d^3\xi = \iiint_{-\infty}^{\infty} \psi \left[ \frac{\partial f}{\partial t} \right]_c d^3\xi = \frac{1}{4} \iiint_{-\infty}^{\infty} (\psi_2 + \psi_1 - \psi'_2 - \psi'_1) \left[ \frac{\partial f}{\partial t} \right]_c d^3\xi \quad (2.19)$$

De la anterior expresión se deduce que, en el caso de que la función arbitraria  $\psi$  sea una combinación lineal de las invariantes de colisión, la Ecuación (2.19) se anula:

$$\psi(\mathbf{r}, \boldsymbol{\xi}, t) = A + \mathbf{B} \cdot \boldsymbol{\xi} + C |\boldsymbol{\xi}|^2 \implies \iiint_{-\infty}^{\infty} \psi(\mathbf{r}, \boldsymbol{\xi}, t) Q(f, f) d^3\xi = 0 \quad (2.20)$$

En el caso de suponer el tiempo de relajación  $\lambda$  constante, el término de colisión BGK  $Q(f)$  de la Ecuación (2.18) también debe cumplir la Ecuación (2.20), con lo

cual:

$$\left. \begin{array}{l} \psi(\mathbf{r}, \boldsymbol{\xi}, t) = A + \mathbf{B} \cdot \boldsymbol{\xi} + C |\boldsymbol{\xi}|^2 \\ \text{BGK con } \lambda = \text{constante} \end{array} \right\} \implies \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} g \psi d^3 \xi = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f \psi d^3 \xi \quad (2.21)$$

La Ecuación (2.21) indica que los momentos<sup>3</sup> de la Ecuación de Boltzmann se pueden evaluar tanto respecto la función de distribución  $f$ , como respecto la función de distribución de equilibrio  $g$ .

La función de distribución es una variable primaria y a partir de ella se deben calcular las propiedades macroscópicas, que son de más utilidad. Una propiedad macroscópica, en un punto del dominio físico en un cierto instante de tiempo, se define como el promedio en todo el rango de velocidades microscópicas de la función de distribución. Cabe mencionar que, una vez evaluado el momento de la función de distribución, el resultado obtenido deja de ser función de la velocidad microscópica. A continuación se definen las siguientes cantidades macroscópicas:

**Densidad de masa**  $\left[ \frac{\text{kg}}{\text{m}^3} \right]$

$$\rho(\mathbf{r}, t) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} m f(\mathbf{r}, \boldsymbol{\xi}, t) d^3 \xi \quad (2.22)$$

**Densidad de momentum**  $\left[ \frac{\text{kg}}{\text{m}^3} \left( \frac{\text{m}}{\text{s}} \right) \right]$

$$\rho(\mathbf{r}, t) \mathbf{u}(\mathbf{r}, t) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} m \boldsymbol{\xi} f(\mathbf{r}, \boldsymbol{\xi}, t) d^3 \xi \quad (2.23)$$

**Densidad de energía total**  $\left[ \frac{\text{kg}}{\text{m}^3} \left( \frac{\text{m}}{\text{s}} \right)^2 \right]$

$$\rho(\mathbf{r}, t) e(\mathbf{r}, t) = \frac{1}{2} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} m |\boldsymbol{\xi}|^2 f(\mathbf{r}, \boldsymbol{\xi}, t) d^3 \xi \quad (2.24)$$

La velocidad peculiar  $\mathbf{C} = \boldsymbol{\xi} - \mathbf{u}$ , es la velocidad de la partícula medida por un observador que se encuentra en una referencia que se mueve a la velocidad macroscópica. Teniendo la definición anterior en cuenta, se demuestra que la densidad de energía total se puede separar en la densidad de energía cinética macroscópica y la

---

<sup>3</sup>La expresión general del momento de grado  $n$  en el caso de tener una única variable  $x$  es  $M_n = \int A x^n f(x) dx$  siendo  $A$  una constante.

densidad de energía interna  $\rho\varepsilon$  definida con la velocidad peculiar:

$$\rho e = \frac{1}{2} \iiint_{-\infty}^{\infty} m |\mathbf{C}|^2 f(\mathbf{r}, \boldsymbol{\xi}, t) d^3 \xi + \frac{1}{2} \rho |\mathbf{u}|^2 \quad (2.25)$$

En la deducción anterior se ha utilizado el hecho de que:

$$\iiint_{-\infty}^{\infty} m |\mathbf{C}| f(\mathbf{r}, \boldsymbol{\xi}, t) d^3 \xi = \rho |\mathbf{u}| - |\mathbf{u}| \rho = 0 \quad (2.26)$$

### Tensor de presión

$$p_{ij}(\mathbf{r}, t) = \iint_{-\infty}^{\infty} m C_i C_j f(\mathbf{r}, \boldsymbol{\xi}, t) d^3 \xi \quad (2.27)$$

El tensor de tensiones  $\sigma_{ij}$  se define con el signo negativo del tensor de presión ( $\sigma_{ij} = -p_{ij}$ ). La presión hidrostática se define como un tercio de la traza del tensor de presión, es decir:

$$p(\mathbf{r}, t) = \frac{1}{3} \text{tr}(\mathbf{p}) = \frac{1}{3} \iiint_{-\infty}^{\infty} m |\mathbf{C}|^2 f(\mathbf{r}, \boldsymbol{\xi}, t) d^3 \xi \quad (2.28)$$

El tensor de tensiones  $\sigma_{ij}$  se puede dividir en una parte isotrópica (esfuerzos normales)  $-p\delta_{ij}$ , siendo  $p$  la presión hidrostática, y en una parte anisotrópica (esfuerzos tangenciales) con traza nula  $\tau_{ij}$ .

$$\sigma_{ij} = -p\delta_{ij} + \tau_{ij} \quad (2.29)$$

Comparando la Ecuación (2.28) con la Ecuación (2.25) se observa que hay una relación entre la presión hidrostática y la densidad de energía interna:

$$p(\mathbf{r}, t) = \frac{2}{3} \rho(\mathbf{r}, t) \varepsilon(\mathbf{r}, t) \quad (2.30)$$

A continuación se utiliza la ecuación de estado de gases ideales  $p = nk_B T$  (siendo  $n$  la cantidad de partículas por unidad de volumen y  $k_B$  la constante de Boltzmann) para obtener una expresión para la temperatura:

$$T(\mathbf{r}, t) = \frac{2}{3} \frac{m}{k_B} \varepsilon(\mathbf{r}, t) \quad (2.31)$$

### Tensor de segundo orden (second order momentum flux tensor)

$$\Pi_{ij} = \iint_{-\infty}^{\infty} m \xi_i \xi_j f(\mathbf{r}, \boldsymbol{\xi}, t) d^3 \xi \quad (2.32)$$

Utilizando la definición del tensor de tensiones se demuestra que:

$$\Pi_{ij} = -\sigma_{ij} + u_i u_j \rho \quad (2.33)$$

### Flujo de calor

$$q_i(\mathbf{r}, t) = \frac{1}{2} \iint_{-\infty}^{\infty} m |\mathbf{C}|^2 C_i f(\mathbf{r}, \boldsymbol{\xi}, t) d^3 \xi \quad (2.34)$$

A continuación se va a deducir la Ecuación de Transferencia, que permitirá recuperar las leyes de conservación de las propiedades macroscópicas. La multiplicación de la Ecuación de Boltzmann (2.10) por una función escalar arbitraria  $\psi(\mathbf{r}, \boldsymbol{\xi}, t)$  y la integración de la ecuación en todo el rango de las velocidades microscópicas, da el siguiente resultado:

$$\iint_{-\infty}^{\infty} \psi \frac{\partial f}{\partial t} d^3 \xi + \iint_{-\infty}^{\infty} (\boldsymbol{\xi} \cdot \nabla_{\mathbf{r}} f) \psi d^3 \xi + \iint_{-\infty}^{\infty} \left( \frac{\mathbf{F}}{m} \cdot \nabla_{\boldsymbol{\xi}} f \right) \psi d^3 \xi = \iint_{-\infty}^{\infty} \psi \left[ \frac{\partial f}{\partial t} \right]_c d^3 \xi \quad (2.35)$$

A continuación se desarrollan los diferentes términos de la Ecuación (2.35) utilizando la regla de la cadena:

$$\frac{\partial}{\partial t} \iint_{-\infty}^{\infty} \psi f d^3 \xi = \iint_{-\infty}^{\infty} \frac{\partial \psi}{\partial t} f d^3 \xi + \iint_{-\infty}^{\infty} \frac{\partial f}{\partial t} \psi d^3 \xi \quad (2.36a)$$

$$\nabla_{\mathbf{r}} \cdot \iint_{-\infty}^{\infty} \boldsymbol{\xi} \psi f d^3 \xi = \iint_{-\infty}^{\infty} (\boldsymbol{\xi} \cdot \nabla_{\mathbf{r}} \psi) f d^3 \xi + \iint_{-\infty}^{\infty} (\boldsymbol{\xi} \cdot \nabla_{\mathbf{r}} f) \psi d^3 \xi \quad (2.36b)$$

$$\nabla_{\boldsymbol{\xi}} \cdot \iint_{-\infty}^{\infty} \frac{\mathbf{F}}{m} \psi f d^3 \xi = \iint_{-\infty}^{\infty} \left( \frac{\mathbf{F}}{m} \cdot \nabla_{\boldsymbol{\xi}} \psi \right) f d^3 \xi + \iint_{-\infty}^{\infty} \left( \frac{\mathbf{F}}{m} \cdot \nabla_{\boldsymbol{\xi}} f \right) \psi d^3 \xi \quad (2.36c)$$

Introduciendo las Ecuaciones (2.36) en la Ecuación (2.35) queda de la siguiente manera:

$$\begin{aligned} & \frac{\partial}{\partial t} \iint_{-\infty}^{\infty} \psi f d^3 \xi + \nabla_{\mathbf{r}} \cdot \iint_{-\infty}^{\infty} \boldsymbol{\xi} \psi f d^3 \xi + \nabla_{\boldsymbol{\xi}} \cdot \iint_{-\infty}^{\infty} \frac{\mathbf{F}}{m} \psi f d^3 \xi - \\ & - \iint_{-\infty}^{\infty} \left[ \frac{\partial \psi}{\partial t} + (\boldsymbol{\xi} \cdot \nabla_{\mathbf{r}} \psi) + \left( \frac{\mathbf{F}}{m} \cdot \nabla_{\boldsymbol{\xi}} \psi \right) \right] f d^3 \xi = \iint_{-\infty}^{\infty} \psi \left[ \frac{\partial f}{\partial t} \right]_c d^3 \xi \end{aligned} \quad (2.37)$$

Se aplica el teorema de divergencia<sup>4</sup> al tercer término de la izquierda de la Ecuación (2.37):

$$\nabla_{\xi} \cdot \iiint_{-\infty}^{\infty} \frac{\mathbf{F}}{m} \psi f d^3\xi = \oint_S \psi \frac{\mathbf{F}}{m} \cdot \mathbf{n} f dS \approx 0 \quad (2.38)$$

En la Ecuación (2.38) la integral se evalúa sobre el contorno del espacio de velocidades, siendo  $\mathbf{n}$  y  $dS$  el vector normal unitario y el elemento de superficie de dicho contorno, respectivamente. La integral tiene un valor aproximadamente nulo ya que la función de distribución decrece rápidamente para los valores grandes de la velocidad microscópica, que es precisamente lo que pasa en el contorno del espacio de velocidades.

En [5] se demuestra que el término de colisiones de la Ecuación (2.37) se puede escribir de la siguiente manera:

$$\iiint_{-\infty}^{\infty} \psi \left[ \frac{\partial f}{\partial t} \right]_c d^3\xi = \frac{1}{4} \iiint_{-\infty}^{\infty} (\psi_2 + \psi_1 - \psi'_2 - \psi'_1) \left[ \frac{\partial f}{\partial t} \right]_c d^3\xi \quad (2.39)$$

siendo  $\psi_1$  y  $\psi_2$  el valor de la función  $\psi$  para las partículas antes de la colisión, y  $\psi'_1$  y  $\psi'_2$  el valor de la función  $\psi$  después de la colisión binaria.

A continuación se definen las siguientes magnitudes:

$$\begin{aligned} \Psi &= \iiint_{-\infty}^{\infty} \psi f d^3\xi & \Phi &= \iiint_{-\infty}^{\infty} \psi C d^3\xi & S &= \iiint_{-\infty}^{\infty} \left( \frac{\mathbf{F}}{m} \cdot \nabla_{\xi} \psi \right) f d^3\xi \\ P &= P_1 + P_2 & P_1 &= \iiint_{-\infty}^{\infty} \left[ \frac{\partial \psi}{\partial t} + (\boldsymbol{\xi} \cdot \nabla_{\mathbf{r}} \psi) \right] f d^3\xi & \\ & & P_2 &= \frac{1}{4} \iiint_{-\infty}^{\infty} (\psi_2 + \psi_1 - \psi'_2 - \psi'_1) \left[ \frac{\partial f}{\partial t} \right]_c d^3\xi & \end{aligned} \quad (2.40)$$

Introduciendo las definiciones en la Ecuación (2.37) se obtiene la Ecuación de Transferencia:

$$\frac{\partial \Psi}{\partial t} + \nabla_{\mathbf{r}} \cdot (\Psi \mathbf{u} + \Phi) = S + P \quad (2.41)$$

Escogiendo de forma apropiada la función  $\psi$  se pueden obtener las leyes de conservación macroscópicas:

---

<sup>4</sup>El teorema de divergencia relaciona la integral de la divergencia de un campo vectorial sobre un volumen  $\Omega$  con la integral sobre su frontera  $\partial\Omega$ .

$$\int_{\Omega} \nabla \cdot \mathbf{a} = \int_{\partial\Omega} (\mathbf{a} \cdot \mathbf{n}) dS$$

siendo  $\mathbf{n}$  el vector normal a la frontera y  $dS$  un elemento diferencial de superficie en la frontera.

- Conservación de la masa:  $\psi = m$

Al particularizar  $\psi = m$ , la cantidad  $\Psi$  representa la densidad. El término  $\Phi$  se anula ya que  $\Phi = \rho\mathbf{u} - \rho\mathbf{u}$ . El término fuente  $S$  también se anula ya que el gradiente de una constante es 0. Por otro lado, los términos de producción también son 0, ya que en  $P_1$  tanto la derivada respecto el tiempo como el gradiente de una constante son 0 y, por último,  $P_2$  es 0 ya que la masa es un invariante de colisión. Teniendo todo lo dicho anteriormente en cuenta, se obtiene la ecuación de continuidad macroscópica:

$$\frac{\partial \rho}{\partial t} + \nabla_{\mathbf{r}} \cdot (\rho\mathbf{u}) = 0 \quad (2.42)$$

- Conservación del momento lineal (componente  $i$ ):  $\psi = m\xi_i$

Al particularizar  $\psi = m\xi_i$  en la Ecuación (2.41), la cantidad  $\Psi$  representa el momento lineal  $\rho\mathbf{u}_i$ ,  $\Phi$  representa una parte del tensor de presiones  $\mathbf{p}_i$  definida como  $\begin{pmatrix} p_{i1} & p_{i2} & p_{i3} \end{pmatrix}^T$ ,  $S$  representa el término de fuerzas externas  $\rho F_i$ ,  $P_2$  vale 0 ya que el momento lineal es un invariante de colisión y, por último,  $P_1$  también vale 0.

$$\frac{(\partial \rho u_i)}{\partial t} + \nabla_{\mathbf{r}} \cdot (\rho u_i \mathbf{u} + \mathbf{p}_i) = \rho F_i \quad (2.43)$$

Al tener en cuenta todos los componentes de la velocidad microscópica se obtiene una ecuación vectorial.

$$\frac{(\partial \rho \mathbf{u})}{\partial t} + \nabla_{\mathbf{r}} \cdot (\rho \mathbf{u} \mathbf{u}) = \nabla_{\mathbf{r}} \cdot \boldsymbol{\sigma} + \rho \mathbf{F} \quad (2.44)$$

- Conservación de la energía:  $\psi = m|\xi|^2/2$

De manera análoga a los dos casos anteriores, se obtiene la siguiente ecuación escalar:

$$\frac{\partial}{\partial t} \left[ \rho \left( \varepsilon + \frac{1}{2} |\mathbf{u}|^2 \right) \right] + \nabla_{\mathbf{r}} \cdot \left( \rho \left( \varepsilon + \frac{1}{2} |\mathbf{u}|^2 \right) \mathbf{u} + \mathbf{q} + \rho \mathbf{u} \right) = \rho \mathbf{F} \cdot \mathbf{u} \quad (2.45)$$

Ahora que se conoce como calcular las propiedades macroscópicas se procede a determinar las constantes  $\alpha$ ,  $\beta$  y  $\gamma$  de la función de distribución de equilibrio (2.17). Para hacerlo se utiliza la propiedad (2.21), de manera que se evalúan la densidad de masa, la densidad de momentum y la densidad de energía interna utilizando la

función de distribución de equilibrio.<sup>5</sup>

$$\rho = \int_{-\infty}^{\infty} m\alpha \exp(-\beta |\boldsymbol{\xi} - \boldsymbol{\gamma}|^2) d^D \boldsymbol{\xi} = m\alpha \left(\frac{\pi}{\beta}\right)^{\frac{D}{2}} \quad (2.46a)$$

$$\rho \mathbf{u} = \int_{-\infty}^{\infty} m \boldsymbol{\xi} \alpha \exp(-\beta |\boldsymbol{\xi} - \boldsymbol{\gamma}|^2) d^D \boldsymbol{\xi} = \gamma m\alpha \left(\frac{\pi}{\beta}\right)^{\frac{D}{2}} \quad (2.46b)$$

$$\rho \varepsilon = \frac{1}{2} \int_{-\infty}^{\infty} m |\boldsymbol{\xi}|^2 \alpha \exp(-\beta |\boldsymbol{\xi} - \boldsymbol{\gamma}|^2) d^D \boldsymbol{\xi} = \frac{3}{4} m\alpha \pi^{\frac{D}{2}} \beta^{-\frac{D}{2}} \beta^{-1} \quad (2.46c)$$

siendo  $D$  la dimensión del problema (2 en caso bidimensional y 3 en caso tridimensional). Comparando las dos primeras ecuaciones resulta ser que  $\rho = m\alpha \left(\frac{\pi}{\beta}\right)^{\frac{D}{2}}$  y  $\boldsymbol{\gamma} = \mathbf{u}$ . Por último, combinando la tercera ecuación con la Ecuación (2.31) se obtiene  $\beta = \frac{1}{2RT}$ , donde se ha introducido la definición de la constante del gas  $R = \frac{k_B}{m}$ . Finalmente la Ecuación de Distribución de Equilibrio local queda de la siguiente manera:

$$g = \frac{\rho}{m (2\pi RT)^{D/2}} e^{-\frac{|\boldsymbol{\xi}-\mathbf{u}|^2}{2RT}} \quad (2.47)$$

## 2.2. Modelos de LBM

El Método de Lattice Boltzmann (LBM) se fundamenta en la discretización del espacio de velocidades  $\boldsymbol{\xi}$  de la Ecuación continua de Boltzmann (2.10). Para poder implementarlo es necesario también una discretización en el espacio y en el tiempo. En esta sección se van a discutir los dos principales métodos de discretización del espacio de velocidades (SRT y MRT). Se utilizará la notación  $DDQq$  para designar el modelo de discretización, siendo la segunda  $D$  el número de dimensiones y  $q$  la cantidad de velocidades discretas.

### 2.2.1. Single Relaxation Time (SRT)

El modelo con un único parámetro de relajación o SRT, se basa en la utilización de la Ecuación de Boltzmann (2.10) con el término de colisión BGK (2.18), y suponiendo

---

<sup>5</sup>Para resolver las integrales es de utilidad hacer el cambio de variable  $x = \boldsymbol{\xi} - \boldsymbol{\gamma}$  y utilizar los siguientes resultados para las integrales impropias:  $\int_{-\infty}^{\infty} e^{-\beta x} dx = \sqrt{\frac{\pi}{\beta}}$ ,  $\int_{-\infty}^{\infty} x e^{-\beta x} dx = 0$  y

$\int_{-\infty}^{\infty} x^2 e^{-\beta x} dx = \frac{1}{2} \sqrt{\frac{\pi}{\beta^3}}$ .

que el tiempo de relajación  $\lambda$  es constante. Las deducciones que se hacen en este apartado para construir los diferentes modelos se basan en el trabajo de Li-Shi Luo expuesto en [8], quien es el primero en obtener las ecuaciones del LBM a partir de la Ecuación continua de Boltzmann.

En este apartado se va a deducir el modelo D2Q9, el cual es el más utilizado para el caso bidimensional. Para el caso tridimensional se utiliza con mucha frecuencia el modelo D3Q19.

Se parte de la ecuación continua de Boltzmann (2.10), sin término de fuerzas externas y utilizando el término de colisión BGK (2.18), y se transforma en una ecuación diferencial ordinaria (EDO) utilizando la derivada a lo largo de la línea característica  $\xi$ :  $\frac{d}{dt} \equiv \frac{\partial}{\partial t} + \xi \cdot \nabla$ .

$$\frac{df}{dt} + \frac{1}{\lambda} f = \frac{1}{\lambda} g \quad (2.48)$$

Seguidamente, se integra la Ecuación (2.48) entre 0 y  $\delta t$ . Se queda con los términos de hasta el orden 1 en  $\delta t$  y de esta manera se obtiene la ecuación de evolución de la función de distribución en tiempo discreto.

$$f(\mathbf{r} + \xi \delta t, \xi, t + \delta t) - f(\mathbf{r}, \xi, t) = -\frac{1}{\tau} (f(\mathbf{r}, \xi, t) - g(\mathbf{r}, \xi, t)) + O(\delta t^2) \quad (2.49)$$

Para evaluar  $g$  es necesario calcular las propiedades macroscópicas evaluando los momentos de la función de distribución  $\rho$ ,  $\rho \mathbf{u}$  y  $\rho e$  de forma exacta utilizando métodos de cuadratura. Para ello se desarrolla en series de Taylor<sup>6</sup> la Función de Distribución de Equilibrio (2.47), y se queda con los términos de hasta orden 2 de  $|\mathbf{u}|$ .

$$f^{(eq)} = \frac{\rho}{m(2\pi RT)^{D/2}} e^{-\frac{|\xi|^2}{2RT}} \left\{ 1 + \frac{\xi \cdot \mathbf{u}}{RT} + \frac{(\xi \cdot \mathbf{u})^2}{2(RT)^2} - \frac{|\mathbf{u}|^2}{2RT} \right\} \quad (2.50)$$

La aproximación anterior limita el número de Mach y hace imposible realizar simulaciones con números de Mach altos.

De esta forma, el objetivo de la discretización del espacio de velocidades es poder

---

<sup>6</sup>Teniendo en cuenta que  $|\xi - \mathbf{u}|^2 = |\xi|^2 + |\mathbf{u}|^2 - 2\xi \cdot \mathbf{u}$

evaluar fácilmente y de manera exacta los momentos de la función de distribución:

$$\rho(\mathbf{r}, t) = \sum_{\alpha=0}^{q-1} W_\alpha f^{(eq)}(\mathbf{r}, \boldsymbol{\xi}_\alpha, t) = \sum_{\alpha=0}^{q-1} f_\alpha^{(eq)}(\mathbf{r}, t) \quad (2.51a)$$

$$\rho(\mathbf{r}, t) \mathbf{u}(\mathbf{r}, t) = \sum_{\alpha=0}^{q-1} \boldsymbol{\xi}_\alpha W_\alpha f^{(eq)}(\mathbf{r}, \boldsymbol{\xi}_\alpha, t) = \sum_{\alpha=0}^{q-1} \boldsymbol{\xi}_\alpha f_\alpha^{(eq)}(\mathbf{r}, t) \quad (2.51b)$$

$$\rho(\mathbf{r}, t) e(\mathbf{r}, t) = \sum_{\alpha=0}^{q-1} |\boldsymbol{\xi}_\alpha - \mathbf{u}|^2 W_\alpha f^{(eq)}(\mathbf{r}, \boldsymbol{\xi}_\alpha, t) = \sum_{\alpha=0}^{q-1} |\boldsymbol{\xi}_\alpha - \mathbf{u}|^2 f_\alpha^{(eq)}(\mathbf{r}, t) \quad (2.51c)$$

A continuación se van a deducir las ecuaciones que rigen el modelo de discretización D2Q9. Para ello se define la expresión general del momento respecto la función de distribución de equilibrio  $I = \iint_{-\infty}^{\infty} \psi(\boldsymbol{\xi}) f^{(eq)} d^2\xi$  y, teniendo en cuenta que la función  $\psi$  puede ser de forma  $\psi_{m,n}(\boldsymbol{\xi}) = \xi_x^m \xi_y^n$ , se obtiene el siguiente resultado para el momento:

$$I_{m,n} = \frac{\rho}{2\pi RT} \iint_{-\infty}^{\infty} \xi_x^m \xi_y^n e^{-\frac{|\boldsymbol{\xi}|^2}{2RT}} \left\{ 1 + \frac{\boldsymbol{\xi} \cdot \mathbf{u}}{RT} + \frac{(\boldsymbol{\xi} \cdot \mathbf{u})^2}{2(RT)^2} - \frac{|\mathbf{u}|^2}{2RT} \right\} d^2\xi \quad (2.52)$$

Introduciendo el cambio de variable  $\zeta = \frac{\boldsymbol{\xi}}{\sqrt{2RT}}$ , y definiendo  $I_m = \int_{-\infty}^{\infty} \zeta^m e^{\zeta^2} d\zeta$ , la Ecuación (2.52) se escribe como:

$$I_{m,n} = \frac{\rho}{2\pi RT} \left( \sqrt{2RT} \right)^{m+n} \left\{ \left( 1 - \frac{|\mathbf{u}|^2}{2RT} \right) I_m I_n + \frac{2}{\sqrt{2RT}} (u I_{m+1} I_n + v I_m I_{n+1}) + \frac{1}{RT} (u^2 I_{m+2} I_n + v^2 I_m I_{n+2} + 2uv I_{m+1} I_{n+1}) \right\} \quad (2.53)$$

La cuadratura de Hermite de orden 3 es la más adecuada para evaluar de forma exacta la integral  $I_m$ :

$$I_m = \int_{-\infty}^{\infty} \zeta^m e^{\zeta^2} d\zeta = \sum_{k=1}^3 \omega_k \zeta_k^m \quad (2.54)$$

siendo  $\omega_k$  y  $\zeta_k$  el peso y la abscisa de la cuadratura recogidas en la siguiente tabla:

$k$	1	2	3
$\omega_k$	$\frac{\sqrt{\pi}}{6}$	$\frac{2\sqrt{\pi}}{3}$	$\frac{\sqrt{\pi}}{6}$
$\zeta_k$	$-\sqrt{\frac{3}{2}}$	0	$\sqrt{\frac{3}{2}}$

Tabla 2.1: Cuadratura de Hermite de orden 3.

Introduciendo la Ecuación (2.54) en la Ecuación (2.53) y agrupando términos, se obtiene la siguiente ecuación:

$$I_{m,n} = \sum_{k=1}^3 \sum_{l=1}^3 \frac{\rho}{\pi} \omega_k \omega_l \psi_{m,n;k,l} \left\{ 1 - \frac{|\mathbf{u}|^2}{2RT} + \frac{(\boldsymbol{\xi}_{k,l} \cdot \mathbf{u})}{RT} + \frac{(\boldsymbol{\xi}_{k,l} \cdot \mathbf{u})^2}{2(RT)^2} \right\} \quad (2.55)$$

siendo  $\psi_{m,n;k,l} = \xi_{x,k}^m \xi_{y,l}^n$  y  $\boldsymbol{\xi}_{k,l} = (\xi_{x,k}, \xi_{y,l})$ . Para simplificar la notación en vez de utilizar un doble sumatorio se reduce a un único sumatorio de nueve elementos coincidentes con la cantidad de velocidades discretas:

$$I_{m,n} = \sum_{\alpha=0}^8 \frac{\rho}{\pi} \psi_{m,n;\alpha} w_{\alpha} \left\{ 1 - \frac{|\mathbf{u}|^2}{2RT} + \frac{(\boldsymbol{\xi}_{\alpha} \cdot \mathbf{u})}{RT} + \frac{(\boldsymbol{\xi}_{\alpha} \cdot \mathbf{u})^2}{2(RT)^2} \right\} \quad (2.56)$$

siendo  $\boldsymbol{\xi}_{\alpha}$  las velocidades discretas representadas en la Tabla (2.2) y  $w_{\alpha}$  los pesos asociados.

$$w_{\alpha} = \frac{\omega_k \omega_l}{\pi} = \begin{cases} \frac{4}{9} & k = l = 2 \quad \alpha = 0 \\ \frac{1}{9} & k = 1, l = 2; k = 2, l = 1; \dots \quad \alpha = 1, 2, 3, 4 \\ \frac{1}{36} & k = l = 1; k = l = 3; \dots \quad \alpha = 5, 6, 7, 8 \end{cases} \quad (2.57)$$

$\alpha$	0	1	2	3	4	5	6	7	8
$\xi_{\alpha,x}$	0	$\sqrt{3RT}$	0	$-\sqrt{3RT}$	0	$\sqrt{3RT}$	$-\sqrt{3RT}$	$-\sqrt{3RT}$	$\sqrt{3RT}$
$\xi_{\alpha,y}$	0	0	$\sqrt{3RT}$	0	$-\sqrt{3RT}$	$\sqrt{3RT}$	$\sqrt{3RT}$	$-\sqrt{3RT}$	$-\sqrt{3RT}$

Tabla 2.2: Representación de los componentes de las velocidades discretas del modelo D2Q9.

En la Figura (2.2) se representan las velocidades discretas.

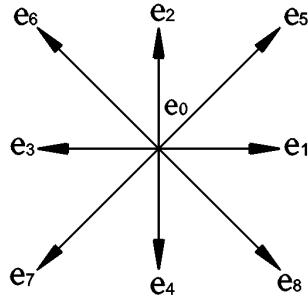


Figura 2.2: Representación del modelo de discretización D2Q9 del espacio de velocidades. Se representan los vectores de velocidad discretos.

A continuación se define la velocidad de la luz como  $c \equiv \frac{\delta x}{\delta t}$ , que en este caso vale  $c = \sqrt{3RT}$ . Por otro lado, se define la velocidad del sonido<sup>7</sup>  $c_s = \frac{\partial p}{\partial \rho} = \sqrt{RT}$ .

Identificando términos de la Ecuación (2.56) e introduciendo la definición de la velocidad de la luz se obtiene la distribución de equilibrio discretizada (en la cual se hace un cambio de notación para indicar las velocidades discretas  $\mathbf{e}_\alpha$ ):

$$f_\alpha^{(eq)} = w_\alpha \rho \left\{ 1 + \frac{3(\mathbf{e}_\alpha \cdot \mathbf{u})}{c^2} + \frac{9(\mathbf{e}_\alpha \cdot \mathbf{u})^2}{2c^4} - \frac{3|\mathbf{u}|^2}{2c^2} \right\} \quad (2.58)$$

A continuación se resumen las ecuaciones del modelo SRT para el modelo D2Q9:

$$f_\alpha(\mathbf{r} + \mathbf{e}_\alpha \delta t, t + \delta t) - f_\alpha(\mathbf{r}, t) = -\frac{1}{\tau} \left( f_\alpha(\mathbf{r}, t) - f_\alpha^{(eq)}(\mathbf{r}, t) \right) \quad (2.59a)$$

$$f_\alpha^{(eq)} = w_\alpha \rho \left\{ 1 + \frac{3(\mathbf{e}_\alpha \cdot \mathbf{u})}{c^2} + \frac{9(\mathbf{e}_\alpha \cdot \mathbf{u})^2}{2c^4} - \frac{3|\mathbf{u}|^2}{2c^2} \right\} \quad (2.59b)$$

$$\rho(\mathbf{r}, t) = \sum_{\alpha=0}^8 f_\alpha(\mathbf{r}, t) \quad (2.59c)$$

$$\rho(\mathbf{r}, t) \mathbf{u}(\mathbf{r}, t) = \sum_{\alpha=0}^8 \mathbf{e}_\alpha f_\alpha(\mathbf{r}, t) \quad (2.59d)$$

### Expansión multiescala de Chapman-Enskog

Llegados a este punto queda aún una pregunta importante sin responder: ¿cómo se sabe que las Ecuaciones (2.59) son equivalentes a las ecuaciones de Navier Stokes? Para responder a esta pregunta se utilizará la expansión multiescala de Chapman-Enskog que, básicamente, relaciona el tiempo de relajación del término de colisión BGK con las propiedades macroscópicas del fluido. La idea básica de la expansión de Chapman-Enskog es separar la función de distribución y sus derivadas respecto

---

<sup>7</sup>Suponiendo la validez de la ecuación del gas ideal ( $p = \rho RT$ ) y suponiendo que el gas es isentrópico ( $\frac{p}{\rho^\gamma} = cte.$ ) se obtiene  $c_s = \frac{\partial p}{\partial \rho} = \sqrt{\gamma RT}$ , mientras que si se supone que el gas es isotérmico ( $T = cte.$ ) se obtiene  $c_s = \frac{\partial p}{\partial \rho} = \sqrt{RT}$

el tiempo y el espacio en diferentes escalas caracterizadas por su orden de magnitud.

$$f_\alpha = \sum_{n=0}^{\infty} \epsilon^n f_\alpha^{(n)} \quad (2.60)$$

$$\frac{\partial}{\partial t} = \sum_{n=1}^{\infty} \epsilon^n \frac{\partial}{\partial t^{(n)}} \quad (2.61)$$

$$\frac{\partial}{\partial r_j} = \sum_{n=1}^{\infty} \epsilon^n \frac{\partial}{\partial r_j^{(n)}} \quad (2.62)$$

siendo  $\epsilon$  un parámetro adimensional para caracterizar el orden de magnitud del término. El primer término  $f_\alpha^{(0)}$  es por definición el estado de equilibrio  $f_\alpha^{(eq)}$  y, como consecuencia, el resto de términos caracterizan el estado de no equilibrio. Respecto a las derivadas, en vez de coger infinitos términos, es lógico pensar que en el dominio del espacio hace falta considerar únicamente una escala, mientras que en el dominio del tiempo sí que hace faltar tener en cuenta más escalas para diferenciar las escalas rápidas de las lentas. Finalmente se escogen una escala en el dominio del espacio y dos escalas en el dominio del tiempo quedando las derivadas de la siguiente manera:

$$\frac{\partial}{\partial t} \cong \epsilon \frac{\partial}{\partial t^{(1)}} + \epsilon^2 \frac{\partial}{\partial t^{(2)}} \quad (2.63)$$

$$\frac{\partial}{\partial r_j} \cong \epsilon \frac{\partial}{\partial r_j^{(1)}} \quad (2.64)$$

Para empezar, se desarrolla en serie de Taylor el término  $f_\alpha(\mathbf{r} + \xi_\alpha \delta t, t + \delta t)$  y se desprecian los términos superiores o iguales a  $\delta t^3$ :

$$\begin{aligned} f_\alpha(\mathbf{r} + \mathbf{e}_\alpha \delta t, t + \delta t) &= f_\alpha(\mathbf{r}, t) + \delta t \left( \frac{\partial}{\partial t} + \mathbf{e}_\alpha \cdot \nabla_{\mathbf{r}} \right) f_\alpha(\mathbf{r}, t) + \\ &\quad + \frac{(\delta t)^2}{2} \left( \frac{\partial}{\partial t} + \mathbf{e}_\alpha \cdot \nabla_{\mathbf{r}} \right)^2 f_\alpha(\mathbf{r}, t) + O(\delta t^3) \end{aligned} \quad (2.65)$$

Sustituyendo la Ecuación (2.65) en la Ecuación (2.59a) y dividiendo por  $\delta t$  se obtiene:

$$\begin{aligned} \left( \frac{\partial}{\partial t} + \mathbf{e}_\alpha \cdot \nabla_{\mathbf{r}} \right) f_\alpha(\mathbf{r}, t) + \frac{\delta t}{2} \left( \frac{\partial}{\partial t} + \mathbf{e}_\alpha \cdot \nabla_{\mathbf{r}} \right)^2 f_\alpha(\mathbf{r}, t) &= \\ = -\frac{1}{\tau \delta t} \left( f_\alpha(\mathbf{r}, t) - f_\alpha^{(eq)}(\mathbf{r}, t) \right) + O(\delta t^2) \end{aligned} \quad (2.66)$$

A continuación se sustituyen las derivadas aproximadas y los infinitos términos de la función de distribución en la Ecuación (2.66), y se separan las diferentes escalas asociadas a  $\epsilon, \epsilon^2, \dots$ :

$$\epsilon : \left( \frac{\partial}{\partial t^{(1)}} + \mathbf{e}_\alpha \cdot \nabla_{\mathbf{r}^{(1)}} \right) f_\alpha^{(0)} = -\frac{1}{\tau \delta t} f_\alpha^{(1)} + O(\delta t^2) \quad (2.67)$$

$$\begin{aligned} \epsilon^2 : & \frac{\partial f_\alpha^{(0)}}{\partial t^{(2)}} + \left( \frac{\partial}{\partial t^{(1)}} + \mathbf{e}_\alpha \cdot \nabla_{\mathbf{r}^{(1)}} \right) f_\alpha^{(1)} + \\ & + \frac{\delta t}{2} \left( \frac{\partial}{\partial t^{(1)}} + \mathbf{e}_\alpha \cdot \nabla_{\mathbf{r}^{(1)}} \right)^2 f_\alpha^{(0)} = -\frac{1}{\tau \delta t} f_\alpha^{(2)} + O(\delta t^2) \end{aligned} \quad (2.68)$$

Combinando las Ecuaciones (2.67) y (2.68) se obtiene:

$$\frac{\partial f_\alpha^{(0)}}{\partial t^{(2)}} + \left( 1 - \frac{1}{2\tau} \right) \left( \frac{\partial}{\partial t^{(1)}} + \mathbf{e}_\alpha \cdot \nabla_{\mathbf{r}^{(1)}} \right) f_\alpha^{(1)} = -\frac{1}{\tau \delta t} f_\alpha^{(2)} + O(\delta t^2) \quad (2.69)$$

Antes de proceder a determinar los momentos de las ecuaciones anteriores se va a deducir una propiedad importante que será de utilidad. La densidad se puede escribir como sigue:

$$\rho = \sum_{\alpha=0}^8 f_\alpha = \sum_{\alpha=0}^8 \sum_{n=0}^{\infty} \epsilon^n f_\alpha^{(n)} = f_\alpha^{(0)} + \sum_{\alpha=0}^8 \sum_{n=1}^{\infty} \epsilon^n f_\alpha^{(n)} \quad (2.70)$$

Teniendo en cuenta la propiedad (2.21) y que  $f_\alpha^{(0)} = f_\alpha^{(eq)}$ , se obtiene el siguiente resultado:

$$\sum_{\alpha=0}^8 f_\alpha^{(n)} = 0 \quad \forall n \geq 1 \quad (2.71)$$

La Ecuación (2.71) quiere decir que el momento de orden 0 de la parte de no equilibrio de la función de distribución vale 0. De forma análoga se puede deducir para los momentos de orden 1 y 2. A continuación, se evalúa el momento de orden 0 ( $\sum_{\alpha=0}^8$ ) de las Ecuaciones (2.67) y (2.69):

$$\frac{\partial \rho}{\partial t^{(1)}} + \nabla_{\mathbf{r}^{(1)}} \cdot (\rho \mathbf{u}) = O(\delta t^2) \quad (2.72a)$$

$$\frac{\partial \rho}{\partial t^{(2)}} = O(\delta t^2) \quad (2.72b)$$

Multiplicando la Ecuación (2.72a) por  $\epsilon$ , y sumándolo a la Ecuación (2.72b) multiplicada por  $\epsilon^2$ , se obtiene la ecuación de conservación de masa (salvo por el error de orden  $O(\delta t^2)$ ):

$$\frac{\partial \rho}{\partial t} + \nabla_{\mathbf{r}} \cdot (\rho \mathbf{u}) = O(\delta t^2) \quad (2.73)$$

A continuación se evalúa el momento de orden 1 ( $\sum_{\alpha=0}^8 \mathbf{e}_\alpha$ ) de las Ecuaciones (2.67) y (2.69) obteniendo el siguiente resultado:

$$\frac{\partial (\rho \mathbf{u})}{\partial t^{(1)}} + \nabla_{\mathbf{r}^{(1)}} \Pi^{(0)} = O(\delta t^2) \quad (2.74a)$$

$$\frac{\partial (\rho \mathbf{u})}{\partial t^{(2)}} + \left( 1 - \frac{1}{2\tau} \right) \nabla_{\mathbf{r}^{(1)}} \Pi^{(1)} = O(\delta t^2) \quad (2.74b)$$

siendo  $\Pi_{ij}^{(n)} = \sum_{\alpha=0}^8 e_{\alpha,i} e_{\alpha,j} f_{\alpha}^{(n)}$  el tensor de momento de segundo orden de la función de distribución  $f_{\alpha}^{(n)}$ . También se define el tensor de momento de tercer orden  $Q_{ijk}^{(n)} = \sum_{\alpha=0}^8 e_{\alpha,i} e_{\alpha,j} e_{\alpha,k} f_{\alpha}^{(n)}$ . Teniendo en cuenta la configuración de las velocidades discretas del modelo D2Q9 y la función de equilibrio (2.59b) se demuestra que:

$$\Pi_{ij}^{(0)} = \frac{1}{3} \rho c^2 \delta_{ij} + \rho u_i u_j \quad (2.75a)$$

$$Q_{ijk}^{(0)} = \frac{1}{3} \rho c^2 (u_k \delta_{ij} + u_j \delta_{ik} + u_i \delta_{jk}) \quad (2.75b)$$

$$\Pi_{ij}^{(1)} = -\tau \delta t \left( \frac{\partial \Pi_{ij}^{(0)}}{\partial t^{(1)}} + \nabla_{\mathbf{r}_k^{(1)}} \cdot Q_{ijk}^{(n)} \right) = -\tau \delta t c_s^2 \rho \left( \frac{\partial u_j}{\partial x_i^{(1)}} + \frac{\partial u_i}{\partial x_j^{(1)}} \right) + O(Ma^3) \quad (2.75c)$$

Identificando términos, se obtiene la siguiente expresión que relaciona la frecuencia de relajación adimensional  $\omega$  con la viscosidad cinemática  $\nu$ :

$$\nu = \frac{\delta x^2}{3\delta t} \left( \frac{1}{\omega} - \frac{1}{2} \right) \quad (2.76)$$

Para que la viscosidad cinemática sea positiva, la frecuencia de relajación debe ser inferior a 2, o dicho de otra manera, el tiempo de relajación debe ser superior a 0,5. Como se verá más adelante, cuando la frecuencia de relajación se acerca a 2 ocurren inestabilidades.

### Modelo incompresible

Como se ha visto anteriormente el modelo SRT D2Q9 con la función de distribución de equilibrio (2.59b) simula las ecuaciones compresibles de Navier-Stokes. Sin embargo, el número de Mach debe ser pequeño para respetar la validez de las ecuaciones obtenidas.

A continuación se va a obtener un modelo para los flujos incompresibles. Idealmente, cuando se habla de la incompresibilidad se refiere a que la densidad es constante. Sin embargo, en el modelo obtenido es prácticamente imposible mantener la densidad exactamente constante. Esta imposibilidad proviene básicamente del hecho de que la densidad y la presión no son dos variables independientes en los modelos LBM. La relación que se establece entre la presión y la densidad es la ecuación de estado de los gases ideales. Evidentemente esto implica que siempre se cometerá un error a la hora de simular las ecuaciones incompresibles de Navier-Stokes. Por ello, lo que se hace es proponer una modificación del modelo SRT D2Q9 para minimizar

los efectos de compresibilidad y, de esta manera, aproximar mejor las ecuaciones de Navier-Stokes incompresibles. Se basará en el método propuesto en [16]. El procedimiento se basa en separar la densidad en una parte constante ( $\rho_{ref}$ ) y en una parte fluctuante ( $\delta\rho$ ) que se demuestra que es de orden  $O(Ma^2)$ . Despreciando el error debido a los términos iguales o superiores a  $O(Ma^3)$  se obtiene el modelo incompresible.

$$f_\alpha(\mathbf{r} + \mathbf{e}_\alpha \delta t, t + \delta t) - f_\alpha(\mathbf{r}, t) = -\frac{1}{\tau} \left( f_\alpha(\mathbf{r}, t) - f_\alpha^{(eq)}(\mathbf{r}, t) \right) \quad (2.77a)$$

$$f_\alpha^{(eq)} = w_\alpha \left\{ \rho + \rho_{ref} \left( \frac{3(\mathbf{e}_\alpha \cdot \mathbf{u})}{c^2} + \frac{9(\mathbf{e}_\alpha \cdot \mathbf{u})^2}{2c^4} - \frac{3|\mathbf{u}|^2}{2c^2} \right) \right\} \quad (2.77b)$$

$$\rho(\mathbf{r}, t) = \rho_{ref} + \delta\rho = \sum_{\alpha=0}^8 f_\alpha(\mathbf{r}, t) \quad (2.77c)$$

$$\rho_{ref} \mathbf{u}(\mathbf{r}, t) = \sum_{\alpha=0}^8 \mathbf{e}_\alpha f_\alpha(\mathbf{r}, t) \quad (2.77d)$$

Normalmente se toma valor unitario para  $\rho_{ref}$ .

### Modelo térmico

Para poder simular problemas de convección natural es necesario incorporar el cálculo de la temperatura y también un término adicional de fuerza de flotación en la ecuación de momentum. En [22] se proponen tres maneras de incorporar la fuerza adicional. De las tres, se ha escogido la que añade simplemente un término extra  $F_\alpha$  a la función de equilibrio  $f_\alpha^{(eq)}$ :

$$F_\alpha = w_\alpha \mathbf{F} \cdot \mathbf{e}_\alpha \frac{1}{c_s^2} \quad (2.78)$$

siendo  $\mathbf{F}$  la fuerza que se calcula de la siguiente manera:  $\mathbf{F} = \rho \mathbf{g} \beta (T - T_0)$ , siendo  $T$  la temperatura y  $T_0$  una temperatura característica del problema (ver apartado de adimensionalización).

Por otro lado, para calcular el campo de la temperatura, se introduce otra función de distribución  $g$  (no confundir con la función de distribución de equilibrio). El modelo utilizado para la función de distribución  $g$  es también D2Q9.

Resumiendo, se tiene las siguientes ecuaciones:

$$f_\alpha(\mathbf{r} + \mathbf{e}_\alpha \delta t, t + \delta t) - f_\alpha(\mathbf{r}, t) = -\omega_f \left( f_\alpha(\mathbf{r}, t) - f_\alpha^{(eq)}(\mathbf{r}, t) \right) \quad (2.79a)$$

$$g_\alpha(\mathbf{r} + \mathbf{e}_\alpha \delta t, t + \delta t) - g_\alpha(\mathbf{r}, t) = -\omega_g \left( g_\alpha(\mathbf{r}, t) - g_\alpha^{(eq)}(\mathbf{r}, t) \right) \quad (2.79b)$$

$$f_\alpha^{(eq)} = w_\alpha \rho \left\{ 1 + \frac{3(\mathbf{e}_\alpha \cdot \mathbf{u})}{c^2} + \frac{9(\mathbf{e}_\alpha \cdot \mathbf{u})^2}{2c^4} - \frac{3|\mathbf{u}|^2}{2c^2} \right\} + F_\alpha \quad (2.79c)$$

$$g_\alpha^{(eq)} = w_\alpha T \left\{ 1 + \frac{3(\mathbf{e}_\alpha \cdot \mathbf{u})}{c^2} + \frac{9(\mathbf{e}_\alpha \cdot \mathbf{u})^2}{2c^4} - \frac{3|\mathbf{u}|^2}{2c^2} \right\} \quad (2.79d)$$

$$\rho(\mathbf{r}, t) = \sum_{\alpha=0}^8 f_\alpha(\mathbf{r}, t) \quad (2.79e)$$

$$\rho(\mathbf{r}, t) \mathbf{u}(\mathbf{r}, t) = \sum_{\alpha=0}^8 \mathbf{e}_\alpha f_\alpha(\mathbf{r}, t) \quad (2.79f)$$

$$T(\mathbf{r}, t) = \sum_{\alpha=0}^8 g_\alpha(\mathbf{r}, t) \quad (2.79g)$$

$$(2.79h)$$

Antes se tenía únicamente una frecuencia de relajación adimensional para la función de distribución  $f$ , en cambio, ahora se tienen dos, una para la función de distribución  $f$  y la otra para  $g$ :

$$\omega_f = \frac{1}{3\nu + 0,5} \quad (2.80a)$$

$$\omega_g = \frac{1}{3a + 0,5} \quad (2.80b)$$

siendo  $a$  la difusividad térmica definida anteriormente y que está relacionada con el número de Prandtl y la viscosidad cinemática (ver apartado de adimensionalización).

### 2.2.2. Multiple Relaxation Time (MRT)

Debido a su simplicidad, el modelo SRT con el operador de colisión BGK, también llamado LBGK, se ha convertido en el método más popular en LBM a pesar de presentar problemas de estabilidad cuando el número de Reynolds es alto. El modelo MRT LBE es la generalización del modelo LBGK que a diferencia del otro tiene más de un parámetro de relajación, y además se ha demostrado su superioridad en el aspecto de estabilidad [14].

Al igual que el modelo LBGK, MRT LBE tiene un conjunto de velocidades discretas  $\{\mathbf{e}_0, \mathbf{e}_1, \dots, \mathbf{e}_{q-1}\}$  y un conjunto de funciones de distribución para cada velocidad

discreta  $\{f_0, f_1, \dots, f_{q-1}\}$ . La diferencia está en la manera de tratar el proceso de colisión. En MRT LBE el proceso de colisión se realiza a través de la matriz de colisión  $S$  de dimensión  $q \times q$ . De esta manera la ecuación de evolución de las funciones de distribución en el tiempo discreto es:

$$\left\{ \begin{array}{l} f_0(\mathbf{r} + \mathbf{e}_0 \delta t, t + \delta t) - f_0(\mathbf{r}, t) \\ f_1(\mathbf{r} + \mathbf{e}_1 \delta t, t + \delta t) - f_1(\mathbf{r}, t) \\ \vdots \\ f_{q-1}(\mathbf{r} + \mathbf{e}_{q-1} \delta t, t + \delta t) - f_{q-1}(\mathbf{r}, t) \end{array} \right\} = - \left[ \begin{array}{ccc} S_{11} & \cdots & S_{1q} \\ \vdots & \ddots & \vdots \\ S_{q1} & \cdots & S_{qq} \end{array} \right] \left\{ \begin{array}{l} f_0(\mathbf{r}, t) - f_0^{(eq)}(\mathbf{r}, t) \\ f_1(\mathbf{r}, t) - f_1^{(eq)}(\mathbf{r}, t) \\ \vdots \\ f_{q-1}(\mathbf{r}, t) - f_{q-1}^{(eq)}(\mathbf{r}, t) \end{array} \right\} \quad (2.81)$$

que escrito en forma vectorial adquiere un aspecto más compacto:

$$\mathbf{f}(\mathbf{r} + \mathbf{e}_\alpha \delta t, t + \delta t) - \mathbf{f}(\mathbf{r}, t) = -S \left( \mathbf{f}(\mathbf{r}, t) - \mathbf{f}^{(eq)}(\mathbf{r}, t) \right) \quad (2.82)$$

Las frecuencias de relajación son los valores propios de la matriz de colisión  $S$ . En el caso de tener todos los valores propios iguales, la matriz de colisión adquiere la forma  $S = \omega I$  (siendo  $I$  la matriz identidad) y se recupera el modelo LBGK.

En modelos MRT habrán  $q$  momentos  $\mathbf{m}$  que son combinaciones lineales de las funciones de distribución y, por tanto, se pueden representar como una transformación lineal mediante una matriz  $M$  de dimensión  $q \times q$ :

$$\mathbf{m}(\mathbf{r}, t) = M \mathbf{f}(\mathbf{r}, t) \quad (2.83)$$

Si se escoge de manera adecuada la matriz  $S$  se puede simplificar la Ecuación (2.82):

$$\mathbf{f}(\mathbf{r} + \mathbf{e}_\alpha \delta t, t + \delta t) - \mathbf{f}(\mathbf{r}, t) = -M^{-1} \hat{S} \left( \mathbf{m}(\mathbf{r}, t) - \mathbf{m}^{(eq)}(\mathbf{r}, t) \right) \quad (2.84)$$

siendo  $\hat{S} = MSM^{-1}$  una matriz diagonal:  $\hat{S} = diag(s_0, s_1, \dots, s_{q-1})$ . Los  $q$  momentos se pueden dividir en dos grupos: los momentos hidrodinámicos (conservados) y los momentos cinéticos (no conservados). El primer grupo consiste en momentos localmente conservados en el proceso de colisión, es decir,  $\mathbf{m}(\mathbf{r}, t) = \mathbf{m}^{(eq)}(\mathbf{r}, t)$ . El segundo grupo consiste en momentos no conservados en el proceso de colisión, es

decir,  $\mathbf{m}(\mathbf{r}, t) \neq \mathbf{m}^{(eq)}(\mathbf{r}, t)$ . Para las simulaciones de flujos isotérmicos, los momentos conservados son la densidad de masa  $\rho$  y la densidad de momentum  $\mathbf{j} = \rho\mathbf{u}$ . Los valores  $s_\alpha$  correspondientes a los momentos conservados valen 0 mientras que los otros son parámetros de ajuste de la simulación. La descripción más detallada de los modelos MRT en 2D se puede encontrar en [15], mientras que para los modelos MRT en 3D en [14].

### 2.3. Aspectos de implementación

En el apartado anterior se han introducido los diferentes modelos de LBM. En particular, se ha presentado con mucho detalle la deducción de la Ecuación de Boltzmann discretizada y las funciones de equilibrio para el modelo D2Q9 SRT. A partir de ahora se centra en el modelo D2Q9 SRT ya que, como ya se ha dicho anteriormente, es el modelo más utilizado. En este apartado se van a discutir las diferentes maneras de implementar el LBM incluyendo el tratamiento de las condiciones de contorno.

En el apartado anterior se ha discretizado la Ecuación de Boltzmann en el espacio de fases (de velocidades microscópicas) y también en el dominio de tiempo, y se ha obtenido la siguiente ecuación:

$$f_\alpha(\mathbf{r} + \mathbf{e}_\alpha \delta t, t + \delta t) = f_\alpha(\mathbf{r}, t) - \omega \left( f_\alpha(\mathbf{r}, t) - f_\alpha^{(eq)}(\mathbf{r}, t) \right) \quad (2.85)$$

Para discretizar en el dominio del espacio de manera uniforme basta con modificar el vector de posición de la siguiente manera:

$$\mathbf{r} \rightarrow \mathbf{r}_{ij} = i\delta x \mathbf{e}_x + j\delta y \mathbf{e}_y \quad (2.86)$$

siendo  $\mathbf{e}_x$  y  $\mathbf{e}_y$  los vectores ortonormales que definen la base en el espacio,  $\delta x$  y  $\delta y$  las distancias entre dos nodos consecutivos en las dos direcciones del espacio y por último,  $i$  y  $j$  números enteros que permiten apuntar el vector posición a cualquier nodo en el espacio. A partir de ahora se entenderá que el vector posición  $\mathbf{r}$  es la versión discretizada  $\mathbf{r}_{ij}$ . Para cada nodo ubicado en  $\mathbf{r}$  habrán  $q$  valores de la función de distribución llamados  $f_\alpha$ , donde  $\alpha$  va de 0 a 8, y que caracterizan el estado local del nodo.

En la Figura (2.3) se representa un ejemplo de discretización con el modelo D2Q9.

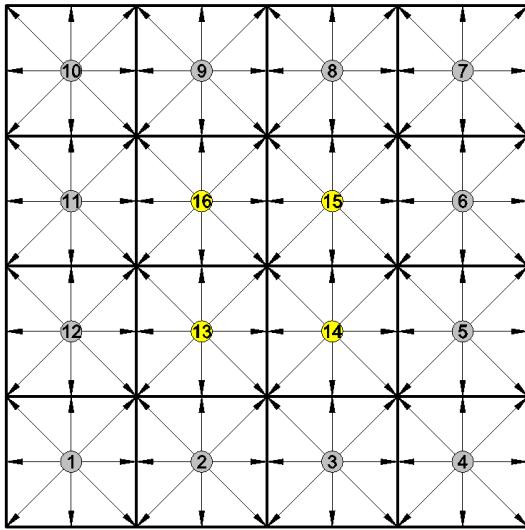


Figura 2.3: Discretización de un dominio cuadrado mediante el modelo D2Q9. Los nodos amarillos representan el dominio de fluido, mientras que los nodos gris definen el contorno. Las flechas representan los vectores de velocidad microscópica para cada nodo  $e_\alpha$ .

Normalmente, para inicializar los valores  $f_\alpha$  para cada nodo se supone que el estado inicial es el correspondiente al de equilibrio con una cierta velocidad y densidad dadas, y por tanto se inicializa tal y como se muestra a continuación:

$$f_\alpha(\mathbf{r}, t = 0) = f_\alpha^{(eq)} \Big|_{\substack{\mathbf{u}=\mathbf{u}_0 \\ \rho=\rho_0}} \quad (2.87)$$

Una vez se tienen todos los valores  $f_\alpha$  para cada nodo que forma el dominio físico en el tiempo  $t$  (incluido la inicial  $t = 0$ ), se quieren calcular los valores de  $f_\alpha$  para cada nodo en el siguiente instante de tiempo  $t + \delta t$ . Es evidente que se tiene que aplicar la Ecuación (2.85), sin embargo, aplicar dicha ecuación de forma directa no es posible ya que los nodos que forman la condición de contorno deben recibir un tratamiento especial. Por este motivo, se separa la Ecuación (2.85) en dos etapas llamadas de propagación (streaming) y de colisión. Durante la primera etapa se produce una propagación de las funciones de distribución según las direcciones microscópicas a los nodos vecinos. Si se utilizan dos matrices (una correspondiente al estado de prepropagación y otra correspondiente al estado postpropagación) se puede realizar la etapa de propagación con un único ciclo barriendo todos los nodos y propagando (push) o bien recibiendo (pull) las funciones de distribución. Dichos métodos se representan en las Figuras (2.4) y (2.5).

Si se decide utilizar una única matriz, entonces se debe hacer más de un reco-

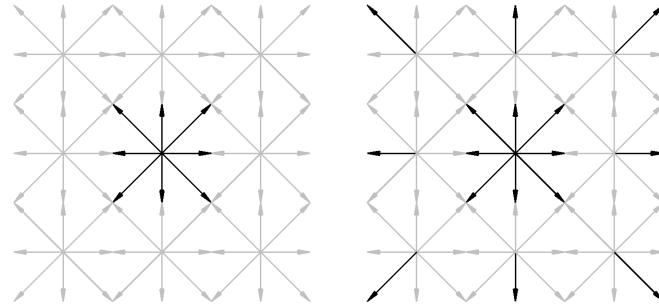


Figura 2.4: Representación del método push. A la izquierda y a la derecha se representan el estado pre y postpropagación para un nodo, respectivamente.

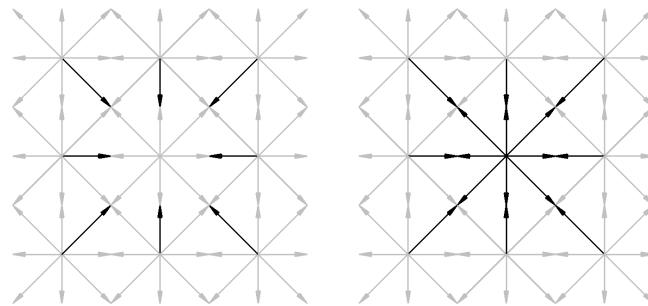


Figura 2.5: Representación del método pull. A la izquierda y a la derecha se representan el estado pre y postpropagación para un nodo, respectivamente.

rrido por todos los nodos con el fin de realizar la etapa de propagación sin perder información. Dicho método se representa en la Figura (2.6).

En ambos casos, utilizando una única matriz o bien dos matrices, en el estado de postpropagación habrán direcciones en los nodos del contorno en las que no se conocerá el valor de la función de distribución. Dichas direcciones se ilustran con flechas grises en la Figura (2.7).

El papel de las condiciones de contorno es, conociendo las propiedades macroscópicas en el contorno, determinar las funciones de distribución que no han quedado definidas después del proceso de propagación. El tratamiento de las condiciones de contorno se discutirá en los siguientes apartados.

En función de si primero se realiza la etapa de propagación y posteriormente la de colisión o viceversa, se obtienen dos esquemas diferentes de implementación:

#### **Esquema propagación-colisión**

1. El ciclo empieza para el instante  $t$  propagando las funciones de distribución según las direcciones microscópicas a los nodos vecinos. Matemáticamente, dicho

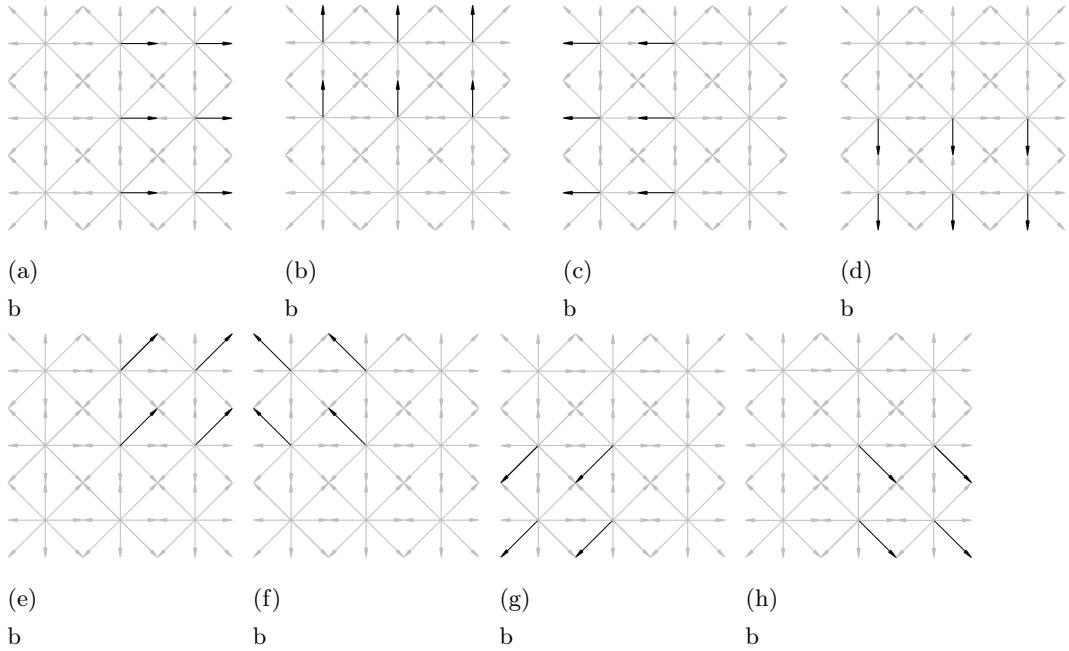


Figura 2.6: En el caso de utilizar una única matriz, el sentido de barrido difiere según la dirección. Las flechas negras indican los valores de la función de distribución propagados después de la etapa de propagación para cada dirección.

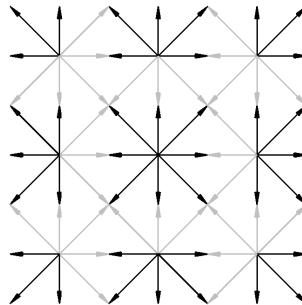


Figura 2.7: Una vez realizada la etapa de propagación hay ciertas direcciones (flechas grises) en las que se desconoce el valor de la función de distribución.

proceso se escribe de la siguiente manera:

$$f_{\alpha}^{post}(\mathbf{r} + \mathbf{e}_{\alpha}\delta t, t) = f_{\alpha}(\mathbf{r}, t) \quad (2.88)$$

siendo  $f_{\alpha}^{post}$  el estado postpropagación en el instante  $t$ .

2. Se aplican las condiciones de contorno con el fin de determinar los valores de  $f_{\alpha}^{post}(\mathbf{r}, t)$ , para  $\mathbf{r}$  perteneciente al contorno, que no han quedado definidos después de la etapa de propagación.

3. Una vez calculados todos los valores  $f_\alpha^{post}$  se procede a calcular las propiedades macroscópicas para las funciones de distribución  $f_\alpha^{post}$  para todos los nodos:

$$\rho(\mathbf{r}, t) = \sum_{\alpha} f_{\alpha}^{post} \quad (2.89)$$

$$u(\mathbf{r}, t) = \frac{1}{\rho(\mathbf{r}, t)} \sum_{\alpha} (\mathbf{e}_{\alpha} \cdot \mathbf{e}_x) f_{\alpha}^{post} \quad (2.90)$$

$$v(\mathbf{r}, t) = \frac{1}{\rho(\mathbf{r}, t)} \sum_{\alpha} (\mathbf{e}_{\alpha} \cdot \mathbf{e}_y) f_{\alpha}^{post} \quad (2.91)$$

4. A continuación se calculan las funciones de distribución de equilibrio  $f_{\alpha}^{(eq)}(\mathbf{r}, t)$  utilizando las propiedades macroscópicas ya calculadas.
5. Por último, se realiza la etapa de colisión en la que se actualizan todos los valores de  $f_{\alpha}$  para el siguiente instante de tiempo  $t + \delta t$ , y por tanto se cierra el ciclo:

$$f_{\alpha}(\mathbf{r}, t + \delta t) = f_{\alpha}^{post}(\mathbf{r}, t) - \omega \left( f_{\alpha}^{post}(\mathbf{r}, t) - f_{\alpha}^{(eq)}(\mathbf{r}, t) \right) \quad (2.92)$$

### Esquema colisión-propagación

1. El ciclo empieza para el instante  $t$  calculando las propiedades macroscópicas para las funciones de distribución  $f_{\alpha}$  para todos los nodos:

$$\rho(\mathbf{r}, t) = \sum_{\alpha} f_{\alpha} \quad (2.93)$$

$$u(\mathbf{r}, t) = \frac{1}{\rho(\mathbf{r}, t)} \sum_{\alpha} (\mathbf{e}_{\alpha} \cdot \mathbf{e}_x) f_{\alpha} \quad (2.94)$$

$$v(\mathbf{r}, t) = \frac{1}{\rho(\mathbf{r}, t)} \sum_{\alpha} (\mathbf{e}_{\alpha} \cdot \mathbf{e}_y) f_{\alpha} \quad (2.95)$$

2. A continuación se calculan las funciones de distribución de equilibrio  $f_{\alpha}^{(eq)}(\mathbf{r}, t)$  utilizando las propiedades macroscópicas ya calculadas.
3. Se realiza la etapa de colisión en la que se actualizan todos los valores de  $f_{\alpha}$  y de esta manera se llega al estado de postcolisión  $f_{\alpha}^{post}$ :

$$f_{\alpha}^{post}(\mathbf{r}, t) = f_{\alpha}(\mathbf{r}, t) - \omega \left( f_{\alpha}(\mathbf{r}, t) - f_{\alpha}^{(eq)}(\mathbf{r}, t) \right) \quad (2.96)$$

4. Se realiza la etapa de propagación:

$$f_{\alpha}(\mathbf{r} + \mathbf{e}_{\alpha} \delta t, t + \delta t) = f_{\alpha}^{post}(\mathbf{r}, t) \quad (2.97)$$

5. Por último, se aplican las condiciones de contorno con el fin de determinar los valores de  $f_\alpha(\mathbf{r}, t + \delta t)$  para  $\mathbf{r}$  perteneciente al contorno, y que no han quedado definidas después de la etapa de propagación.

### 2.3.1. Tratamiento de las condiciones de contorno

Cada modelo LBM puede llegar a tener un tratamiento diferente de las condiciones de contorno. Dependiendo de si se trata de un caso bidimensional o tridimensional, y también dependiendo de la cantidad de velocidades discretas, el tratamiento de las condiciones de contorno es diferente. Aun fijando un modelo LBM, la dimensión del problema y la cantidad de velocidades discretas, existen varios tratamientos de las condiciones de contorno propuestas por diferentes autores. Debido a tal diversidad, en este apartado se centrará únicamente sobre el modelo LBGK D2Q9 y se discutirán algunos de los tratamientos más conocidos.

Antes de entrar en la discusión de los diferentes métodos de tratamiento de las condiciones de contorno, se introduce un concepto importante respecto a la clasificación del tipo de nodo. Los nodos pueden ser clasificados como wet (mojado) o dry (seco). Un nodo dry es aquel que se encuentra infinitamente cerca de la pared y forma parte de la condición de contorno, o bien está dentro de un cuerpo y no forma parte de la condición de contorno. Por tanto un nodo dry no forma parte del dominio fluido y no se ejecuta la etapa de colisión tradicional (en el sentido de un proceso de relajación de las funciones de distribución). Por otro lado, un nodo wet sí que se encuentra en el dominio fluido y puede o no formar parte de la condición de contorno. Por tanto en un nodo wet sí que se ejecuta la etapa de colisión tradicional.

#### Bounce-Back

Bounce-Back (BB) es la condición de contorno de pared más famosa en LBM. Se basa en la idea intuitiva de que las funciones de distribución que se dirigen a la pared rebotan al tocarla y vuelven al dominio fluido. Hay dos tipos diferentes: BB Half-Way y BB Full-Way. En el caso de BB Half-Way se debe utilizar el orden propagación-colisión en el que después de la etapa de propagación tradicional se produce la sustitución de las funciones de distribución desconocidas mediante un proceso de inversión para los nodos del contorno. Por el contrario, en el caso de BB Full-Way se debe utilizar el orden colisión-propagación en el que se produce la inversión en todas las direcciones sin importar la orientación de la pared durante la

etapa de colisión. En BB Half-Way, el nodo de contorno es de tipo wet mientras que en BB Full-Way, es de tipo dry. A continuación se representa la expresión que rige la condición de contorno de tipo BB Half-Way:

$$f_{\bar{\alpha}}^{post}(\mathbf{r}, t) = f_{\alpha}^{post}(\mathbf{r}, t) \quad (2.98)$$

siendo  $f_{\alpha}^{post}$  el estado postpropagación y  $\bar{\alpha}$  la dirección inversa de  $\alpha$  ( $e_{\bar{\alpha}} = -e_{\alpha}$ ). Hay que especificar que  $\mathbf{r}$  apunta a los nodos de contorno y  $\alpha$  son las direcciones en las que se da el rebote, y que dependen de la orientación de la pared.

La ecuación que rige la condición de contorno de tipo BB Full-Way es:

$$f_{\bar{\alpha}}(\mathbf{r}, t + \delta t) = f_{\alpha}(\mathbf{r}, t) \quad (2.99)$$

para todos los valores de  $\alpha$ .

Las Ecuaciones (2.98-2.99) son para condición de contorno de velocidad nula. Para imponer una cierta velocidad  $\mathbf{u}_w$  se añade un término extra  $-2\rho w_{\alpha} \frac{e_{\alpha} \cdot \mathbf{u}_w}{c_s^2}$  a la derecha de la Ecuación (2.99) [13].

### Zou & He

En este método los nodos que forman parte del contorno son de tipo wet y, en el caso de una condición de contorno horizontal, tal y como se muestra en la Figura (2.8), después del proceso de propagación los valores  $f_2$ ,  $f_5$  y  $f_6$  quedan indeterminados. Para obtener los valores indeterminados se dispone de las siguientes

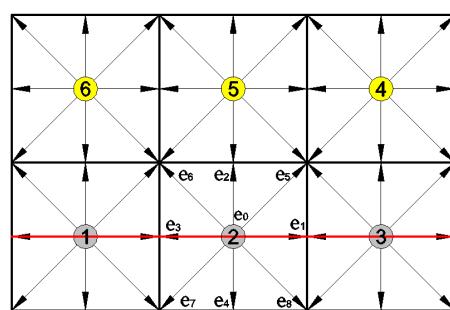


Figura 2.8: Representación de un contorno horizontal. La línea roja representa la pared y los nodos que están encima de la línea roja representan los nodos en los que hay que aplicar la condición de contorno.

ecuaciones:

$$\rho = f_0 + f_1 + f_2 + f_3 + f_4 + f_5 + f_6 + f_7 + f_8 \quad (2.100a)$$

$$\rho u = f_1 + f_5 + f_8 - f_6 - f_3 - f_7 \quad (2.100b)$$

$$\rho v = f_5 + f_2 + f_6 - f_8 - f_4 - f_7 \quad (2.100c)$$

Por tanto, habrán 6 variables (las tres distribuciones de funciones, los dos componentes de la velocidad y la densidad (presión)) y 3 ecuaciones. En consecuencia, para resolver las incógnitas se tendrían que proporcionar 3 valores de las 6 variables en la condición de contorno. Es evidente que falta otra ecuación ya que, por ejemplo, en el caso de una condición de contorno de velocidad nula no se conocerá la densidad. En el trabajo de Q. Zou y X. He [9] se propone utilizar como tercera ecuación la condición de rebote de la parte de no equilibrio de la función de distribución. Para el caso horizontal representado en la Figura (2.8) la ecuación adicional será:

$$f_2^{(neq)} = f_4^{(neq)} \implies f_2 - f_2^{(eq)} = f_4 - f_4^{(eq)} \quad (2.101)$$

Teniendo la ecuación anterior en cuenta pueden haber los siguientes casos:

*Conocidos los dos componentes de la velocidad, encontrar la densidad (presión) y las funciones de distribución desconocidas.* De las Ecuaciones (2.100a) y (2.100c) se despeja el término  $f_5 + f_2 + f_6$  y se igualan entre ellos. De esta manera se obtiene la densidad:

$$\rho = \frac{1}{1-v} [f_0 + f_1 + f_3 + 2(f_4 + f_7 + f_8)] \quad (2.102)$$

A continuación se remplazan las expresiones de la función de distribución de equilibrio  $f_2^{(eq)}$  y  $f_4^{(eq)}$  dentro de la Ecuación (2.101) obteniendo el siguiente resultado:

$$f_2 = f_4 + \frac{2}{3}\rho v \quad (2.103)$$

Por último, resolviendo las Ecuaciones (2.100b) y (2.100c) se obtienen las funciones de distribución  $f_5$  y  $f_6$ :

$$f_5 = f_7 - \frac{1}{2}(f_1 - f_3) + \frac{1}{2}\rho u + \frac{1}{6}\rho v \quad (2.104a)$$

$$f_5 = f_8 + \frac{1}{2}(f_1 - f_3) - \frac{1}{2}\rho u + \frac{1}{6}\rho v \quad (2.104b)$$

*Conocidos la densidad (presión) y el componente tangencial (a la condición de contorno) de la velocidad, determinar la velocidad normal y las funciones de distribución desconocidas.* Entre los dos casos lo único que cambia es el primer paso, siendo

para el primer caso el cálculo de la densidad (presión), y para el segundo el cálculo de la velocidad normal. De las Ecuaciones (2.100a) y (2.100c) se despeja el término  $f_5 + f_2 + f_6$  y se igualan entre ellos. De esta manera se obtiene la componente vertical de la velocidad:

$$v = 1 - \frac{1}{\rho} [f_0 + f_1 + f_3 + 2(f_4 + f_7 + f_8)] \quad (2.105)$$

Se utilizan las Ecuaciones (2.103) y (2.104) para obtener las funciones de distribución desconocidas.

Los nodos que se encuentran en las esquinas reciben un tratamiento particular ya que habrá dos direcciones más en las que se desconocerán los valores de la función de distribución y, en consecuencia, se tendrían que proporcionar dos ecuaciones más para poder determinar todas las incógnitas en el nodo de contorno. Una incorrecta aplicación de la condición de contorno a dichos nodos implicará la propagación del error por todo el dominio a través de la etapa de propagación. En la Figura (2.9) se

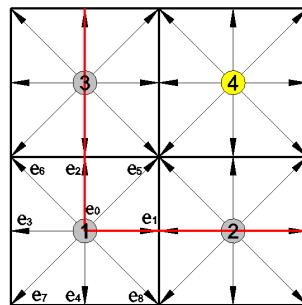


Figura 2.9: Representación de un contorno esquina. La línea roja representa la pared y los nodos que están encima de la línea roja representan los nodos en los que hay que aplicar la condición de contorno.

representa un nodo que se encuentra en la intersección de un contorno horizontal y vertical. En este caso se aplica la misma idea propuesta en el trabajo de Q. Zou y X. He [CC1]. Habrá dos direcciones(1 y 2) en las qué se producirá el rebote de las partes de no equilibrio de la función de distribución.

$$f_2^{(neq)} = f_4^{(neq)} \quad (2.106a)$$

$$f_1^{(neq)} = f_3^{(neq)} \quad (2.106b)$$

Sustituyendo las expresiones de equilibrio en las Ecuaciones (2.106) se obtiene:

$$f_5 = f_7 + \frac{1}{6}\rho(u+v) \quad (2.107a)$$

$$f_2 = f_4 + \frac{2}{3}\rho v \quad (2.107b)$$

$$f_1 = f_3 + \frac{2}{3}\rho u \quad (2.107c)$$

Suponiendo que la condición de contorno es de velocidad nula, las Ecuaciones (2.107) se convierten en  $f_2 = f_4$ ,  $f_1 = f_3$  y  $f_5 = f_7$ . A partir de las Ecuaciones (2.100b) y (2.100c) se deduce que  $f_6 = f_8$  y  $f_5 = f_7$ . Por último, de la Ecuación (2.100a) se obtiene el valor de  $f_6$  y  $f_8$ :

$$f_6 = f_8 = \frac{1}{2}[\rho - (f_0 + f_1 + f_2 + f_3 + f_4 + f_5 + f_7)] \quad (2.108)$$

Por tanto, conociendo la densidad (presión) se podrá obtener  $f_6$  y  $f_8$ . En el caso de desconocer el valor de la densidad (presión) se puede extrapolar a partir de los nodos vecinos.

### Chih-Fung et al.

La solución propuesta en el apartado anterior tiene ciertos inconvenientes debido a que la hipótesis del rebote de la parte de no equilibrio de la función de distribución viola las leyes de conservación y hace que la densidad total aumente. También es más laboriosa a la hora de resolver los nodos en las esquinas.

La idea principal de [10], que es la misma que en [9], consiste en encontrar las funciones de distribución desconocidas después del proceso de propagación. Sin embargo, las ecuaciones adicionales que se introducen suponen que las funciones de distribución desconocidas son funciones de las funciones de distribución locales y del vector corrector. Posteriormente se ajustará el vector corrector para cumplir las leyes de conservación (de masa y de momento). Matemáticamente se escribe como:

$$f_\alpha(\mathbf{r}, t) = f_\alpha^*(\mathbf{r}, t) + w_\alpha \mathbf{e}_\alpha \cdot \mathbf{Q} \quad (2.109)$$

siendo  $f_\alpha(\mathbf{r}, t)$  la función de distribución desconocida en la dirección  $\alpha$ ,  $f_\alpha^*(\mathbf{r}, t)$  la función de distribución local y  $\mathbf{Q}$  es el vector corrector con componentes  $Q_x$ ,  $Q_y$  y  $Q_z$  (para el caso de un modelo 3D). Para escoger  $f_\alpha^*(\mathbf{r}, t)$  hay varias opciones:  $f_\alpha^*(\mathbf{r}, t) = f_{\bar{\alpha}}(\mathbf{r}, t)$ ,  $f_\alpha^*(\mathbf{r}, t) = f_\alpha(\mathbf{r}, t - \delta t)$  o  $f_\alpha^*(\mathbf{r}, t) = f_\alpha^{(eq)}(\mathbf{r}, t)$ . Según el autor del [10], la diferencia de resultados debido a la diferente elección de  $f_\alpha^*(\mathbf{r}, t)$  es despreciable.

Como ejemplo se va a resolver un nodo de la cara norte donde las funciones de distribución  $f_4$ ,  $f_7$  y  $f_8$  quedan indefinidas después de la etapa de propagación. Aplicando la Ecuación (2.109) para las direcciones donde las funciones de distribución quedan indefinidas se obtienen:

$$f_4 = f_4^* - w_4 Q_y \quad (2.110a)$$

$$f_7 = f_7^* - w_7 (Q_x + Q_y) \quad (2.110b)$$

$$f_8 = f_8^* + w_8 (Q_x - Q_y) \quad (2.110c)$$

A continuación, sustituyendo las Ecuaciones (2.110) en las Ecuaciones (2.100) y despejando  $Q_x$  y  $Q_y$  se obtiene:

$$Q_x = \frac{1}{18} (\rho u - f_1 - f_5 - f_8^* + f_6 + f_3 + f_7^*) \quad (2.111a)$$

$$Q_y = \frac{1}{18} (\rho v - f_5 - f_2 - f_6 + f_8^* + f_4 + f_7^*) \quad (2.111b)$$

### Inamuro et al.

En el trabajo [11] Inamuro et al. proponen un manera diferente de abordar el problema de las condiciones de contorno de pared. La idea se basa en la utilización de la función de distribución de equilibrio para obtener las funciones de distribución indeterminadas después de la etapa de propagación. Se demostrará su uso para una pared horizontal, representada en la Figura (2.8), donde las funciones  $f_2$ ,  $f_5$  y  $f_6$  son las que hay que determinar. Si se intentan evaluar las funciones de equilibrio con la velocidad ( $\mathbf{u}_w$ ) y la densidad ( $\rho_w$ ) de la pared y asignarlas a las funciones de distribución indeterminadas se puede demostrar que la velocidad obtenida en la pared no es exactamente la velocidad impuesta. Para hacer que la velocidad impuesta a través de la función de distribución de equilibrio concuerde con la velocidad calculada a partir de los momentos se introducen dos variables  $\rho'$  y  $u'$ . Se calculan  $\rho'$  y  $u'$  de tal manera que cumplan las siguientes ecuaciones:

$$\rho_w = f_0 + f_1 + f_2^{(eq)} \Big|_{\substack{u=u_w+u' \\ v=v_w \\ \rho=\rho'}} + f_3 + f_4 + f_5^{(eq)} \Big|_{\substack{u=u_w+u' \\ v=v_w \\ \rho=\rho'}} + f_6^{(eq)} \Big|_{\substack{u=u_w+u' \\ v=v_w \\ \rho=\rho'}} + f_7 + f_8 \quad (2.112a)$$

$$\rho_w u_w = f_1 + f_5^{(eq)} \Big|_{\substack{u=u_w+u' \\ v=v_w \\ \rho=\rho'}} + f_8 - f_6^{(eq)} \Big|_{\substack{u=u_w+u' \\ v=v_w \\ \rho=\rho'}} - f_3 - f_7 \quad (2.112b)$$

$$\rho_w v_w = f_5^{(eq)} \Big|_{\substack{u=u_w+u' \\ v=v_w \\ \rho=\rho'}} + f_2^{(eq)} \Big|_{\substack{u=u_w+u' \\ v=v_w \\ \rho=\rho'}} + f_6^{(eq)} \Big|_{\substack{u=u_w+u' \\ v=v_w \\ \rho=\rho'}} - f_8 - f_4 - f_7 \quad (2.112c)$$

Resolviendo las Ecuaciones anteriores se obtienen la densidad de pared y las dos variables introducidas:

$$\rho_w = \frac{1}{1 - v_w} [f_0 + f_1 + f_3 + 2(f_4 + f_7 + f_8)] \quad (2.113a)$$

$$\rho' = 6 \frac{\rho_w v_w + f_4 + f_7 + f_8}{1 + 3v_w + 3v_w^2} \quad (2.113b)$$

$$u' = \frac{1}{1 + 3v_w} \left( 6 \frac{\rho_w u_w - f_1 + f_3 - f_8 + f_7}{\rho'} - u_w - 3u_w v_w \right) \quad (2.113c)$$

Una vez conocidas la densidad de pared y las dos variables introducidas, se evalúan las funciones de distribución de equilibrio y se asignan a las funciones de distribución desconocidas. El tratamiento de las esquinas es un poco más laborioso y se puede encontrar en [11].

### Condición de contorno para dominios abiertos

En muchas ocasiones pueden haber casos en los que hayan contornos en los que se desconozcan las propiedades macroscópicas (como por ejemplo, la velocidad). Para encontrar las funciones de distribución que quedan indeterminadas después de la etapa de propagación se utilizan métodos de extrapolación. Por ejemplo si se trata del contorno que está ubicado en la posición  $i = Nx$ , entonces la extrapolación cuadrática será:

$$f_3(Nx, j) = 2f_3(Nx - 1, j) - f_3(Nx - 2, j) \quad (2.114a)$$

$$f_6(Nx, j) = 2f_6(Nx - 1, j) - f_6(Nx - 2, j) \quad (2.114b)$$

$$f_7(Nx, j) = 2f_7(Nx - 1, j) - f_7(Nx - 2, j) \quad (2.114c)$$

La extrapolación cuadrática puede llevar a inestabilidades y es preferible utilizar la extrapolación lineal:

$$f_3(Nx, j) = f_3(Nx - 1, j) \quad (2.115a)$$

$$f_6(Nx, j) = f_6(Nx - 1, j) \quad (2.115b)$$

$$f_7(Nx, j) = f_7(Nx - 1, j) \quad (2.115c)$$

### Condición de contorno para modelo térmico

Han surgido dos casos para imponer condiciones de contorno para la temperatura. El primer caso es cuando se trata de una condición de contorno de tipo adiabático,

es decir, en el cual el flujo de calor debe ser nulo. En este caso se utiliza la condición de derivada nula para la función de distribución  $g$  (igual que en el apartado de condiciones de contorno para dominios abiertos). El segundo caso es cuando se trata de una condición de contorno fijo isotérmico, en la cual la temperatura se fija a un valor  $T_{dado}$  y la velocidad es nula. En este caso, se utiliza la función de equilibrio  $g^{(eq)}$  ( $T = T_{dado}, \mathbf{u} = \mathbf{0}$ ) para calcular las funciones de distribución en el contorno. [12] incluye una discusión más profunda sobre las condiciones de contorno del modelo térmico.

### 2.3.2. Evaluación de fuerzas

El cálculo de las fuerzas sobre un obstáculo es una de las finalidades más importantes que se persiguen cuando se hace una simulación en el ámbito de CFD. Por esta razón, se dedica esta sección a hablar exclusivamente sobre la evaluación de fuerzas en LBM. En el marco de este método de simulación hay básicamente dos maneras de evaluar la fuerza [17]. La primera de ellas se basa en la integración del tensor de tensiones sobre el contorno del obstáculo, mientras que la segunda manera se basa en el intercambio de momentum que sufren las partículas cuando chocan contra el contorno del obstáculo. A continuación se van a discutir los dos métodos introducidos anteriormente.

#### Integración del tensor de tensiones

La fuerza sobre un obstáculo de contorno  $\partial\Omega$  se evalúa de la siguiente manera:

$$\mathbf{F} = \int_{\partial\Omega} (-p\mathbf{I} + \boldsymbol{\tau}) \cdot \mathbf{n} dS \quad (2.116)$$

La presión  $p$  se calcula a partir de la densidad mediante la ecuación de estado  $p = c_s^2 \rho$  y el tensor de tensiones tangenciales se evalúa utilizando la parte de no equilibrio de la función de distribución.

$$\tau_{ij} = \left(1 - \frac{1}{2\tau}\right) \sum f_\alpha^{(neq)} \left( e_{\alpha,i} e_{\alpha,j} - \frac{1}{D} \mathbf{e}_\alpha \cdot \mathbf{e}_\alpha \delta_{ij} \right) \quad (2.117)$$

Se tendrá que extrapolar para obtener los valores adecuados de presión y tensor viscoso en el contorno del obstáculo. Este método es muy laborioso y por eso se utilizará el método de intercambio de momentum.

## Intercambio de momentum

Para utilizar este método se definen dos variables  $w(i, j)$  y  $w_b(i, j)$ .  $w(i, j)$  vale 0 si el nodo  $(i, j)$  pertenece al dominio fluido (wet node), y por el contrario, si vale 1 significa que se encuentra en el interior del sólido (dry node).  $w_b(i, j)$  vale 0 en todos los sitios excepto en los nodos que forman parte del contorno del obstáculo, en los cuales vale 1. Teniendo esto en cuenta, dado un nodo en el contorno que se encuentra en el interior del obstáculo caracterizado por la posición  $\mathbf{r}_b$  ( $w = 1$  y  $w_b = 1$ ), es posible el intercambio de momentum con los nodos vecinos de valor:

$$\sum_{\alpha \neq 0} \mathbf{e}_\alpha [f_\alpha(\mathbf{r}_b, t) + f_{\bar{\alpha}}(\mathbf{r}_b + \mathbf{e}_{\bar{\alpha}}\delta t, t)] [1 - w(\mathbf{r} + \mathbf{e}_{\bar{\alpha}}\delta t)] \quad (2.118)$$

La fuerza total será la suma de las contribuciones individuales de cada nodo.

$$\mathbf{F} = \sum_{\text{todos } \mathbf{r}_b} \sum_{\alpha \neq 0} \mathbf{e}_\alpha [f_\alpha(\mathbf{r}_b, t) + f_{\bar{\alpha}}(\mathbf{r}_b + \mathbf{e}_{\bar{\alpha}}\delta t, t)] [1 - w(\mathbf{r} + \mathbf{e}_{\bar{\alpha}}\delta t)] \quad (2.119)$$

siendo  $\mathbf{e}_{\bar{\alpha}}$  la dirección opuesta de  $\mathbf{e}_\alpha$ . Un detalle importante es el hecho de que la fuerza se debe evaluar después de la etapa de colisión.

### 2.3.3. Adimensionalización

En las simulaciones LBM se utilizan unidades de lattice, es decir, al tener  $\Delta x = 1$ ,  $\Delta y = 1$  y  $\Delta t = 1$ , el tiempo y la posición serán números enteros. Para obtener las variables adimensionalizadas se utilizan magnitudes características que también están en unidades de lattice. En la siguiente tabla se resumen las principales magnitudes características mediante las cuales se deben adimensionalizar los resultados obtenidos mediante LBM:

	Convección forzada	Convección natural
Distancia	$N$	$N$
Velocidad	$U_0$	$\frac{a}{N}$
Tiempo	$\frac{N}{U_0}$	$\frac{N^2}{a}$

Tabla 2.3: Magnitudes características en unidades lattice.

### 2.3.4. Aspectos de implementación paralela

Este apartado debería ser uno de los más importantes y más extensos ya que la implementación paralela es uno de los puntos más fuertes de LBM. La importancia

radica en que el proceso de colisión es totalmente local (se refiere únicamente al nodo y no a sus vecinos) y que el proceso de propagación es uniforme y no requiere mucho esfuerzo computacional. Sin embargo, debido al tiempo limitado, no se han hecho estudios de implementación paralela.

## Capítulo 3

# Finite Volume Method (FVM)

El método de volúmenes finitos divide el dominio espacial en un número finito de pequeños volúmenes de control que se definen mediante el proceso de mallado. Una vez definidos los volúmenes de control, se integran las ecuaciones gobernantes (diferenciales) sobre cada volumen de control y se obtienen, en consecuencia, las ecuaciones algebraicas que representan las leyes de conservación de masa, momento, etc. sobre cada volumen de control.

Las ecuaciones gobernantes adimensionalizadas para el caso de problemas de convección forzada son las siguientes:

$$\nabla \cdot \mathbf{u} = 0 \quad (3.1a)$$

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot (\mathbf{u}\mathbf{u}) = -\nabla p + \frac{1}{Re} \Delta \mathbf{u} \quad (3.1b)$$

$$\frac{\partial T}{\partial t} + \nabla \cdot (\mathbf{u}T) = \frac{1}{Pe} \Delta T \quad (3.1c)$$

en las que se ha hecho un cambio de notación respecto a las ecuaciones presentadas en el apartado de adimensionalización omitiendo el símbolo de asterisco para indicar la variable adimensionalizada. Se agrupan los términos convectivo y difusivo de las ecuaciones de momentum y de energía en  $\mathbf{R}(\mathbf{u})$  y  $Q(\mathbf{u}, T)$  definidas de la siguiente manera:

$$\mathbf{R}(\mathbf{u}) = \frac{1}{Re} \Delta \mathbf{u} - \nabla \cdot (\mathbf{u}\mathbf{u}) \quad (3.2a)$$

$$Q(\mathbf{u}, T) = \frac{1}{Pe} \Delta T - \nabla \cdot (\mathbf{u}T) \quad (3.2b)$$

Antes de entrar en la metodología de resolución de las ecuaciones de Navier-Stokes, se debe entender mejor el papel que juega el gradiente de presión en la ecuación de

momentum utilizando el teorema de Helmholtz-Hodge.

El teorema de Helmholtz-Hodge afirma que dado un campo vectorial  $\omega$  definido en un dominio acotado  $\Omega$  con el contorno liso  $\partial\Omega$ , puede ser descompuesto en la suma de dos campos vectoriales  $a$  y  $b$  de manera que uno de ellos tiene divergencia nula y el otro rotacional nulo dentro del dominio acotado  $\Omega$ . En el contorno  $\partial\Omega$  se debe cumplir la condición de  $a \cdot n = 0$ .

$$\omega = a + b \quad (3.3)$$

donde

$$\nabla \cdot a = 0 \quad a \in \Omega \quad (3.4a)$$

$$\nabla \times b = 0 \quad b \in \Omega \quad (3.4b)$$

$$a \cdot n = 0 \quad a \in \partial\Omega \quad (3.4c)$$

Para demostrar el teorema basta con demostrar que los campos vectoriales  $a$  y  $b$  son ortogonales entre ellos y por tanto podrán construir cualquier otro campo vectorial. En primer lugar, es evidente que si el rotacional del campo vectorial  $b$  es nulo quiere decir que proviene del gradiente de un campo escalar y en consecuencia  $b = \nabla\varphi$ . A continuación se integra  $\nabla \cdot (a\varphi)$  en el dominio  $\Omega$ :

$$\int_{\Omega} \nabla \cdot (a\varphi) d\Omega = \int_{\Omega} \varphi \nabla \cdot a d\Omega + \int_{\Omega} (\nabla\varphi) \cdot a d\Omega \quad (3.5)$$

Teniendo en cuenta la Ecuación (3.4a) y aplicando el teorema de divergencia al término de la izquierda de la Ecuación (3.5), se obtiene:

$$\int_{\partial\Omega} (a\varphi) \cdot n dS = \int_{\Omega} (\nabla\varphi) \cdot a d\Omega \quad (3.6)$$

Por último teniendo en cuenta la Ecuación (3.4c) se obtiene:

$$\int_{\Omega} (\nabla\varphi) \cdot a d\Omega = 0 \quad (3.7)$$

que quiere decir que el campo vectorial  $\nabla\varphi$  es perpendicular al campo vectorial  $a$ . Hace falta notar que si a la función  $\varphi$  se le suma un valor constante, su gradiente queda inalterado y, por tanto, existen infinitos campos escalares  $\varphi$  que cumplen el teorema.

A continuación se aplica el teorema de Helmholtz-Hodge a la Ecuación (3.1b) particularizando  $a = \frac{\partial u}{\partial t}$ ,  $b = \nabla p$  y  $\omega = R(u)$ . Es fácil comprobar que las variables

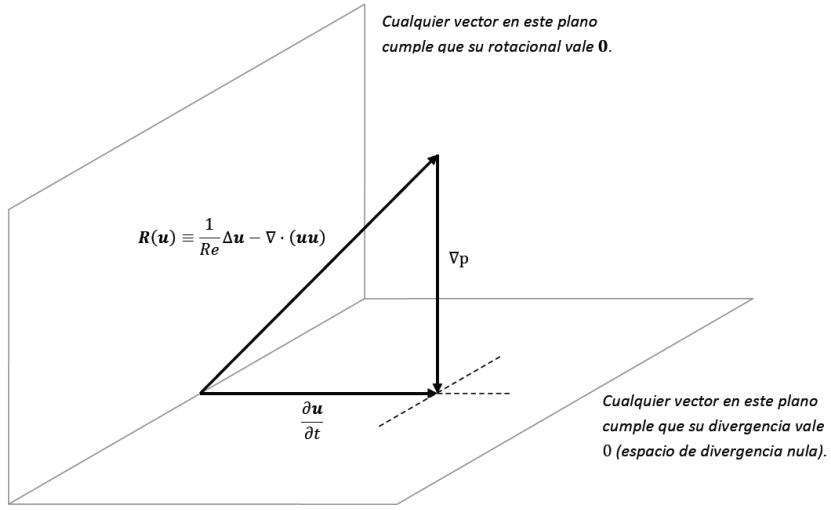


Figura 3.1: Representación de cómo se proyecta el término convectivo/difusivo mediante el gradiente de presiones y se obtiene la derivada temporal de la velocidad.

escogidas cumplen las Ecuaciones (3.4) y por tanto satisfacen el teorema. En la Figura (3.1) se representa la configuración espacial de los campos vectoriales.

Se puede ver que si se resuelve la ecuación de momentum sin considerar el término de gradiente de presión, la solución obtenida no cumplirá la condición de incompresibilidad (divergencia nula). Por tanto, como se ve en la Figura (3.1), el gradiente de presión es el encargado de proyectar el término convectivo/difusivo de la ecuación de momentum y, en consecuencia, forzar que se cumpla la condición de incompresibilidad impuesta por la Ecuación (3.1a).

El método de Proyección o Fractional Step Method se basa en el teorema de Helmholtz Hodge y tiene la ventaja que desacopla el cálculo de la presión del cálculo de la velocidad. El algoritmo se basa en dos etapas consecutivas. En primer lugar, se calcula una velocidad intermedia (llamada velocidad predictora), la cual no satisface la condición de incompresibilidad. En segundo y último lugar, se utiliza la pseudopresión (que se definirá más adelante) para proyectar la velocidad intermedia en un espacio de divergencia nula y por tanto hacer que se cumpla la condición de incompresibilidad.

Se discretizan las Ecuaciones (3.1) en el dominio temporal utilizando el método

explícito de segundo orden de Adams-Bashforth <sup>1</sup>:

$$\nabla \cdot \mathbf{u}^{(n+1)} = 0 \quad (3.8a)$$

$$\frac{\mathbf{u}^{(n+1)} - \mathbf{u}^{(n)}}{\Delta t^{(n)}} = \frac{3}{2} \mathbf{R}(\mathbf{u}^{(n)}) - \frac{1}{2} \mathbf{R}(\mathbf{u}^{(n-1)}) - \nabla p^{(n+1)} \quad (3.8b)$$

$$\frac{T^{(n+1)} - T^{(n)}}{\Delta t^{(n)}} = \frac{3}{2} Q(\mathbf{u}^{(n)}, T^{(n)}) - \frac{1}{2} Q(\mathbf{u}^{(n-1)}, T^{(n-1)}) \quad (3.8c)$$

Se define la velocidad predictora  $\mathbf{u}_p$  como aquella velocidad que se obtendría al resolver  $\mathbf{u}^{(n+1)}$  de la Ecuación (3.1b) sin tener en cuenta el gradiente de presión.

$$\frac{\mathbf{u}_p - \mathbf{u}^{(n)}}{\Delta t^{(n)}} = \frac{3}{2} \mathbf{R}(\mathbf{u}^{(n)}) - \frac{1}{2} \mathbf{R}(\mathbf{u}^{(n-1)}) \quad (3.9)$$

Despejando de la ecuación anterior la velocidad predictora se obtiene:

$$\mathbf{u}_p = \mathbf{u}^{(n)} + \Delta t^{(n)} \left( \frac{3}{2} \mathbf{R}(\mathbf{u}^{(n)}) - \frac{1}{2} \mathbf{R}(\mathbf{u}^{(n-1)}) \right) \quad (3.10)$$

La relación que se establece entre el campo de velocidad en el instante de tiempo  $n + 1$  y la velocidad predictora es la siguiente:

$$\mathbf{u}^{(n+1)} = \mathbf{u}_p - \nabla \tilde{p}^{(n+1)} \quad (3.11)$$

siendo  $\tilde{p}$  la pseudopresión, definida como  $\tilde{p}^{(n+1)} = \Delta t^{(n)} p^{(n+1)}$ . Se puede aplicar otra vez el teorema de Helmholtz Hodge a la Ecuación (3.11) particularizando  $\mathbf{a} = \mathbf{u}^{(n+1)}$ ,  $\mathbf{b} = \nabla \tilde{p}^{(n+1)}$  y  $\boldsymbol{\omega} = \mathbf{u}_p$ .

A continuación, se evalúa la divergencia de la Ecuación (3.11) y, teniendo en cuenta que el campo de velocidad en el instante de tiempo  $n+1$  debe ser incompresible como se impone en la Ecuación (3.8a), se obtiene la Ecuación de Poisson:

$$\Delta \tilde{p}^{(n+1)} = \nabla \cdot \mathbf{u}_p \quad (3.12)$$

Para resolver la Ecuación (3.12) se impone como condición de contorno derivada nula de presión en el contorno.

---

<sup>1</sup>Se utiliza la siguiente notación:

$$\phi^{(n)} = \phi \left( t^{(0)} + \sum_{n-1 \geq 0} \Delta t^{(n)} \right)$$

siendo  $n$  un número entero que cuantifica la cantidad de incrementos de tiempos calculados en el tiempo,  $\phi$  una función arbitraria que depende del tiempo,  $\Delta t^{(n)}$  el incremento de tiempo empleado para el instante  $n$  y  $t^{(0)}$  el tiempo inicial de simulación.

A continuación, se procede a detallar el proceso de mallado, el cual sirve para definir las caras de los volúmenes de control. Para que el desarrollo no sea muy complicado, se utilizará malla cartesiana rectangular no uniforme y un dominio espacial rectangular de dimensiones  $L_x \times L_y$ . Se divide el eje  $x$  e  $y$  en  $N_x + 1$  y  $N_y + 1$  caras de manera que habrán  $N_x \times N_y$  volúmenes de control internos. Las posiciones de cada cara, que van de 0 a  $N_x$  o  $N_y$ , vienen impuestas por dos funciones,  $x_i$  e  $y_j$ . Para el caso de una malla cartesiana rectangular uniforme las expresiones para  $x_i$  e  $y_j$  son las siguientes:

$$x_i = \frac{L_x}{N_x} i = 0, 1, \dots, N_x \quad (3.13a)$$

$$y_j = \frac{L_y}{N_y} j = 0, 1, \dots, N_y \quad (3.13b)$$

En la Figura (3.2) se representa un ejemplo de una malla cartesiana rectangular uniforme. Normalmente en las paredes se coloca una malla más fina para poder

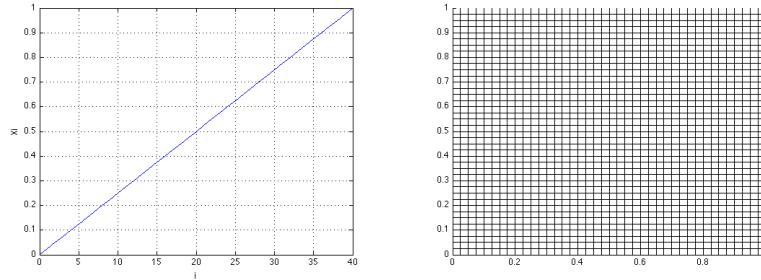


Figura 3.2: Representación de una malla cartesiana rectangular uniforme de  $N_x = 40$ ,  $N_y = 40$ ,  $L_x = 1$  y  $L_y = 1$ .

aproximar mejor la solución. En estos casos se utilizan mallas no uniformes. Una manera fácil de generar una malla cartesiana rectangular no uniforme es utilizar la función de tangente hiperbólica para las funciones  $x_i$  e  $y_j$ :

$$d_k = A \tanh \left( C \left( k - \frac{N}{2} \right) \right) + \frac{L}{2} \quad k = 0, 1, \dots, N \quad (3.14)$$

siendo  $A = -\frac{L/2}{\tanh(-CN/2)}$  y  $C$  un parámetro de ajuste. En la Figura (3.3) se representa un ejemplo de una malla cartesiana rectangular no uniforme. La generación de la malla no se refiere únicamente a la definición de las caras sino que se refiere también al cálculo de parámetros geométricos, que son las distancias entre los centroides de los volúmenes de control vecinos y los tamaños de los volúmenes de control. A

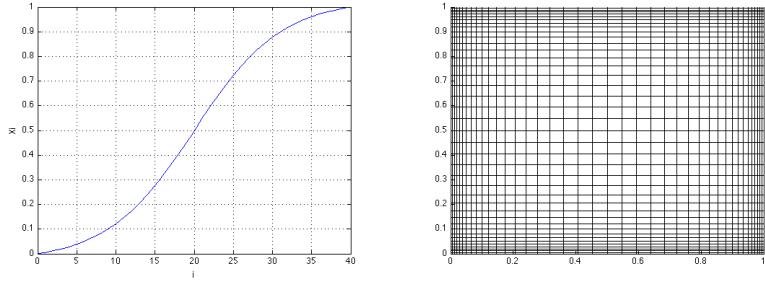


Figura 3.3: Representación de una malla rectangular no uniforme de  $N_x = 40$ ,  $N_y = 40$ ,  $L_x = 1$ ,  $L_y = 1$  y  $C = 0,09$ .

continuación se presentan las expresiones que sirven para calcular los parámetros geométricos de la malla representada en la Figura (3.4).

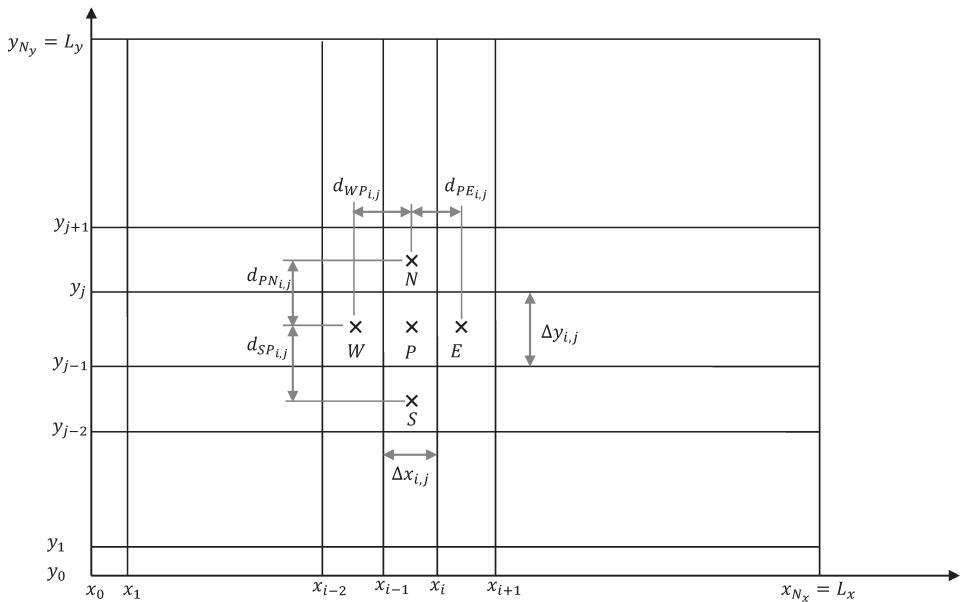


Figura 3.4: Representación de una malla genérica.

$$\begin{aligned}
 \Delta x_{W_{i,j}} &= x_i - x_{i-1} & d_{WP_{i,j}} &= \frac{1}{2} (\Delta x_{W_{i,j}} + \Delta x_{i,j}) \\
 \Delta x_{E_{i,j}} &= x_{i+2} - x_{i+1} & d_{PE_{i,j}} &= \frac{1}{2} (\Delta x_{E_{i,j}} + \Delta x_{i,j}) \\
 \Delta y_{N_{i,j}} &= y_{j+2} - y_{j+1} & d_{SP_{i,j}} &= \frac{1}{2} (\Delta y_{S_{i,j}} + \Delta y_{i,j}) \\
 \Delta y_{S_{i,j}} &= y_j - y_{j-1} & d_{PN_{i,j}} &= \frac{1}{2} (\Delta y_{N_{i,j}} + \Delta y_{i,j}) \\
 \Delta x_{i,j} &= x_{i+1} - x_i \\
 \Delta y_{i,j} &= y_{j+1} - y_j
 \end{aligned} \tag{3.15}$$

Una vez generada la malla se debe especificar dónde colocar las incógnitas (campo

de velocidad, presión, temperatura, etc.) sobre la malla. Hay dos maneras de hacerlo. La primera de ellas es el *Colocated Arrangement*, que coloca todas las variables en los centroides de los volúmenes de control y de esta manera utiliza los mismos volúmenes de control para integrar las ecuaciones gobernantes. La segunda manera es el *Staggered Arrangement*, que coloca la presión en los centroides de los volúmenes de control y las velocidades en las caras de los volúmenes de control y además perpendiculares a éstas. Para obtener más información sobre la ventaja/desventaja de ambas maneras de colocación de variables se puede consultar en [18].

De aquí en adelante se escoge el método *Staggered Arrangement*. En la Figura (3.5) se puede ver un ejemplo de colocación de las variables para el caso de tener  $N_x = 2$  y  $N_y = 2$ . A nivel de implementación, se agrupan las velocidades y presiones

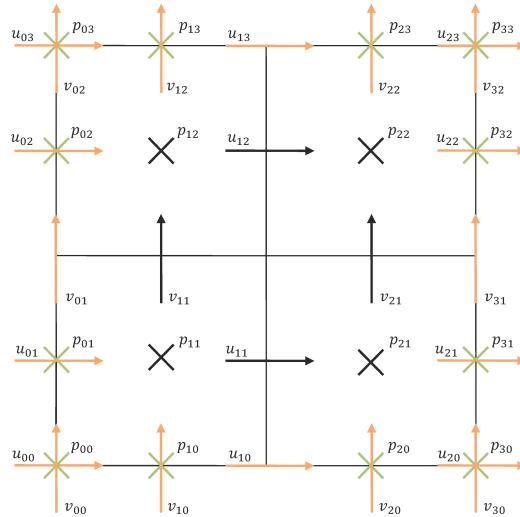


Figura 3.5: Representación de un ejemplo de colocación de las variables para el caso  $2 \times 2$ . La cruz indica la ubicación de las variables de presión y coincide con los centroides de los volúmenes de control para los nodos internos. La cruz negra indica que la presión está definida en el interior del dominio mientras que la de color verde indica que se encuentra en el contorno. Los vectores de velocidad de color naranja también están en el contorno y por tanto forman parte de las condiciones de contorno del problema.

discretas en tres matrices de diferentes dimensiones:

$$U = \begin{pmatrix} u_{0,0} & u_{1,0} & \cdots & u_{N_x,0} \\ u_{0,1} & u_{1,1} & \cdots & u_{N_x,1} \\ \vdots & \vdots & \ddots & \vdots \\ u_{0,N_y+1} & u_{1,N_y+1} & \cdots & u_{N_x,N_y+1} \end{pmatrix} \quad (3.16)$$

$$V = \begin{pmatrix} v_{0,0} & v_{1,0} & \cdots & v_{N_x+1,0} \\ v_{0,1} & v_{1,1} & \cdots & v_{N_x+1,1} \\ \vdots & \vdots & \ddots & \vdots \\ v_{0,N_y} & v_{1,N_y} & \cdots & v_{N_x+1,N_y} \end{pmatrix} \quad (3.17)$$

$$P = \begin{pmatrix} p_{1,1} & p_{2,1} & \cdots & p_{N_x,1} \\ p_{1,2} & p_{2,2} & \cdots & p_{N_x,2} \\ \vdots & \vdots & \ddots & \vdots \\ p_{1,N_y} & p_{2,N_y} & \cdots & p_{N_x,N_y} \end{pmatrix} \quad (3.18)$$

Las presiones de contorno no se incluyen en la matriz  $P$  ya que la condición de contorno de la Ecuación de Poisson (3.12) es de derivada de presión nula en la dirección normal al contorno y por tanto, los valores de presión en el contorno no intervienen en la resolución del problema y en consecuencia no hay necesidad de guardarlos.

A continuación se va a desarrollar *la función proyectora* (ver la Figura (3.6)). Dado un campo de velocidad compresible  $\mathbf{u}_c$  (que a nivel de implementación serán dos matrices  $U_c$  y  $V_c$ ), los parámetros del solver y la geometría de la malla, devuelve un campo de velocidad incompresible  $\mathbf{u}_i$  (que a nivel de implementación serán dos matrices  $U_i$  y  $V_i$ ) y también la pseudopresión  $\tilde{p}$  que la hace posible (que a nivel de implementación será una matriz  $\tilde{P}$ ). El campo de entrada de velocidad compresible será la velocidad predictora  $\mathbf{u}_p$ , mientras que el campo de salida de velocidad incompresible será la velocidad  $\mathbf{u}^{(n+1)}$  en el siguiente instante del tiempo.

Para resolver la Ecuación (3.12) se aplica el método de volúmenes finitos a la Ecuación de Poisson. El método consiste en la integración de la Ecuación (3.12) sobre el volumen de control representado en la Figura (3.7).

$$\int_{\Omega} \nabla \cdot \nabla \tilde{p} d\Omega = \int_{\Omega} \nabla \cdot \mathbf{u}_p d\Omega \quad (3.19)$$

A continuación se aplica el teorema de divergencia a los dos términos de la Ecuación (3.19):

$$\int_{\partial\Omega} \nabla \tilde{p} \cdot \mathbf{n} dS = \int_{\partial\Omega} \mathbf{u}_p \cdot \mathbf{n} dS \quad (3.20)$$

Seguidamente se dividen las caras del volumen de control  $\partial\Omega$  en cuatro partes correspondientes a la cara este ( $e$ ), oeste ( $w$ ), norte ( $n$ ) y sur ( $s$ ), y la Ecuación (3.20)

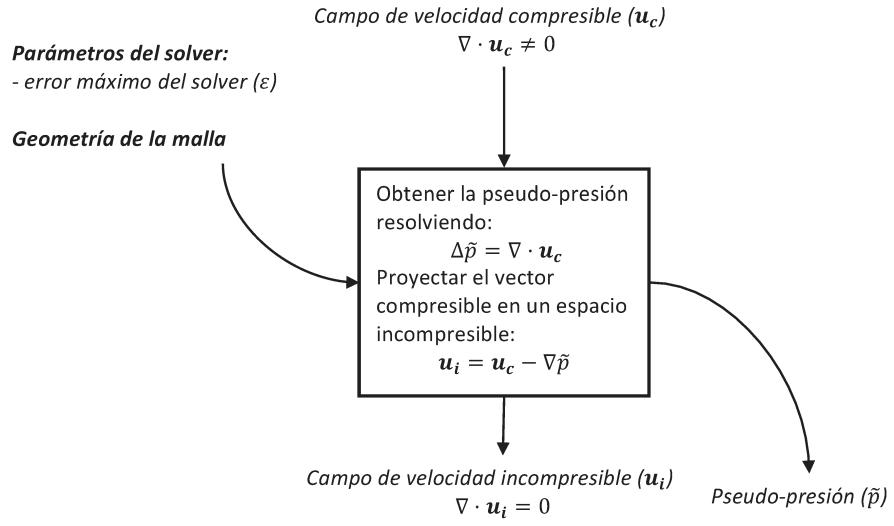


Figura 3.6: Representación esquemática de la función que, dado un campo de velocidad compresible, proyecta en un espacio de velocidades incompresibles.

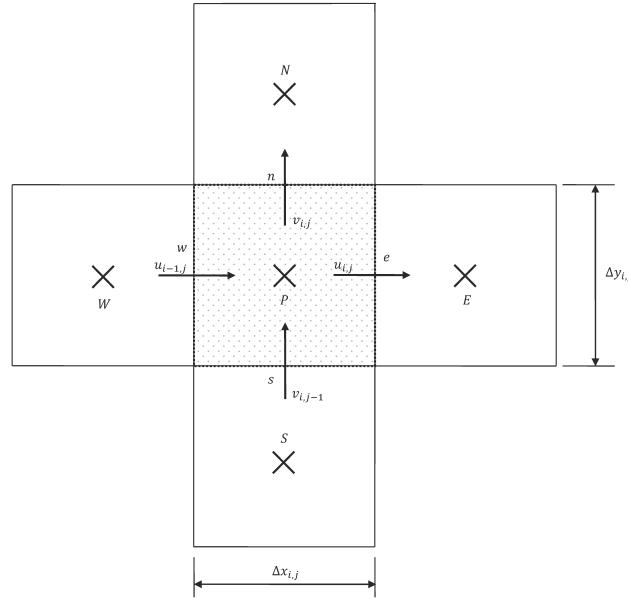


Figura 3.7: Representación del volumen de control (punteado) para la evaluación de la ecuación de continuidad.

se convierte en:

$$\begin{aligned}
 & \int_e \frac{\partial \tilde{p}}{\partial x} dS - \int_w \frac{\partial \tilde{p}}{\partial x} dS + \int_n \frac{\partial \tilde{p}}{\partial y} dS - \int_s \frac{\partial \tilde{p}}{\partial y} dS = \\
 &= \int_e u_p dS - \int_w u_p dS + \int_n v_p dS - \int_s v_p dS
 \end{aligned} \tag{3.21}$$

Por un lado, se supone que la derivada de la pseudopresión en las caras es constante y se aproxima mediante diferencias finitas y, por otro lado, se supone que las velocidades en las caras son constantes y se pueden extraer fuera de la integral obteniendo:

$$\begin{aligned} \frac{\tilde{p}_E - \tilde{p}_P}{d_{PE_{i,j}}} \Delta y_{i,j} - \frac{\tilde{p}_P - \tilde{p}_W}{d_{WP_{i,j}}} \Delta y_{i,j} + \frac{\tilde{p}_N - \tilde{p}_P}{d_{PN_{i,j}}} \Delta x_{i,j} - \frac{\tilde{p}_P - \tilde{p}_S}{d_{SP_{i,j}}} \Delta y_{i,j} = \\ = u_{p_{i,j}} \Delta y_{i,j} - u_{p_{i-1,j}} \Delta y_{i,j} + v_{p_{i,j}} \Delta x_{i,j} - v_{p_{i,j-1}} \Delta x_{i,j} \end{aligned} \quad (3.22)$$

siendo  $\tilde{p}_P = \tilde{p}_{i,j}$ ,  $\tilde{p}_E = \tilde{p}_{i+1,j}$ ,  $\tilde{p}_W = \tilde{p}_{i-1,j}$ ,  $\tilde{p}_N = \tilde{p}_{i,j+1}$  y  $\tilde{p}_S = \tilde{p}_{i,j-1}$ . A continuación se reordena la Ecuación (3.22) y se obtiene:

$$a_P(i,j) \tilde{p}_P = a_E(i,j) \tilde{p}_E + a_E(i,j) \tilde{p}_E + a_W(i,j) \tilde{p}_W + a_N(i,j) \tilde{p}_N + a_S(i,j) \tilde{p}_S + b_P(i,j) \quad (3.23)$$

siendo:

$$a_E(i,j) = -\frac{\Delta y_{i,j}}{d_{PE_{i,j}}} \quad (3.24a)$$

$$a_W(i,j) = -\frac{\Delta y_{i,j}}{d_{WP_{i,j}}} \quad (3.24b)$$

$$a_N(i,j) = -\frac{\Delta x_{i,j}}{d_{PN_{i,j}}} \quad (3.24c)$$

$$a_S(i,j) = -\frac{\Delta x_{i,j}}{d_{SP_{i,j}}} \quad (3.24d)$$

$$a_P(i,j) = a_E(i,j) + a_W(i,j) + a_N(i,j) + a_S(i,j) \quad (3.24e)$$

$$b_P(i,j) = u_{p_{i,j}} \Delta y_{i,j} - u_{p_{i-1,j}} \Delta y_{i,j} + v_{p_{i,j}} \Delta x_{i,j} - v_{p_{i,j-1}} \Delta x_{i,j} \quad (3.24f)$$

Para resolver el sistema lineal  $A\tilde{p} = b$ , representado mediante la Ecuación (3.24), se utilizan métodos iterativos. En este trabajo se han implementado dos métodos iterativos. El primero de ellos es el método Gauss-Sheidel con factor de relajación, representado en la Figura (3.8).

El segundo método iterativo es el Conjugate Gradient [20], que se representa en la Figura (3.9). Para poder aplicar el método, la matriz  $A$  tiene que ser simétrica y definida positiva.

Hace falta especificar que el sistema lineal tiene determinante nulo ( $\det A = 0$ ) y por tanto es indeterminado y tiene infinitas soluciones. Para fijar una solución, lo que se hace es, o bien fijar la presión en un nodo interno del dominio (por ejemplo  $\tilde{p}_{1,1} = 1$ ), o bien multiplicar el coeficiente  $a_P(i,j)$  de un nodo interno por una constante (por ejemplo  $a_P(1,1) \rightarrow 1,1a_P(1,1)$ ). El primer método para fijar la

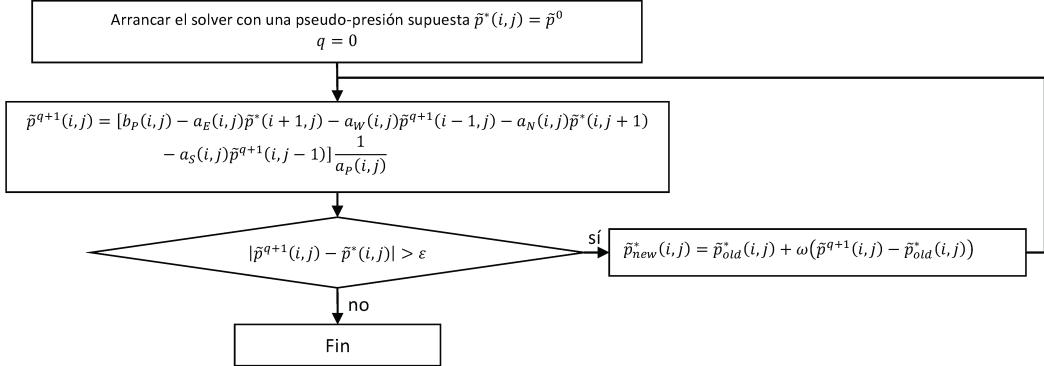


Figura 3.8: Esquema de implementación del solver Gauss-Sheidel

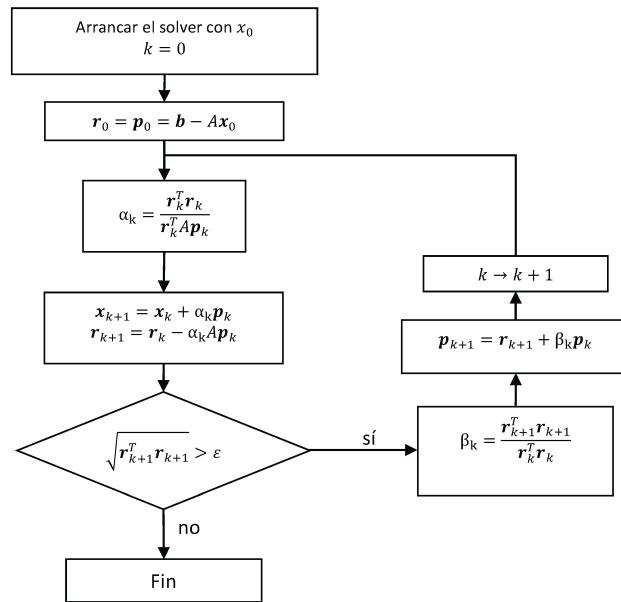


Figura 3.9: Esquema de implementación del solver Conjugate Gradient

presión hace que la matriz  $A$  deje de ser simétrica, y por tanto hace que no se pueda aplicar el método Conjugate Gradient.

Una vez calculado el campo  $\tilde{p}$ , se proyecta la velocidad predictora mediante la Ecuación (3.11):

$$u_{i,j}^{(n+1)} = u_{p_{i,j}} - \frac{\tilde{p}_E - \tilde{p}_P}{d_{PE_{i,j}}} \quad (3.25a)$$

$$v_{i,j}^{(n+1)} = v_{p_{i,j}} - \frac{\tilde{p}_N - \tilde{p}_P}{d_{PN_{i,j}}} \quad (3.25b)$$

Llegados a este punto queda pendiente el método de cálculo de la velocidad pre-

dictora. Para hacerlo se definen dos volúmenes de control (ver Figuras (3.10 - 3.11)) según la dirección de la componente de la velocidad, y se integra la Ecuación (3.10) sobre los respectivos volúmenes de control.

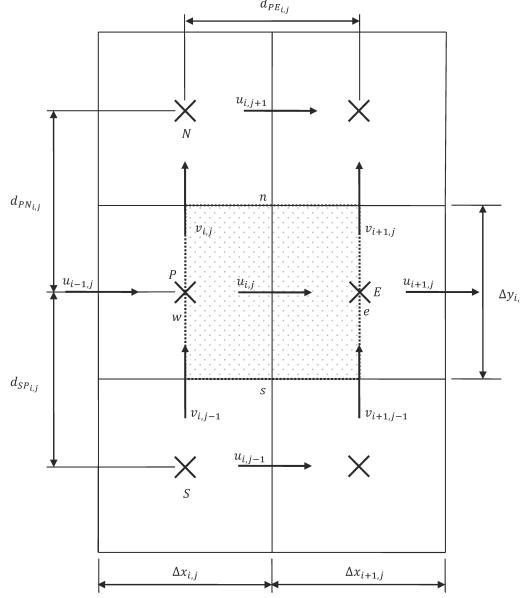


Figura 3.10: Representación del volumen de control (punteado) para la evaluación del componente  $x$  de la ecuación de momentum.

$$\int_{\Omega} \mathbf{u}_p d\Omega = \int_{\Omega} \mathbf{u}^{(n)} d\Omega + \Delta t^{(n)} \left( \frac{3}{2} \int_{\Omega} \mathbf{R}(\mathbf{u}^{(n)}) d\Omega - \frac{1}{2} \int_{\Omega} \mathbf{R}(\mathbf{u}^{(n-1)}) d\Omega \right) \quad (3.26)$$

Particularizando para las direcciones  $x$  e  $y$  se obtiene:

$$u_{p_{i,j}} = u_{i,j}^{(n)} + \frac{\Delta t^{(n)}}{\Delta y_{i,j} d_{PE_{i,j}}} \left( \frac{3}{2} \int_{\Omega} R_x(\mathbf{u}^{(n)}) d\Omega - \frac{1}{2} \int_{\Omega} R_x(\mathbf{u}^{(n-1)}) d\Omega \right) \quad (3.27a)$$

$$v_{p_{i,j}} = v_{i,j}^{(n)} + \frac{\Delta t^{(n)}}{\Delta x_{i,j} d_{PN_{i,j}}} \left( \frac{3}{2} \int_{\Omega} R_y(\mathbf{u}^{(n)}) d\Omega - \frac{1}{2} \int_{\Omega} R_y(\mathbf{u}^{(n-1)}) d\Omega \right) \quad (3.27b)$$

La integral de  $R_x$  se evalúa utilizando otra vez el teorema de divergencia:

$$\begin{aligned} \int_{\Omega} R_x(\mathbf{u}) d\Omega &= - \int_{\Omega} \nabla \cdot (\mathbf{u}\mathbf{u}) d\Omega + \frac{1}{Re} \int_{\Omega} \nabla \cdot \nabla u d\Omega = - \int_{\partial\Omega} (\mathbf{u}\mathbf{u}) \cdot \mathbf{n} dS + \\ &\quad + \frac{1}{Re} \int_{\partial\Omega} \nabla u \cdot \mathbf{n} dS = \\ &= \frac{1}{Re} \left( - \frac{u_{i,j} - u_{i-1,j}}{\Delta x_{i,j}} \Delta y_{i,j} + \frac{u_{i+1,j} - u_{i,j}}{\Delta x_{i+1,j}} \Delta y_{i,j} + \frac{u_{i,j+1} - u_{i,j}}{d_{PN_{i,j}}} d_{PE_{i,j}} - \right. \\ &\quad \left. - \frac{u_{i,j} - u_{i,j-1}}{d_{SP_{i,j}}} d_{PE_{i,j}} \right) - (u_{e_{i,j}} F_{e_{i,j}} + u_{n_{i,j}} F_{n_{i,j}} - u_{w_{i,j}} F_{w_{i,j}} - u_{s_{i,j}} F_{s_{i,j}}) \quad (3.28) \end{aligned}$$

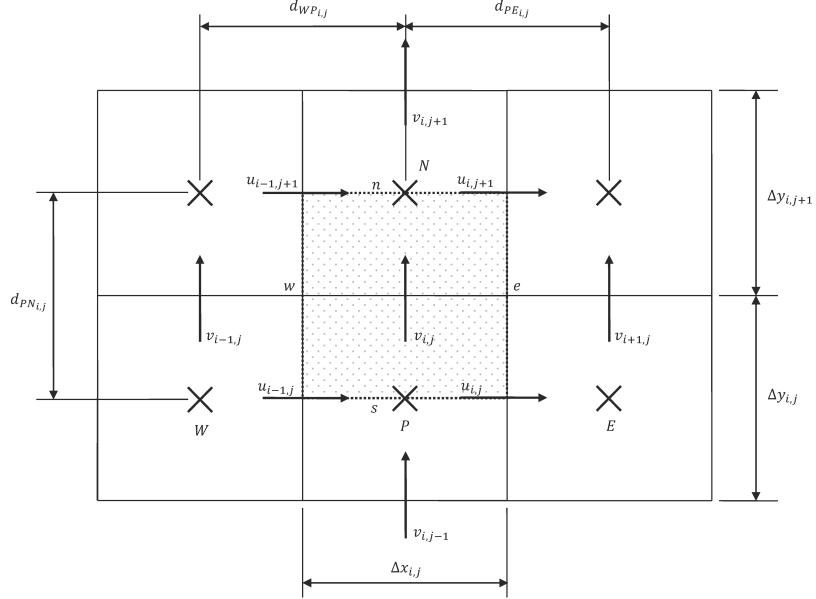


Figura 3.11: Representación del volumen de control (punteado) para la evaluación del componente  $y$  de la ecuación de momentum.

siendo:

$$u_{e_{i,j}} = \frac{1}{2} (u_{i+1,j} + u_{i,j}) \quad (3.29a)$$

$$u_{w_{i,j}} = \frac{1}{2} (u_{i-1,j} + u_{i,j}) \quad (3.29b)$$

$$u_{n_{i,j}} = \frac{1}{2} (u_{i,j+1} + u_{i,j}) \quad (3.29c)$$

$$u_{s_{i,j}} = \frac{1}{2} (u_{i,j-1} + u_{i,j}) \quad (3.29d)$$

$$F_{e_{i,j}} = \frac{1}{2} (u_{i,j} \Delta y_{i,j} + u_{i+1,j} \Delta y_{i,j}) \quad (3.29e)$$

$$F_{w_{i,j}} = \frac{1}{2} (u_{i,j} \Delta y_{i,j} + u_{i-1,j} \Delta y_{i,j}) \quad (3.29f)$$

$$F_{n_{i,j}} = \frac{1}{2} (v_{i,j} \Delta x_{i,j} + v_{i+1,j} \Delta x_{i+1,j}) \quad (3.29g)$$

$$F_{s_{i,j}} = \frac{1}{2} (v_{i,j-1} \Delta x_{i,j} + v_{i+1,j-1} \Delta x_{i+1,j}) \quad (3.29h)$$

De manera análoga, se obtiene para  $R_y$ :

$$\begin{aligned}
\int_{\Omega} R_y(\mathbf{u}) d\Omega &= - \int_{\Omega} \nabla \cdot (v\mathbf{u}) d\Omega + \frac{1}{\text{Re}} \int_{\Omega} \nabla \cdot \nabla v d\Omega = - \int_{\partial\Omega} (v\mathbf{u}) \cdot \mathbf{n} dS + \\
&\quad + \frac{1}{\text{Re}} \int_{\partial\Omega} \nabla v \cdot \mathbf{n} dS = \\
&= \frac{1}{\text{Re}} \left( - \frac{v_{i,j} - v_{i,j-1}}{\Delta y_{i,j}} \Delta x_{i,j} + \frac{v_{i,j+1} - v_{i,j}}{\Delta y_{i,j+1}} \Delta x_{i,j} + \frac{v_{i+1,j} - v_{i,j}}{d_{PE_{i,j}}} d_{PN_{i,j}} - \right. \\
&\quad \left. - \frac{v_{i,j} - v_{i-1,j}}{d_{WP_{i,j}}} d_{PN_{i,j}} \right) - (v_{e_{i,j}} F_{e_{i,j}} + v_{n_{i,j}} F_{n_{i,j}} - v_{w_{i,j}} F_{w_{i,j}} - v_{s_{i,j}} F_{s_{i,j}}) \quad (3.30)
\end{aligned}$$

siendo:

$$v_{e_{i,j}} = \frac{1}{2} (v_{i+1,j} + v_{i,j}) \quad (3.31a)$$

$$v_{w_{i,j}} = \frac{1}{2} (v_{i-1,j} + v_{i,j}) \quad (3.31b)$$

$$v_{n_{i,j}} = \frac{1}{2} (v_{i,j+1} + v_{i,j}) \quad (3.31c)$$

$$v_{s_{i,j}} = \frac{1}{2} (v_{i,j-1} + v_{i,j}) \quad (3.31d)$$

$$F_{e_{i,j}} = \frac{1}{2} (u_{i,j} \Delta y_{i,j} + u_{i,j+1} \Delta y_{i,j+1}) \quad (3.31e)$$

$$F_{w_{i,j}} = \frac{1}{2} (u_{i-1,j} \Delta y_{i,j} + u_{i-1,j+1} \Delta y_{i,j+1}) \quad (3.31f)$$

$$F_{n_{i,j}} = \frac{1}{2} (v_{i,j} \Delta x_{i,j} + v_{i,j+1} \Delta x_{i,j}) \quad (3.31g)$$

$$F_{s_{i,j}} = \frac{1}{2} (v_{i,j} \Delta x_{i,j} + v_{i,j-1} \Delta x_{i,j}) \quad (3.31h)$$

El esquema de resolución para problemas de convección forzada se representa en la Figura (3.12). Para problemas de convección natural las ecuaciones gobernantes cambian y son las siguientes:

$$\nabla \cdot \mathbf{u} = 0 \quad (3.32a)$$

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} = -\nabla p + \text{Pr} \Delta \mathbf{u} + \text{Ra} \cdot \text{Pr} T \mathbf{e}_y \quad (3.32b)$$

$$\frac{\partial T}{\partial t} + (\mathbf{u} \cdot \nabla) T = \Delta T \quad (3.32c)$$

y por tanto los términos convectivo y difusivo de las ecuaciones de momentum y de energía  $\mathbf{R}(\mathbf{u})$  y  $Q(\mathbf{u}, T)$  se redefinen de la siguiente manera:

$$\mathbf{R}(\mathbf{u}) = \text{Pr} \Delta \mathbf{u} + \text{Ra} \cdot \text{Pr} T \mathbf{e}_y - \nabla \cdot (\mathbf{u} \mathbf{u}) \quad (3.33a)$$

$$Q(\mathbf{u}, T) = \Delta T - \nabla \cdot (\mathbf{u} T) \quad (3.33b)$$

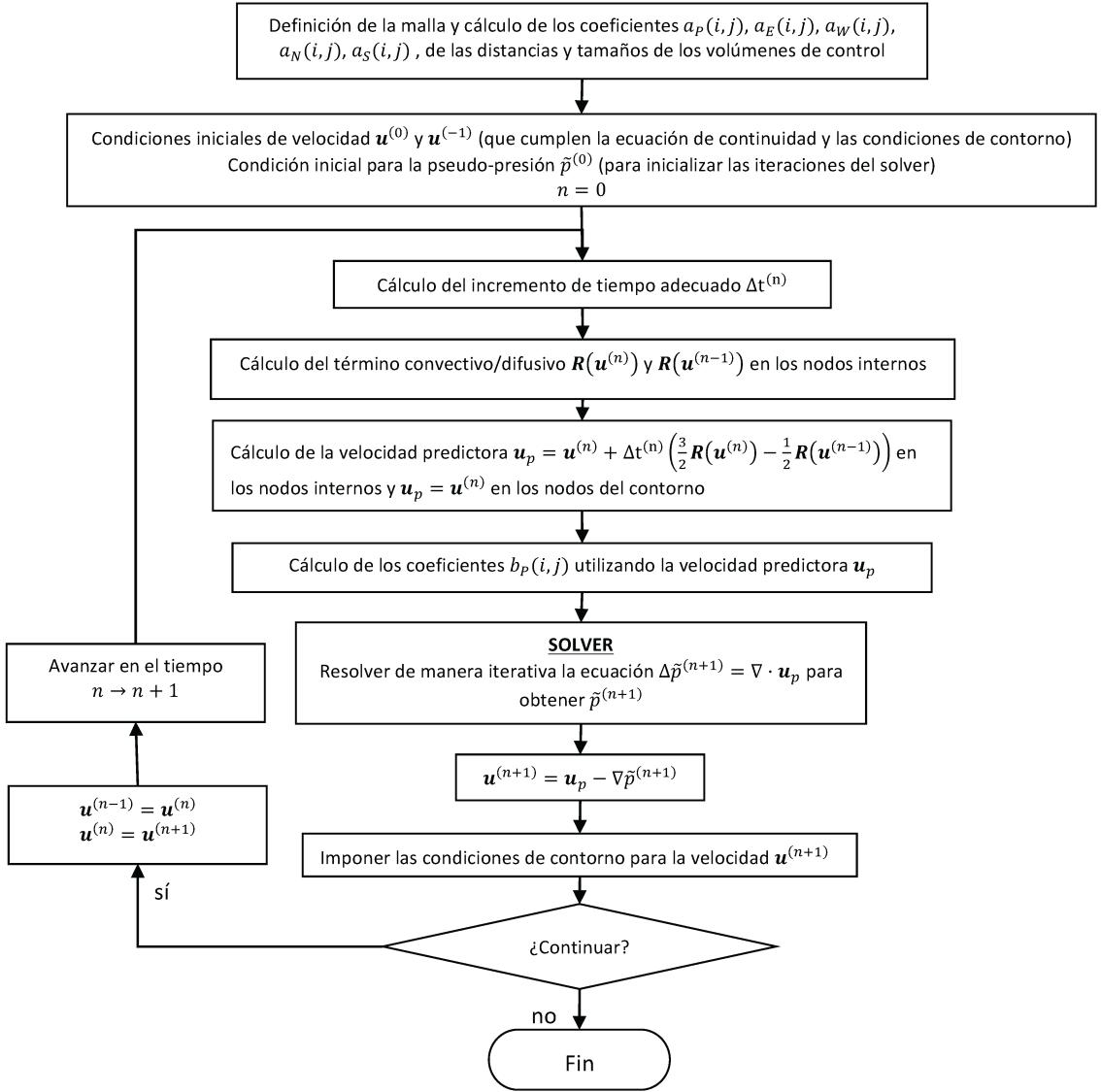


Figura 3.12: Representación del algoritmo de implementación del método de volúmenes finitos basado en Fractional Step Method para convección forzada.

El esquema de resolución para problemas de convección natural se representa en la Figura (3.13).

Para visualizar el flujo se utiliza el concepto de línea de corriente, que es el lugar geométrico de los puntos tangentes al vector velocidad para un cierto instante de tiempo. Para visualizar las líneas de corriente se trazan las curvas de nivel de la

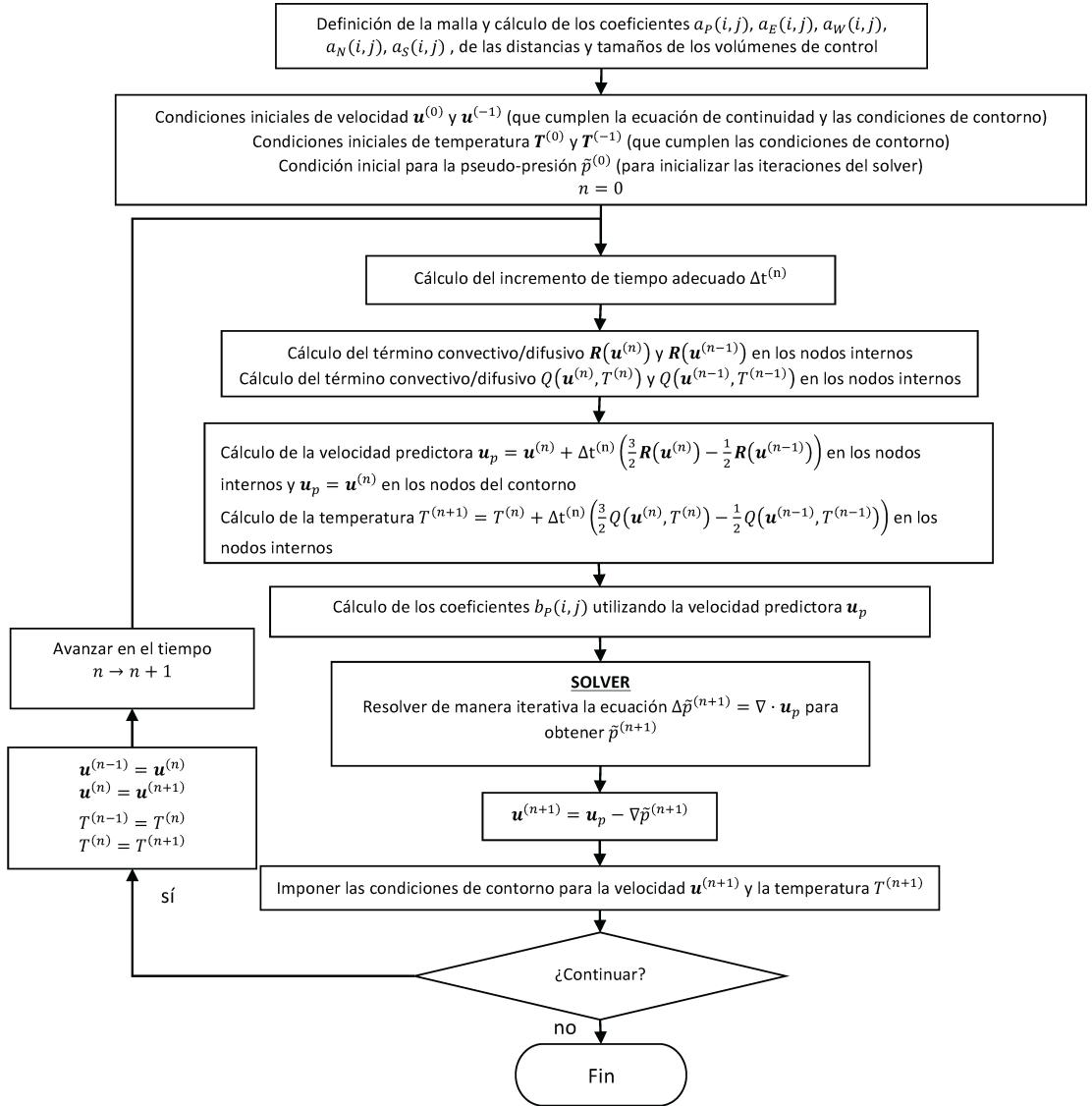


Figura 3.13: Representación del algoritmo de implementación del método de volúmenes finitos basado en Fractional Step Method para convección natural.

función de corriente  $\psi$ , definida de la siguiente manera:

$$u = -\frac{\partial \psi}{\partial y} \quad (3.34a)$$

$$v = \frac{\partial \psi}{\partial x} \quad (3.34b)$$

A partir de la definición anterior se puede llegar a la siguiente expresión:

$$\nabla^2 \psi = (\nabla \times \mathbf{u}) \cdot \mathbf{e}_z \quad (3.35)$$

siendo  $e_z$  el vector normal que apunta a la dirección  $z$  y que es perpendicular al campo de velocidad  $\mathbf{u}$ . Se puede utilizar el método de volúmenes finitos para integrar la Ecuación (3.35) sobre un volumen de control (en el cual el centroide es la intersección de las caras mediante las que se han discretizado el dominio de espacio) y obtener así un sistema lineal de estilo  $a_P\psi_P = a_E\psi_E + a_W\psi_W + a_N\psi_N + a_S\psi_S + b_P$ . Sin embargo, es más fácil utilizar la definición de la función de corriente  $\psi$  y, mediante diferencias finitas, obtener el valor de  $\psi$  para cada nodo. La condición de contorno es de un valor fijo para un nodo del contorno, por ejemplo,  $\psi_{0,0} = 0$ .

## Capítulo 4

# Casos prácticos

### 4.1. Lid Driven Cavity

El flujo laminar incompresible en una cavidad cuya pared superior se mueve con una velocidad uniforme, se ha utilizado multitud de veces para comprobar nuevos métodos de simulación de flujos a pesar de tener dos singularidades en las dos esquinas superiores (ver la Figura (4.1) ).

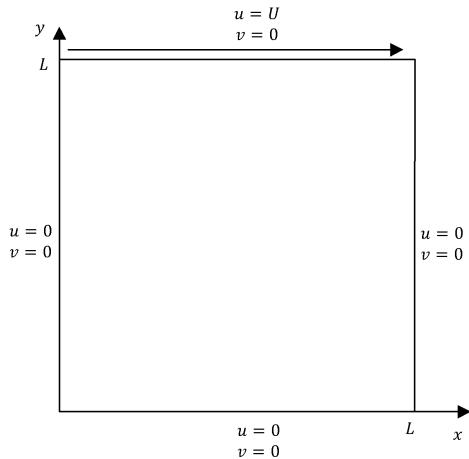


Figura 4.1: Representación esquemática del problema de Driven Cavity.

La velocidad en la cara superior,  $U$ , es la que genera el movimiento dentro de la cavidad formando un vórtice principal en el centro y otros más pequeños en las esquinas, dependiendo del número de Reynolds. Al tratarse de un problema de convección forzada, el número de Reynolds es el único parámetro adimensional que caracteriza la dinámica del problema (representado mediante el campo de velocidad adimensional).

Como solución de referencia se utilizará el trabajo de Ghia [19], el cual proporciona, entre otras cosas, los perfiles de velocidad horizontal en función de la coordenada  $y$  fijando  $x = 0,5$ , y los perfiles de velocidad vertical en función de la coordenada  $x$  fijando  $y = 0,5$ , para diferentes números de Reynolds.

Para cuantificar la desviación de la solución de referencia (error) se utilizará el módulo  $L_2$  del error relativo, que está definido de la siguiente manera para una variable genérica  $u$ :

$$e_u = \sqrt{\frac{\sum_{n=1}^q (u_n - u_n^{ref})^2}{\sum_{n=1}^q (u_n^{ref})^2}} \quad (4.1)$$

siendo  $u$  la variable calculada,  $u^{ref}$  la variable de referencia (el valor exacto) y  $q$  la cantidad de puntos que se van a comparar.

Para definir el estado estacionario se introduce el concepto de error estacionario definido de la siguiente manera:

$$\varepsilon_{est} = \min \left\{ \frac{u^{(n)} - u^{(n-1)}}{\Delta t^{(n)}}, \frac{v^{(n)} - v^{(n-1)}}{\Delta t^{(n)}} \right\} \quad (4.2)$$

Se utiliza  $\varepsilon_{est}$  como condición para saber cuando parar la simulación, y entonces se dice que se ha alcanzado el estado estacionario con un error  $\varepsilon_{est}$ .

Para caracterizar los resultados se han utilizado los siguientes indicadores: tiempo de computación total  $t_{comp}$  en segundos, tiempo de computación promedio por ciclo  $\bar{t}_{ciclo}$  en segundos, tiempo de simulación total  $t_{sim}$ , número de ciclos  $n$ , norma  $L_2$  del error relativo (%) de la componente horizontal de la velocidad  $e_u$  y norma  $L_2$  del error relativo (%) de la componente vertical de la velocidad  $e_v$ .

## FVM

Para estudiar el problema mediante el método de Volúmenes Finitos se establecen  $U = 1$  y  $L = 1$ . Los parámetros  $C_{conv}$  y  $C_{visc}$  son muy importantes ya que de ellos depende si la simulación converge o diverge. En [21] se recomienda utilizar  $C_{conv} = 0,35$  y  $C_{visc} = 0,2$ , sin embargo se ha comprobado que utilizando dichos valores la simulación diverge y en consecuencia se ha aplicado un factor de 0,4 a dichos valores resultando finalmente en  $C_{conv} = 0,14$  y  $C_{visc} = 0,08$ . Se ha escogido el solver Gauss-Sheidel con factor de relajación de 1,1. La simulación parte de una condición inicial de velocidad nula en los nodos interiores y, cumpliendo la condición

de contorno en los nodos de contorno (hay que tener en cuenta que el campo de velocidad inicial debe cumplir la condición de incompresibilidad).

Los otros principales parámetros de simulación son el tamaño de discretización  $N(Nx = Ny)$ , el error del solver  $\varepsilon_{sol}$  y el error estacionario  $\varepsilon_{est}$ .

Para estudiar el efecto del error del solver y del error estacionario se ha fijado el tamaño de discretización uniforme en  $N = 40$ , y se han probado varias combinaciones de los dos parámetros mencionados para el número de Reynolds de  $Re = 100$ .

$\varepsilon_{sol}$	$\varepsilon_{est}$	$\bar{t}_{ciclo}$	$t_{comp}$	$t_{sim}$	$n$	$e_u$ (%)	$e_v$ (%)
$10^{-1}$	$10^{-9}$	0,002633	19,647489	28,352834	7461	15,80969121	61,28635407
$10^{-2}$	$10^{-9}$	0,002624	19,575273	28,352834	7461	15,80969121	61,28635407
$10^{-3}$	$10^{-9}$	0,002626	17,166172	25,155441	6537	8,54051378	24,58180936
$10^{-4}$	$10^{-9}$	0,002626	18,543157	27,008739	7062	3,300697041	8,061219044
$10^{-5}$	$10^{-9}$	0,002644	19,466036	28,039319	7362	0,667783351	2,792519723
$10^{-6}$	$10^{-9}$	0,00272	21,70997	30,372852	7982	0,37264222	2,714343715
$10^{-7}$	$10^{-9}$	0,002477	25,139456	31,021903	8154	0,384320466	2,80524605
$10^{-1}$	$10^{-6}$	0,002631	11,982813	17,331499	4554	15,80969121	61,28635407
$10^{-2}$	$10^{-6}$	0,002635	11,997843	17,331499	4554	15,80969121	61,28635407
$10^{-3}$	$10^{-6}$	0,002643	11,324689	16,565296	4300	8,539982951	24,58180936
$10^{-4}$	$10^{-6}$	0,002633	12,151368	17,691901	4615	3,300697041	8,061219044
$10^{-5}$	$10^{-6}$	0,002655	12,997901	18,683995	4896	0,667783351	2,792519723
$10^{-6}$	$10^{-6}$	0,002766	15,152452	20,879791	5478	0,37264222	2,714343715
$10^{-7}$	$10^{-6}$	0,003299	18,588529	21,459131	5634	0,384320466	2,80524605
$10^{-1}$	$10^{-3}$	0,002631	4,443024	6,469348	1689	15,89037728	61,80086129
$10^{-2}$	$10^{-3}$	0,002643	4,463317	6,569348	1689	15,89037728	61,80086129
$10^{-3}$	$10^{-3}$	0,002631	5,300961	7,790762	2015	8,543698757	24,81451933
$10^{-4}$	$10^{-3}$	0,002471	5,725661	8,382573	2170	3,335731779	8,326653861
$10^{-5}$	$10^{-3}$	0,002683	6,379051	9,13132	2378	0,660882569	2,79979191
$10^{-6}$	$10^{-3}$	0,00294	7,055221	9,210349	2400	0,385382125	2,294374929
$10^{-7}$	$10^{-3}$	0,004185	10,114714	9,273949	2417	0,394937053	2,338008049

Tabla 4.1: Combinaciones del error del solver y del error estacionario para  $Re = 100$  y  $N = 40$ .

Observando los datos expuestos en la Tabla (4.1) y representados en la Figura (4.2), se llega a la conclusión de que el error del solver es un parámetro importante ya que de él depende la calidad de la solución obtenida. Cuanto más pequeño es el error del solver, menor es el error en los perfiles de la velocidad pero mayor es el tiempo de computación total. Por otro lado, se observa que el error estacionario no varía mucho la calidad de la solución, sin embargo sí que aumenta el tiempo computación considerablemente.

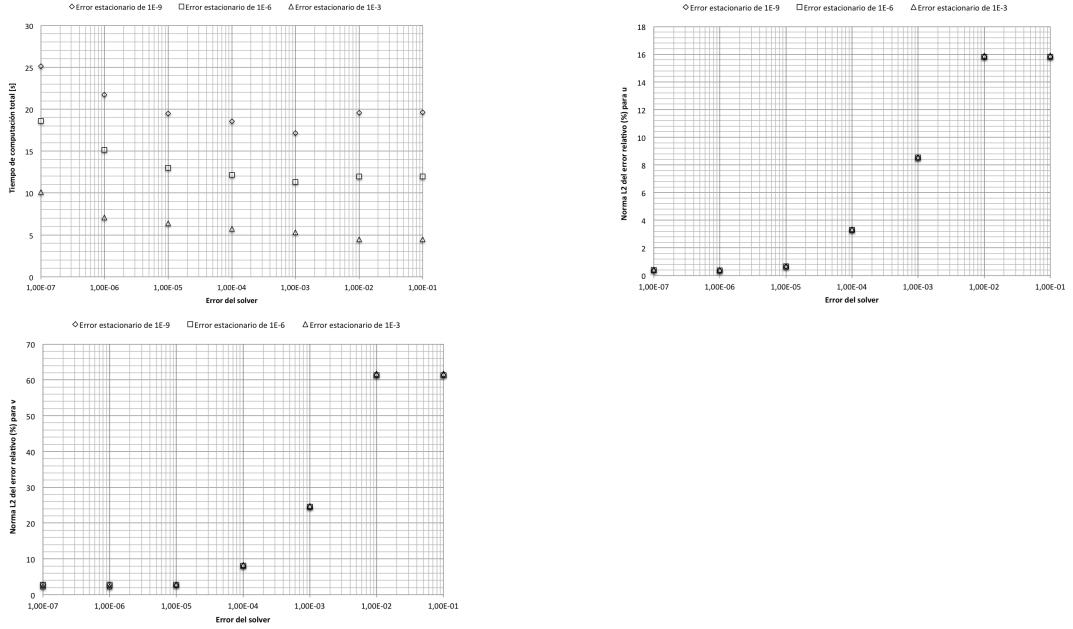


Figura 4.2: Tiempo de computación total, norma  $L_2$  del error relativo de la componente horizontal y vertical de la velocidad en función del error de solver y error en estacionario para  $N = 40$  y  $Re = 100$ .

## LBM

En el método de Lattice Boltzmann, al tener  $\Delta x = 1$ ,  $L = \Delta x N = N$  (siendo  $N$  la cantidad de nodos en cada cara), el número de Reynolds depende del tamaño de la discretización. Por otro lado, como se ha visto anteriormente, para mantener el número de Mach pequeño la velocidad más grande del problema debe ser pequeña, luego  $U$  no puede ser igual a 1. Para escoger la frecuencia de relajación adimensional, ésta debe ser inferior a 2 (en la práctica, a 1,8) para que la viscosidad cinemática no sea negativa (según la Ecuación (2.76)). En la Figura (4.3) se representa la frecuencia de relajación adimensional  $\omega = \frac{1}{3\frac{UN}{Re} + \frac{1}{2}}$  en función de la velocidad  $U$  y el tamaño de la discretización  $N$  para  $Re = 100$  y  $Re = 1000$ , respectivamente.

Se puede ver que a mayor número de Reynolds el tamaño de la discretización debe aumentar inevitablemente para mantener la velocidad  $U$  pequeña y la frecuencia de relajación adimensional inferior a 1,8.

Respecto a la condición de contorno se han utilizado la condición de contorno de Zou & He para la cara superior y BB Full-Way para el resto de las caras. La condición inicial es de velocidad nula y densidad de masa de valor  $\rho_0 = 1,0$ .

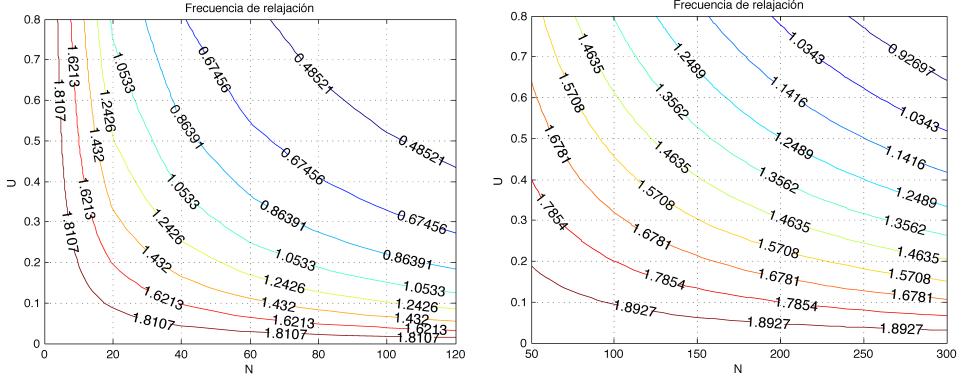


Figura 4.3: Representación de la frecuencia de relajación adimensional en función de la velocidad  $U$  y el tamaño de la discretización  $N$  para  $\text{Re} = 100$  y  $\text{Re} = 1000$ .

Los parámetros más importantes son la frecuencia de relajación adimensional  $\omega$  y el error estacionario  $\varepsilon_{est}$ . Para estudiar el efecto de dichos parámetros, se fijan el número de Reynolds a  $\text{Re} = 100$  y el tamaño de discretización a  $N = 40$ . Los resultados se exponen en la Tabla (4.2) y se representan en la Figura (4.4).

$\varepsilon_{est}$	$U$	$\omega$	$\bar{t}_{ciclo}$	$t_{comp}$	$t_{sim}$	$n$	$e_u (\%)$	$e_v (\%)$
$10^{-6}$	0,625	0,8	DIVER.	DIVER.	DIVER.	DIVER.	DIVER.	DIVER.
$10^{-6}$	0,416666667	1	0,001147	4,348877	39,48958333	3791	2,890275506	21,7008184
$10^{-6}$	0,340909091	1,1	0,001138	4,839645	36,25568182	4254	2,471456865	19,67257395
$10^{-6}$	0,277777778	1,2	0,001137	5,602876	34,21527778	4927	2,229331864	18,49438162
$10^{-6}$	0,224358974	1,3	0,001139	6,544935	32,1786859	5737	2,08170854	17,77796327
$10^{-6}$	0,178571429	1,4	0,001138	7,59971	29,81696429	6679	1,99814116	17,37728297
$10^{-6}$	0,138888889	1,5	0,001139	9,134651	27,85416667	8022	1,96644882	17,22635278
$10^{-6}$	0,104166667	1,6	0,00114	11,281971	25,77604167	9898	1,984753228	17,28013337
$10^{-6}$	0,073529412	1,7	0,001132	14,078148	22,86764706	12440	2,082624986	17,45464319
$10^{-6}$	0,046296296	1,8	DIVER.	DIVER.	DIVER.	DIVER.	DIVER.	DIVER.
$10^{-3}$	0,625	0,8	DIVER.	DIVER.	DIVER.	DIVER.	DIVER.	DIVER.
$10^{-3}$	0,416666667	1	0,001171	2,010088	17,88541667	1717	2,884447093	21,7285509
$10^{-3}$	0,340909091	1,1	0,001155	2,29768	16,96022727	1990	2,463608804	19,69523566
$10^{-3}$	0,277777778	1,2	0,001148	2,672986	16,17361111	2329	2,240882254	18,48678414
$10^{-3}$	0,224358974	1,3	0,001146	3,133317	15,3349359	2734	2,093856382	17,77753028
$10^{-3}$	0,178571429	1,4	0,001149	3,606782	14,01339286	3139	2,014377518	17,39035971
$10^{-3}$	0,138888889	1,5	0,001153	4,24113	12,76736111	3677	1,989030229	17,26396025
$10^{-3}$	0,104166667	1,6	0,001135	5,08511	11,671875	4482	2,017394562	17,3589108
$10^{-3}$	0,073529412	1,7	0,001133	6,364865	10,33088235	5620	2,134216206	17,65039618
$10^{-3}$	0,046296296	1,8	DIVER.	DIVER.	DIVER.	DIVER.	DIVER.	DIVER.

Tabla 4.2: Efecto de variación de la frecuencia de relajación adimensional y del error estacionario para  $\text{Re} = 100$  y  $N = 40$ .

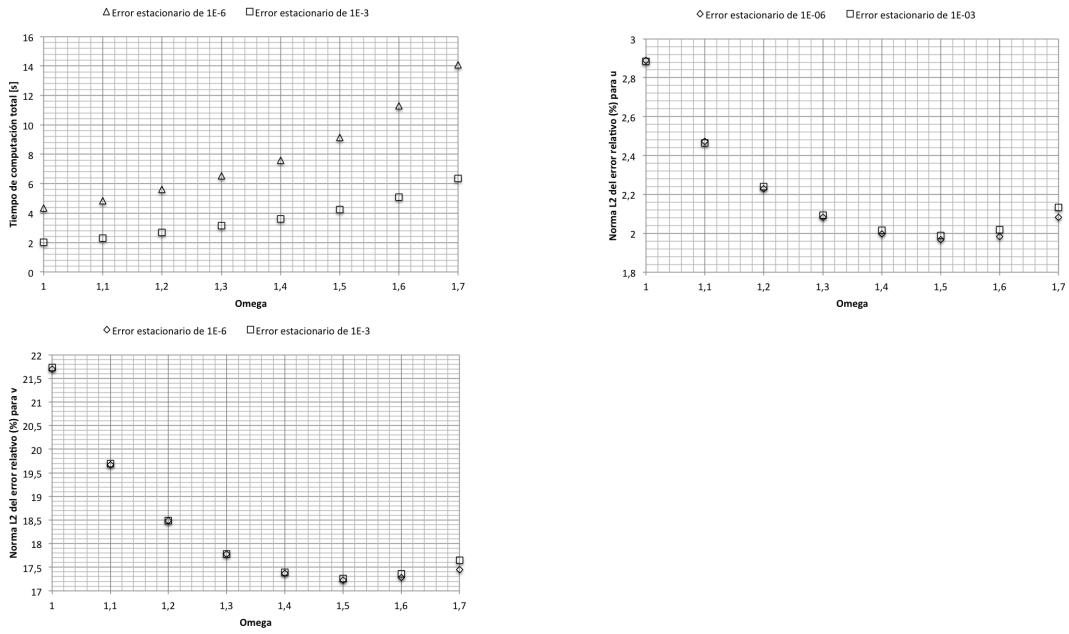


Figura 4.4: Tiempo de computación total, norma  $L_2$  del error relativo de la componente horizontal y vertical de la velocidad en función del error de solver y error en estacionario para  $N = 40$ ,  $Re = 100$  con  $\varepsilon_{est} = 10^{-3}$  y  $\varepsilon_{est} = 10^{-6}$ .

Examinando los datos se concluye que para valores pequeños de  $\omega$  la velocidad  $U$  es grande y se cometan errores debidos a los efectos de compresibilidad, los cuales pueden crear inestabilidades. Para valores grandes de  $\omega$  la viscosidad cinemática tiende a 0 y crea problemas de inestabilidad. Se observa que el valor óptimo de  $\omega$  para obtener la mejor calidad de la solución (menor error) es, para este caso,  $\omega = 1,5$ . A menor error estacionario más tiempo de computación se requiere para llegar al estado estacionario. El aumento del tiempo de computación se hace más notable para valores grandes de  $\omega$ . Para valores pequeños de  $\omega$  el error estacionario no influye mucho en el error en los perfiles de velocidad, sin embargo, para valores grandes de  $\omega$  el error estacionario sí que puede afectar considerablemente al valor del error en los perfiles de velocidad.

### Comparación de FVM y LBM

Al tratarse de métodos muy diferentes, la tarea de comparación se hace muy difícil ya que en ambos métodos intervienen muchos parámetros. Lo que se ha hecho es un estudio de cómo varían los indicadores definidos anteriormente al ir aumentando el

tamaño de la discretización (para un mismo error estacionario). La comparación se ha realizado para  $\text{Re} = 100$  y para una malla uniforme de  $N = 10$  hasta  $N = 110$  utilizando para FVM  $\varepsilon_{sol} = 10^{-6}$  y  $\varepsilon_{est} = 10^{-6}$ ; y para LBM  $\varepsilon_{est} = 10^{-6}$  y  $\omega \approx 1,6$ . Los resultados se exponen en la Tabla (4.3) y se representan en la Figura (4.5).

Tipo	$N$	$\bar{t}_{ciclo}$	$t_{comp}$	$t_{sim}$	$n$	$e_u (\%)$	$e_v (\%)$
LBM	10	0,000063	0,337067	224,3488	5393	12,61547474	58,10955721
FVM	10	0,000198	0,199607	22,117957	1008	4,814622412	17,14236228
LBM	20	0,000272	3,094082	118,2168	11367	3,923162064	34,00134626
FVM	20	0,00073	1,831419	21,1398	2508	1,036178936	3,759720564
LBM	30	0,000619	8,655016	64,7462	13974	2,654125664	23,36469975
FVM	30	0,0016	6,474764	21,215062	4046	0,291425326	2,496178112
LBM	40	0,001131	11,193615	25,7348	9898	1,984971688	17,28068926
FVM	40	0,002768	15,165121	20,879791	5478	0,37264222	2,714343715
LBM	50	0,001799	26,375467	24,3439	14665	1,567776546	13,23095379
FVM	50	0,004286	28,153579	19,659997	6568	0,247897317	2,79979191
LBM	60	0,002655	53,449234	23,15295	20133	1,27523334	10,48754794
FVM	60	0,005535	52,66939	19,222222	8650	0,286117032	2,928873225
LBM	70	0,003662	95,066817	22,25142857	25960	1,071144715	8,470193284
FVM	70	0,00818	97,615392	19,484082	11934	0,280277909	2,961598065
LBM	80	0,004863	161,953105	21,64695	33303	0,926211175	6,964638409
FVM	80	0,009681	168,143565	20,085	16068	0,265945516	3,085225241
LBM	90	0,006035	247,970333	21,45811111	41090	0,813810508	5,810444777
FVM	90	0,01394	264,447754	18,735802	18970	0,221355848	2,814336283
LBM	100	0,007504	381,764157	21,36876	50878	0,729284732	4,894698619
FVM	100	0,016298	386,337645	18,964	23705	0,210208432	2,943417598
LBM	110	0,008954	555,94125	21,44789091	62086	0,665483159	4,167879895
FVM	110	0,020029	577,129717	19,050579	28814	0,205961797	2,948871739

Tabla 4.3: Comparación de LBM y FVM para diferentes tamaños de discretización.

Analizando los datos se ve claramente que el tiempo de computación total es prácticamente igual en ambos métodos para un mismo tamaño de discretización y error estacionario. Por otro lado, la diferencia en el tiempo de computación promedio por ciclo aumenta de manera proporcional a  $cN^2$  dando ventaja al LBM. La calidad de la solución va mejorando a medida que aumenta el tamaño de la discretización. FVM proporciona resultados con menor porcentaje de error en los perfiles de velocidad. Por último, las últimas dos gráficas muestran los errores  $e_u$  y  $e_v$  en función del tiempo de computación total. En ellas se puede ver que para un mismo tiempo de computación FVM proporciona resultados con menor error que LBM.

A continuación se van a comparar los resultados obtenidos para  $\text{Re} = 1000$ . Para LBM se escogen los siguientes parámetros:  $N = 150$ ,  $\omega = 1,5$  y  $\varepsilon_{est} = 6 \cdot 10^{-4}$ ;

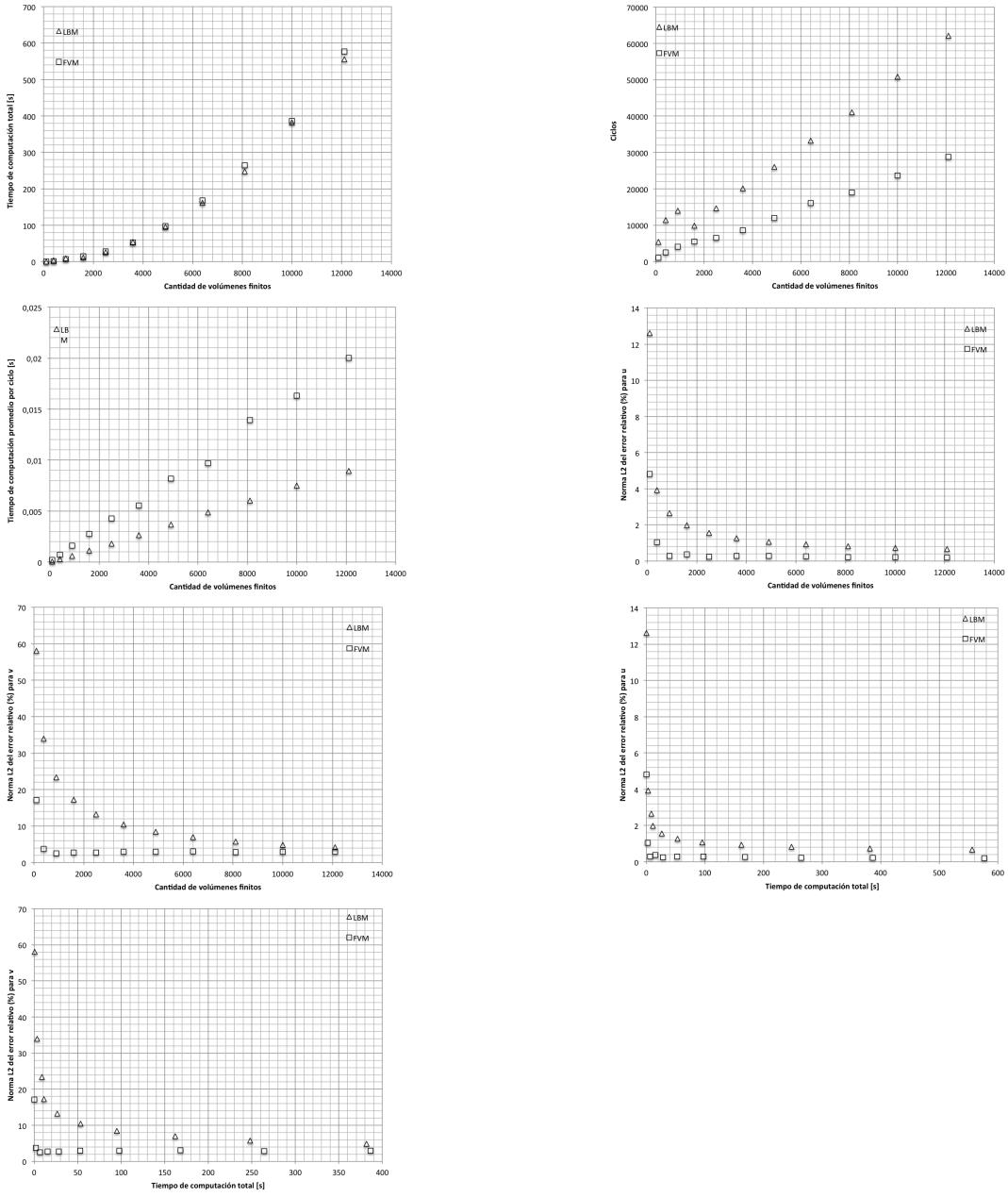


Figura 4.5: Comparación de LBM y FVM para diferentes tamaños de discretización.

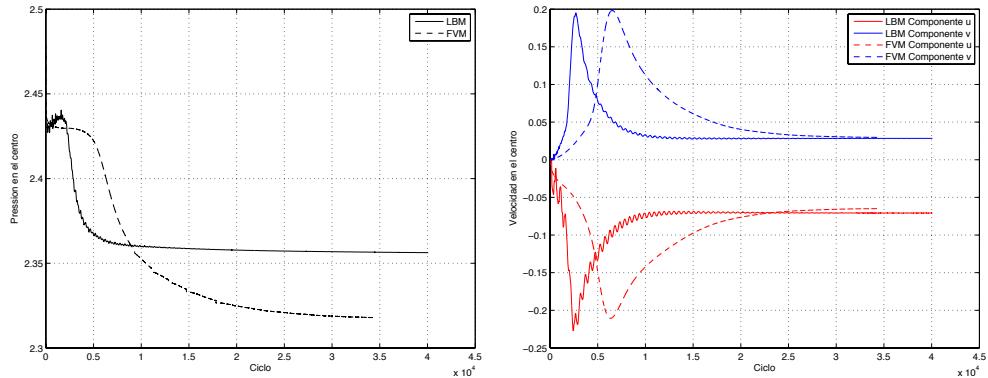


Figura 4.6: Comparación de LBM y FVM para diferentes tamaños de discretización.

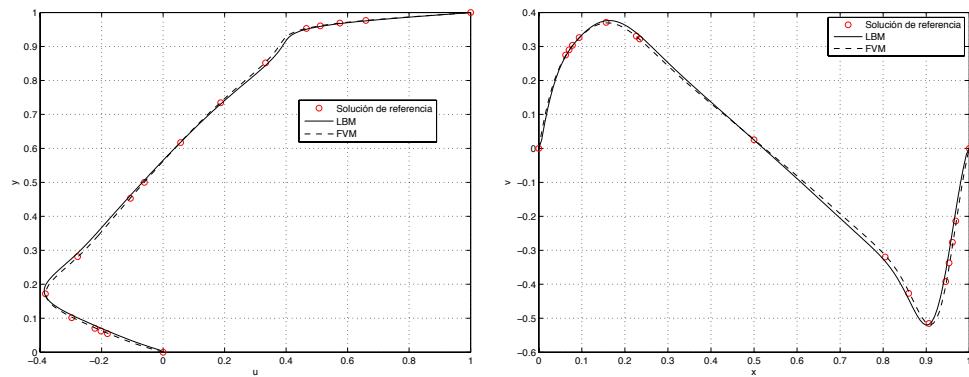


Figura 4.7: Comparación de LBM y FVM para diferentes tamaños de discretización.

mientras que para FVM se escogen los siguientes:  $N = 150$ ,  $\varepsilon_{sol} = 10^{-8}$  (Conjugate Gradient) y  $\varepsilon_{est} = 6 \cdot 10^{-4}$ . Se obtienen los siguientes resultados: para LBM  $t_{comp} = 672,40s$ ,  $e_u = 2,75017\%$  y  $e_v = 12,61961\%$ ; para FVM  $t_{comp} = 6463,68s$ ,  $e_u = 0,47886\%$  y  $e_v = 4,21449\%$ . Los resultados se muestran en las Figuras (4.6-4.8).

Analizando de forma cualitativa se observa que los resultados coinciden de manera considerable entre LBM (línea continua) y FVM (línea discontinua). En la gráfica del campo de presión se observa que LBM tiene fluctuaciones de presión considerables en las dos singularidades (las dos esquinas superiores) mientras que FVM proporciona una solución más suave. Los perfiles de velocidad coinciden de manera notable con la solución de referencia en ambos métodos.

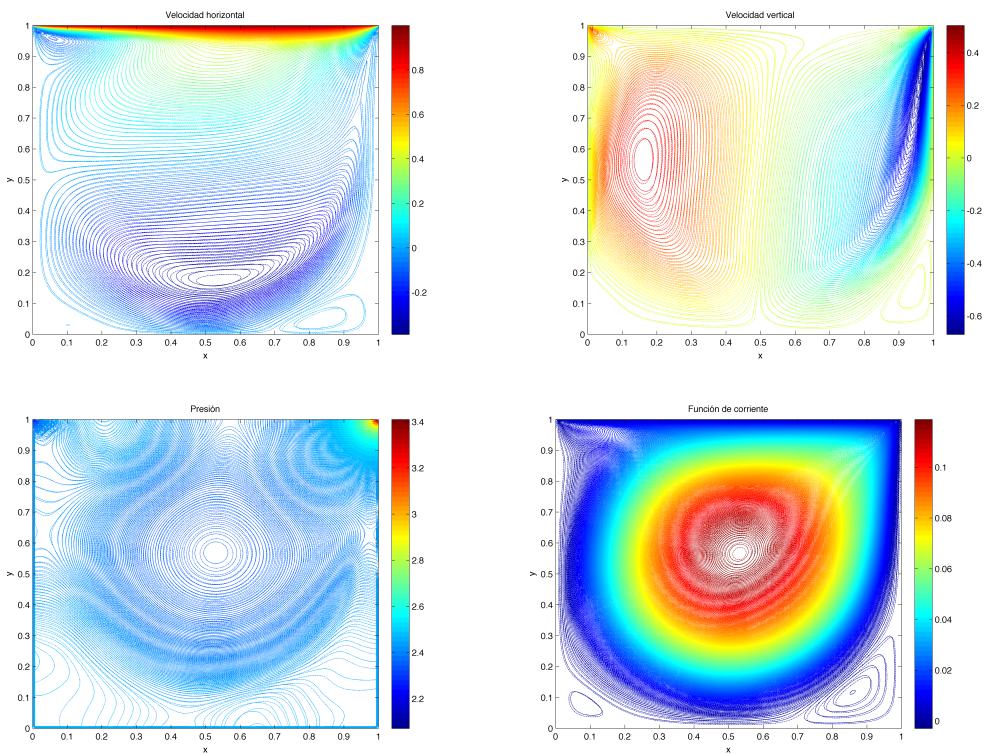


Figura 4.8: Comparación de LBM (línea continua) y FVM (línea discontinua) para diferentes tamaños de discretización.

## 4.2. Differentially Heated Cavity

Se trata de una cavidad cuadrada cerrada de lado  $L$  en la que la pared izquierda de la cavidad se mantiene isotérmica a temperatura  $T_{hot}$  mientras que la pared de la derecha se mantiene a temperatura  $T_{cold}$ . La diferencia de temperatura de ambas paredes se denota  $\Delta T = T_{hot} - T_{cold}$ . Las paredes superior e inferior son adiabáticas y por tanto no permiten la transferencia de calor a través de ellas. El fluido se calienta a lo largo de la pared izquierda y, por el efecto de la flotación, va subiendo y cuando llega a la pared derecha se enfriá y baja y de esta forma hay una circulación de fluido dentro de la cavidad. Como ya se ha dicho anteriormente, los dos parámetros importantes en este tipo de problemas son el número de Prandtl y el número de Rayleigh definidos en el apartado de adimensionalización.

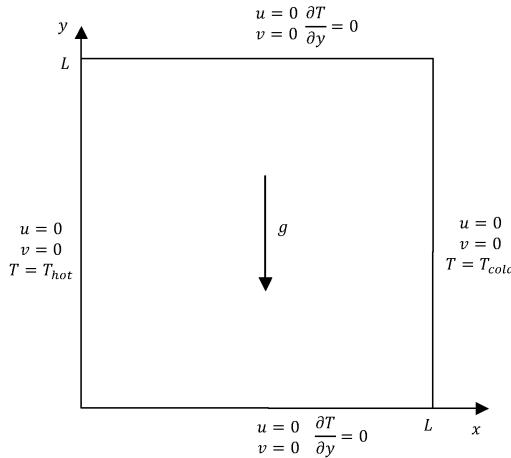


Figura 4.9: Representación esquemática del problema de Heated Cavity.

El número de Nusselt es el flujo de calor adimensional. El número de Nusselt local depende de la posición  $y$  y se calcula de la siguiente manera:

$$\text{Nu}_{x^*y^*} = -\frac{\partial T^*}{\partial x^*} + u^*T^* \quad (4.3)$$

siendo  $T^*$  y  $u^*$  la temperatura y la velocidad horizontal adimensional. Se define el número de Nusselt  $\text{Nu}_x$  como:

$$\text{Nu}_x^* = \int_{y^*=0}^{y^*=1} \text{Nu}_{x^*y^*} dy^* \quad (4.4)$$

$\text{Nu}_{x^*}$  debe de ser independiente de la posición  $x^*$  ya que el flujo de calor total para una sección que está ubicada en la posición  $x^*$  es la misma, sin embargo, debido a los errores en la simulación puede haber variación.

El número de Nusselt medio  $\bar{Nu}$ :

$$\bar{Nu} = \int_{x^*=0}^{x^*=1} Nu_{x^*} dx^* \quad (4.5)$$

Como solución de referencia se utilizará el trabajo de Vahl [24] el cuál proporciona, entre otras cosas, la velocidad horizontal máxima  $u_{max}$  y su posición  $y^*$  ( $x^* = 0, 5$ ), la velocidad vertical máxima  $v_{max}$  y su posición  $x^*$  ( $y^* = 0, 5$ ), la función de corriente en el centro de la cavidad  $\psi_{centro}$ , el número de Nusselt medio  $\bar{Nu}$ , el número de Nusselt local máximo en la pared caliente  $Nu_{max}^{hot}$  y su posición  $y^*$  y por último, el número de Nusselt local mínimo en la pared caliente  $Nu_{min}^{hot}$  y su posición  $y^*$ .

Para caracterizar los resultados se han utilizado los siguientes indicadores: tiempo de computación total  $t_{comp}$  en segundos, tiempo de simulación total  $t_{sim}$ , y las variables que tienen soluciones de referencia y sus respectivos errores relativos (%).

Para definir el estado estacionario se utilizará el error estacionario  $\varepsilon_{est}$  definido en la Ecuación (4.2).

## FVM

Para que los resultados sean adimensionales se toman  $L = 1$ ,  $T_{hot} = 1$  y  $T_{cold} = 0$  ( $\Delta T = 1$ ). Para adimensionalizar el campo de velocidad se utiliza la velocidad característica  $U_0 = a/L_0$  siendo  $L_0 = L = 1$ . Los parámetros  $C_{conv}$  y  $C_{visc}$  son los mismos que en caso de Driven Cavity,  $C_{conv} = 0,14$  y  $C_{visc} = 0,08$ . En este caso se han utilizado dos solvers: por un lado el Gauss-Sheidel (GS) con factor de relajación de 1,1, y por otro lado el Conjugate Gradient (CG). La simulación parte de una condición inicial de velocidad nula y temperatura inicial de 0,5 en los nodos interiores.

## LBM

Para que los resultados sean adimensionales se toman  $L = N$ ,  $T_{hot} = 1$  y  $T_{cold} = 0$  ( $\Delta T = 1$ ). Para adimensionalizar el campo de velocidad se utiliza la velocidad característica  $U_0 = a/N$ . En LBM hace falta definir otra velocidad característica  $U = \sqrt{g\beta\Delta T L}$  para calcular el número de Mach y mantenerlo inferior a un cierto valor para que los errores debido a los efectos de la compresibilidad sean menores. La relación entre  $U_0$  y  $U$  es la siguiente:

$$U = U_0 \sqrt{\text{PrRa}\Delta T} \quad (4.6)$$

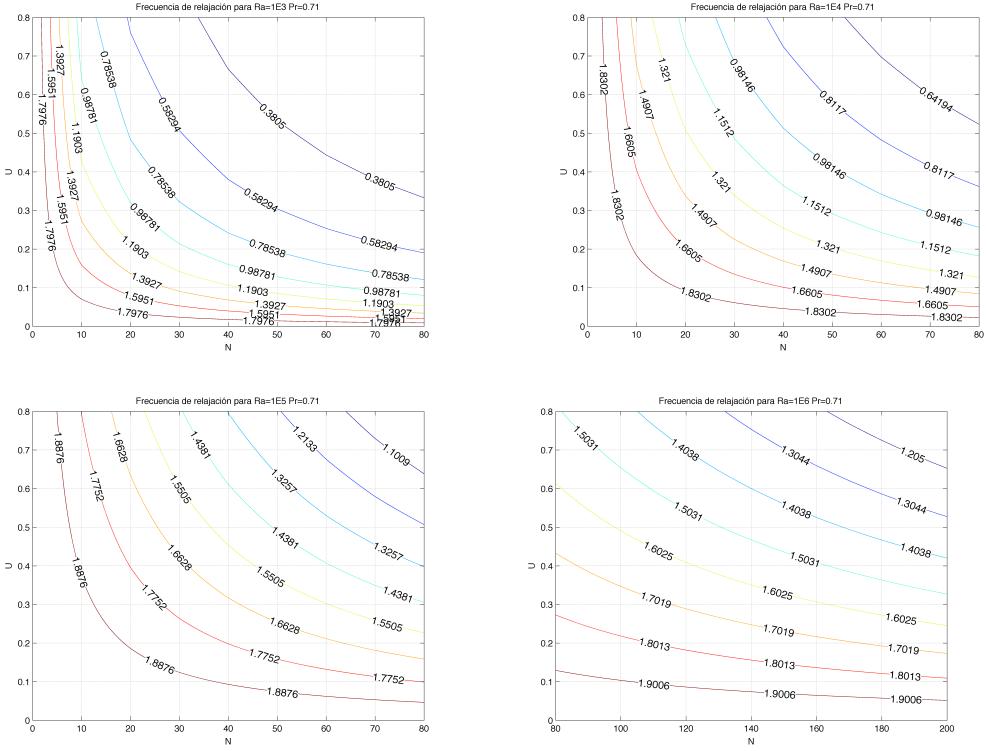


Figura 4.10: Representación de la frecuencia de relajación adimensional en función de la velocidad  $U$  y el tamaño de la discretización  $N$  para  $\text{Ra} = 10^3$ ,  $\text{Ra} = 10^4$ ,  $\text{Ra} = 10^5$  y  $\text{Ra} = 10^6$ .

El procedimiento para calcular todos los parámetros es, dado un valor para la viscosidad cinemática  $\nu$ , calcular el término  $g\beta$  de la definición del número de Rayleigh, posteriormente calcular la velocidad  $U$ , calcular  $U_0$  y por último calcular las frecuencias de relajación  $\omega_f$  y  $\omega_g$ . En [23] se propone otra manera para calcular el valor de la velocidad característica  $U$  utilizando la definición del número de Knudsen. Sin embargo, los resultados obtenidos no son coherentes ya que proporcionan valores de viscosidad cinemática muy pequeños, los cuales hacen que las frecuencias de relajación se acerquen mucho a 2 y por tanto introduzca inestabilidades a la hora de hacer la simulación.

En la Figura (4.10) se representa la frecuencia de relajación adimensional  $\omega_f = \frac{1}{3UN\sqrt{\frac{\text{Pr}}{\text{Ra}} + \frac{1}{2}}}$  en función de la velocidad  $U$  y el tamaño de la discretización  $N$  para  $\text{Ra} = 10^3$ ,  $\text{Ra} = 10^4$ ,  $\text{Ra} = 10^5$  y  $\text{Ra} = 10^6$ , respectivamente.

Respecto a la condición de contorno, se han utilizado la condición de contorno BB

Full-Way para la velocidad y condición adiabática y isotérmica para la temperatura (ver apartado Condición de contorno para modelo térmico). La condición inicial es de velocidad nula, densidad de masa de valor  $\rho_0 = 1,0$  y temperatura de valor  $T = 0,5$ .

Se han hecho simulaciones para diferentes números de Rayleigh en el caso de tener aire en la cavidad ( $\text{Pr} = 0,71$ ) y los resultados obtenidos se muestran en las Tablas (4.4-4.7) para  $\text{Ra} = 10^3\text{-}10^6$ , respectivamente.

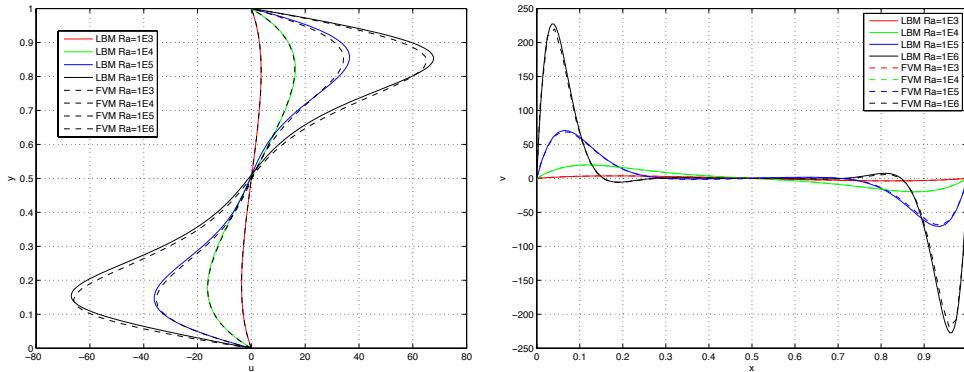


Figura 4.11: Perfiles de velocidad horizontal y vertical para diferentes números de Rayleigh.

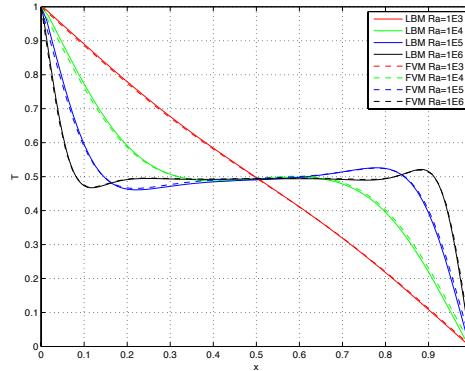


Figura 4.12: Perfil de temperatura para diferentes números de Rayleigh.

Analizando los resultados, se llega a la conclusión de que LBM es apto para hacer simulaciones de convección natural para números de Rayleigh relativamente altos. Sin embargo, respecto a la precisión, se ve claramente la superioridad de FVM, ya que proporciona resultados que son más concordantes con las soluciones de referencia. A medida que se aumenta el número de Rayleigh los errores también aumentan si se mantiene el mismo tamaño de discretización.

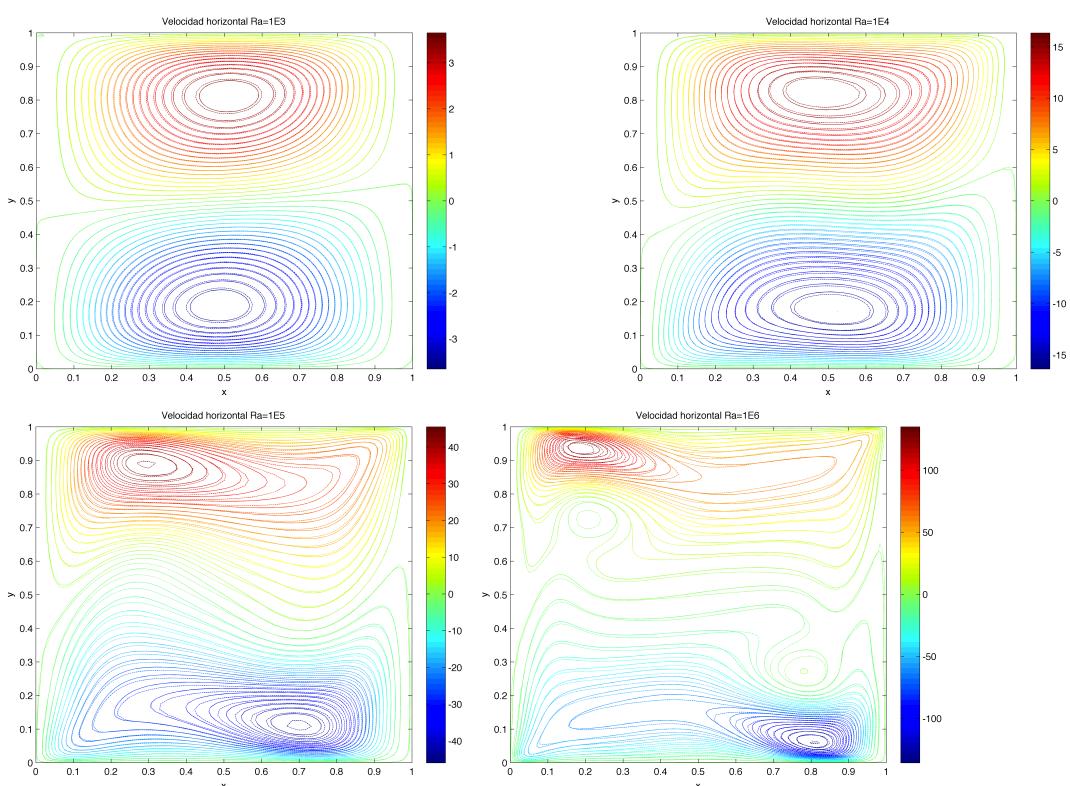


Figura 4.13: Comparación de LBM (línea continua) y FVM (línea discontinua) para el contorno de la velocidad horizontal para diferentes números de Rayleigh.

Tipo	$N$	$\varepsilon_{sol}$	$\nu$	$\varepsilon_{est}$	$t_{comp}$	$t_{sim}$	$n$	$\bar{\text{Nu}}$	$\text{Nu}_{max}^{hot}$	$\text{Nu}_{min}^{hot}$	$u_{max}$	$u_{max}$	$\psi_{centro}$
Ref.	-	-	-	-	-	-	-	1,118(0,000 %)	1,505(0,000 %)	0,692(0,000 %)	3,649(0,000 %)	3,697(0,000 %)	1,174(0,000 %)
LBM	40	-	0,01	$10^{-4}$	DIVER.	DIVER.	DIVER.	1,132(1,226 %)	1,543(2,515 %)	0,703(1,538 %)	3,571(2,133 %)	3,620(2,083 %)	1,144 (2,541 %)
LBM	40	-	0,03	$10^{-4}$	38	0,481	18227	1,1242	1,529(1,573 %)	0,702(1,421 %)	3,555(2,589 %)	3,605(2,488 %)	1,140 (2,898 %)
LBM	40	-	0,05	$10^{-4}$	23	0,495			$y^* = 0,0000$	$y^* = 1,0000$	$y^* = 0,8205$	$x^* = 0,1795$	
LBM	40	-	0,07	$10^{-4}$	18	0,538	8728	1,118(0,031 %)	1,514(0,602 %)	0,701(1,361 %)	3,537(3,063 %)	3,591(2,880 %)	1,136 (3,224 %)
LBM	40	-	0,09	$10^{-4}$	15	0,595	7507	1,111(0,582 %)	1,499(0,375 %)	0,702(1,391 %)	3,519(3,555 %)	3,576(3,261 %)	1,133 (3,521 %)
LBM	40	-	0,11	$10^{-4}$	14	0,632	6630	1,105(1,206 %)	1,485(1,344 %)	0,702(1,410 %)	3,501(4,067 %)	3,563(3,630 %)	1,130 (3,789 %)
LBM	100	-	0,03	$10^{-4}$	1596	0,509	120435	1,123(0,488 %)	1,520(1,016 %)	0,695(0,362 %)	3,618(0,859 %)	3,664(0,883 %)	1,133 (3,521 %)
FVM	40	$10^{-8}$	-	$10^{-4}$	128	0,553	7859	1,119(0,104 %)	1,513(0,512 %)	0,689(0,467 %)	3,658(0,236 %)	3,710(0,340 %)	1,180 (0,545 %)
FVM	80	$10^{-8}$	-	$10^{-4}$	568	0,576	18413	1,118(0,036 %)	1,509(0,284 %)	0,690(0,290 %)	3,654(0,134 %)	3,712(0,412 %)	1,177 (0,273 %)
								$y^* = 0,0917$	$y^* = 0,9917$	$y^* = 0,8083$	$x^* = 0,1750$		

Tabla 4.4: Comparación de LBM y FVM para  $\text{Ra} = 10^3$ .

Tipo	$N$	$\varepsilon_{sol}$	$\nu$	$\varepsilon_{est}$	$t_{comp}$	$t_{sim}$	$n$	$\bar{N}_{\text{u}}$	$N_{\text{u}}^{hot}_{max}$	$N_{\text{u}}^{hot}_{min}$	$u_{max}$	$v_{max}$	$\psi_{centro}$
Ref.	-	-	-	-	-	-	-	$2,243(0,000 \%)$	$3,528(0,000 \%)$	$0,586(0,000 \%)$	$16,178(0,000 \%)$	$19,617(0,000 \%)$	$5,071(0,000 \%)$
LBM	40	-	0,03	$10^{-4}$	39	0,498	18868	$2,299(2,497 \%)$	$3,723(5,528 \%)$	$0,578(1,444 \%)$	$16,511(2,057 \%)$	$19,922(1,556 \%)$	$5,185(2,254 \%)$
LBM	60	-	0,03	$10^{-4}$	184	0,456	38824	$2,238(0,211 \%)$	$3,629(2,865 \%)$	$0,581(0,781 \%)$	$16,343(1,019 \%)$	$19,830(1,087 \%)$	$5,136(1,272 \%)$
FVM	40	$10^{-8}$	-	$10^{-4}$	198	0,663	9408	$2,260(0,742 \%)$	$3,585(1,616 \%)$	$0,579(1,129 \%)$	$16,089(0,549 \%)$	$19,756(0,707 \%)$	$5,089(0,345 \%)$
FVM	60	$10^{-8}$	-	$10^{-4}$	995	0,667	21317	$2,251(0,375 \%)$	$3,556(0,787 \%)$	$0,582(0,750 \%)$	$16,142(0,225 \%)$	$19,676(0,301 \%)$	$5,080(0,183 \%)$

Tabla 4.5: Comparación de LBM y FVM para  $\text{Ra} = 10^4$ .

Typo	$N$	$\varepsilon_{sol}$	$\nu$	$\varepsilon_{est}$	$t_{comp}$	$t_{sim}$	$n$	$\bar{\text{Nu}}$	$\text{Nu}_{max}^{hot}$	$\text{Nu}_{min}^{hot}$	$u_{max}$	$v_{max}$	$\psi_{centro}$
Ref.	-	-	-	-	-	-	-	4,519(0,000 %)	7,717(0,000 %)	0,729(0,000 %)	34,730(0,000 %)	68,590(0,000 %)	9,111(0,000 %)
LBM	40	-	0,03	$10^{-4}$	46	0,576	21816	4,748(5,066 %)	8,336(8,018 %)	0,556(23,788 %)	37,676(8,482 %)	72,937(6,337 %)	10,207 (12,029 %)
LBM	60	-	0,03	$10^{-4}$	200	0,492	41946	4,696(3,919 %)	8,368(8,441 %)	0,654(10,354 %)	36,374(4,733 %)	71,659(4,475 %)	9,680 (6,246 %)
LBM	80	-	0,03	$10^{-4}$	563	0,437	66250	4,661(3,132 %)	8,215(6,450 %)	0,685(5,998 %)	35,815(3,125 %)	70,626(2,969 %)	9,480 (4,054 %)
FVM	40	$10^{-8}$	-	$10^{-4}$	285	0,578	11491	4,623(2,305 %)	8,168(5,846 %)	0,713(2,174 %)	34,015(2,058 %)	69,324(1,070 %)	9,213 (1,116 %)
FVM	60	$10^{-8}$	-	$10^{-4}$	1104	0,546	17509	4,567(1,063 %)	7,925(2,691 %)	0,720(1,226 %)	34,268(1,330 %)	68,550(0,058 %)	9,159 (0,527 %)

Tabla 4.6: Comparación de LBM y FVM para  $\text{Ra} = 10^5$ .

Tipo	$N$	$\varepsilon_{sol}$	$\nu$	$\varepsilon_{est}$	$t_{comp}$	$t_{sim}$	$n$	$\bar{N}_{\text{u}}$	$\text{Nu}_{max}^{hot}$	$\text{Nu}_{min}^{hot}$	$u_{max}$	$v_{max}$	$\psi_{centro}$
Ref.	-	-	-	-	-	-	-	8,800(0,000 %)	17,925(0,000 %)	0,989(0,000 %)	64,630(0,000 %)	219,360(0,000 %)	16,320(0,000 %)
LBM	100	-	0,03	$10^{-4}$	1356	0,444	105042	9,119(3,623 %)	19,536(8,986 %)	0,466(52,852 %)	73,656(13,966 %)	238,920(8,917 %)	18,328 (12,306 %)
LBM	200	-	0,03	$10^{-4}$	20630	0,339	320463	9,036(2,676 %)	18,582(3,668 %)	0,793(19,783 %)	67,710(4,766 %)	227,072(3,516 %)	17,048 (4,462 %)
FVM	40	$10^{-8}$	-	$10^{-4}$	614	0,327	21017	9,427(7,120 %)	20,384(13,716 %)	0,928(6,137 %)	65,484(1,321 %)	224,246(2,228 %)	17,015 (4,260 %)
FVM	60	$10^{-8}$	-	$10^{-4}$	2608	0,336	31877	9,102(3,427 %)	19,388(8,162 %)	0,956(3,304 %)	65,121(0,760 %)	219,985(0,285 %)	16,695 (2,298 %)

Tabla 4.7: Comparación de LBM y FVM para  $\text{Ra} = 10^6$ .

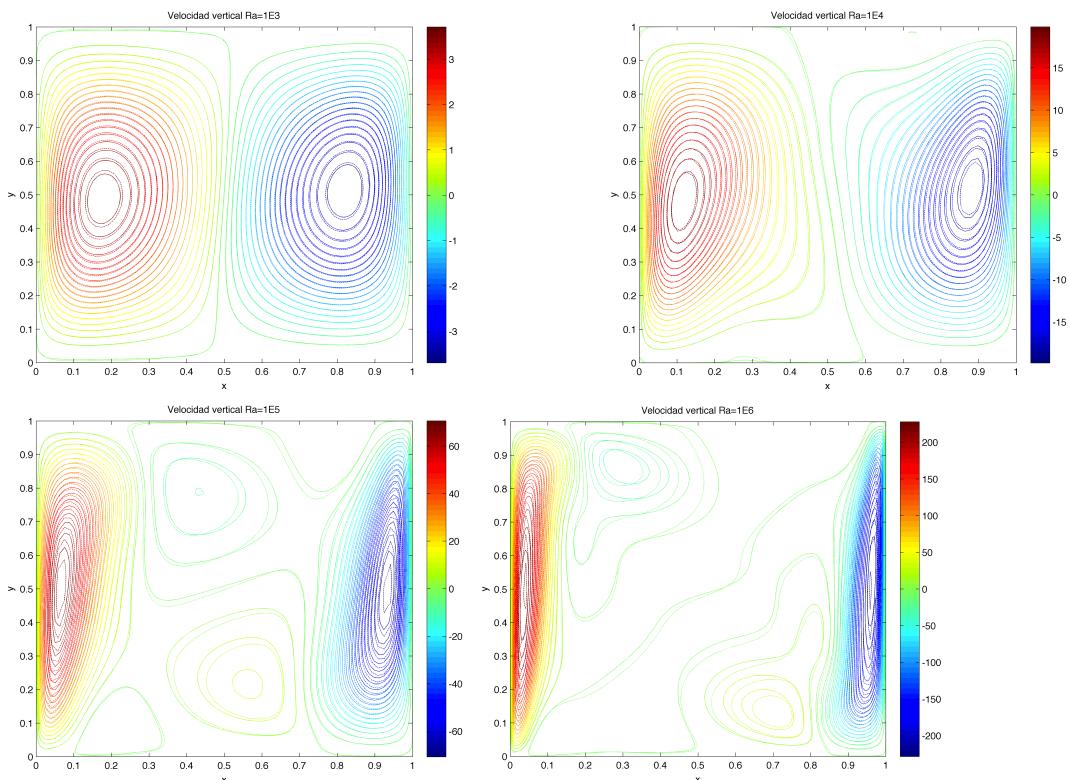


Figura 4.14: Comparación de LBM (línea continua) y FVM (línea discontinua) para el contorno de la velocidad vertical para diferentes números de Rayleigh.

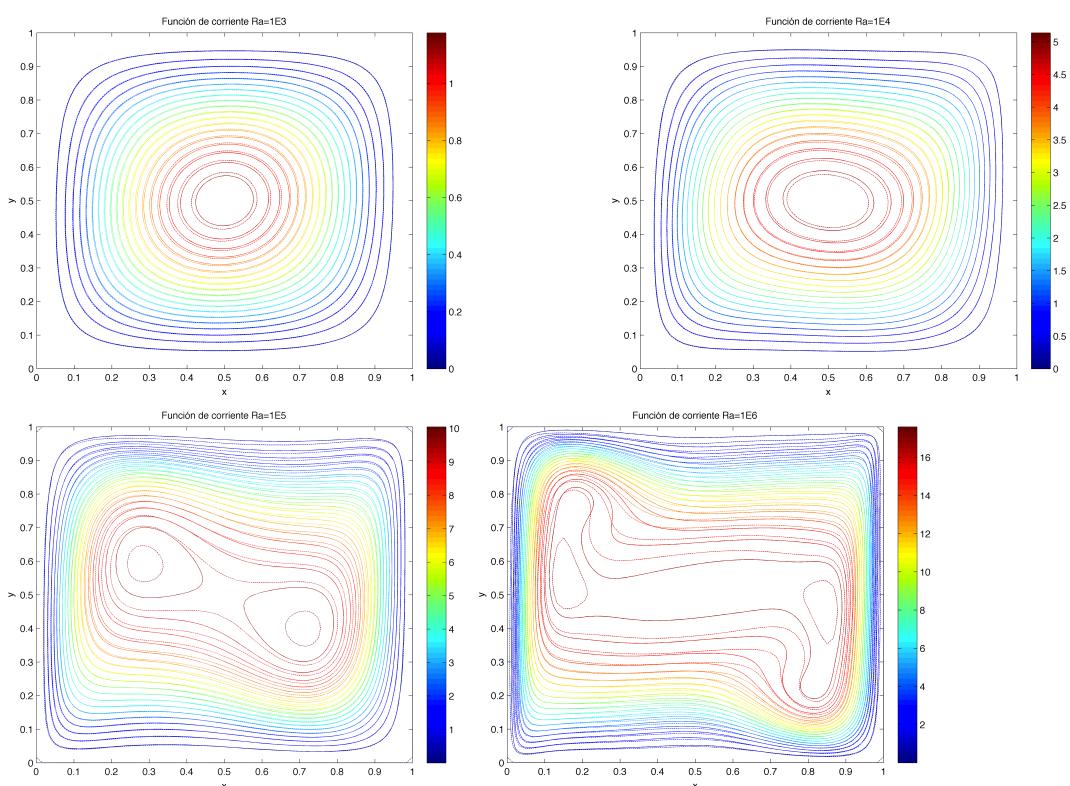


Figura 4.15: Comparación de LBM (línea continua) y FVM (línea discontinua) para la función de corriente para diferentes números de Rayleigh.

## Capítulo 5

# Conclusiones

A continuación se listan las conclusiones principales.

- La teoría sobre la que se basa LBM resulta ser más complicada y laboriosa que la de FVM.
- La implementación práctica de LBM es más fácil que en FVM.
- A nivel de implementación, en LBM es más fácil incorporar fronteras dentro del dominio que en FVM.
- LBM puede ofrecer soluciones precisas a problemas de dinámica de fluidos. Sin embargo, hay restricciones que se deben considerar. La más importante de ellas es mantener el número de Mach pequeño, con lo cual no es posible hacer simulaciones compresibles.
- Para un mismo tamaño de discretización y error estacionario, FVM requiere un tiempo de computación similar a LBM. Sin embargo, el error en la solución del FVM es más pequeño que en LBM.

## Capítulo 6

# Impacto medioambiental

El estudio apenas ha tenido impacto ambiental alguno ya que tanto la energía eléctrica consumida por los ordenadores, como los recursos físicos utilizados (el papel, CD, etc.) son de cantidad reducida. Sin embargo, el estudio hecho puede llegar a tener efectos favorables (o menos desfavorables) para el medio ambiente, en el sentido de que el trabajo puede llegar a motivar a otros a hacer un estudio más profundo para mejorar las herramientas de simulación de dinámica de fluidos y que éstas puedan llegar a diseñar máquinas que sean más eficientes enérgicamente y, por tanto, respecten más el medio ambiente.

## Capítulo 7

# Presupuesto

Según los créditos que posee un Proyecto de Final de Carrera en la Escuela Técnica Superior de Ingenierías Industrial y Aeronáutica de Terrassa de la Universidad Politécnica de Catalunya, las horas de trabajo correspondientes son 338. Aun así, se han añadido 300 horas más de dedicación extra. Suponiendo que el sueldo de un ingeniero becario es de 10 €/hora el coste personal asciende a 6380 €.

Para la realización del estudio ha sido necesario cierto material informático. En concreto, se ha utilizado un ordenador personal de coste de 1500 €.

Finalmente, se debe de tener en cuenta el coste del software empleado. Se han utilizado el sistema operativo de Mac OS incorporado en el portátil, el software de programación gratuito NetBeans, el software Matlab para representar los resultados y LaTeX para escribir la memoria. Matlab (versión de estudiante) tiene un coste de 80 euros mientras que la distribución de LaTeX utilizado es gratuito.

El coste total del trabajo asciende a 7960 €.

## Capítulo 8

# Futuros estudios

Debido a que el tiempo que se ha dedicado para realizar este proyecto es limitado, hay muchos aspectos que han quedado fuera del alcance pero que son muy importantes de estudiar. A continuación se van a listar estudios futuros posibles.

- Estudio más profundo sobre los métodos de implementación optimizados de LBM.
- Utilización del modelo MRT.
- Utilización del modelo Extended Lattice Boltzmann Method (ELBE) que introduce modelos de turbulencia y hace posible simular problemas con número de Reynolds o Rayleigh altos.
- Extensión a casos 3D.
- Geometrías curvas.
- Implementación paralela.

# Bibliografía

- [1] Ricardo Brito López, “Teoría cinética de gases de red”, Tesis Doctoral, 2001.
- [2] Leopoldo García-Colín Scherer, “La física de los procesos irreversibles”.
- [3] Kerson Huang, “Statistical Mechanics” (second edition), Massachusetts Institute of Technology, 1987.
- [4] Sauro Succi, “The Lattice Boltzmann Equation for Fluid Dynamics and Beyond”, Oxford University Press, 2001.
- [5] Gilberto Medeiros Kremer, “An Introduction to the Boltzmann Equation and Transport Processes in Gases”, Springer, 2010.
- [6] Dieter A. Wolf-Gladrow, “Lattice-Gas Cellular Automata and Lattice Boltzmann Models – An Introduction”, Springer, 2005.
- [7] P. L. Bhatnagar, E. P. Gross, and M. Krook, A model for collision processes in gas. I. Small amplitude processes in charged and neutral one-component systems, Phys. Rev., 94 (1954), pp. 511–525.
- [8] X. He and L.-S. Luo. Theory of the lattice Boltzmann method: from Boltzmann equation to the lattice Boltzmann equation. Physical Review E, 56:6811, 1997.
- [9] Q. Zou and X. He, “On Pressure and Velocity Flow Boundary Conditions and Bounceback for the Lattice Boltzmann BGK Model”, Physics of Fluids, Vol. 9, 1997, pp. 1591-1598. doi:10.1063/1.869307.
- [10] Chih-Fung Ho, Cheng Chang, Kuen-Hau Lin and Chao-An Lin, “Consistent Boundary Conditions for 2D and 3D Lattice Boltzmann Simulations”, CMES (Computer Modeling in Engineering & Sciences), vol.44, no. 2, pp. 137-155, 2009.

- [11] Inamuro T, Yoshino M, Ogino F. A non-slip boundary condition for lattice Boltzmann simulations. *Phys Fluids* 1995;7:2928–30.
- [12] Chih-Hao Liu, Kuen-Hau Lin, Hao-Chueh Mai, Chao-An Lin, Thermal boundary conditions for thermal lattice Boltzmann simulations, *Computers & Mathematics with Applications*, Volume 59, Issue 7, April 2010, Pages 2178-2193, ISSN 0898-1221, 10.1016/j.camwa.2009.08.043.
- [13] Yu, D., Mei, R., Luo L.S., and Shyy, W., Viscous flow computations with the method of lattice Boltzmann equation, *Progress in Aerospace Sciences*, 2003, 39, pp. 329-367.
- [14] D. d’Humières, I. Ginzburg, M. Krafczyk, P. Lallemand and Li-Shi Luo. Multiple-relaxation-time lattice Boltzmann models in three dimensions. *Philosophical Transactions of Royal Society of London A*,360(1792):437–451, 2002.
- [15] Bouzidi M, d’Humières D, Lallemand P, Luo L-S. Lattice Boltzmann equation on a two-dimensional rectangular grid. *J Comput Phys* 2001;172:704-17.
- [16] X.He, L.-S. Luo: Lattice Boltzmann model for the incompressible Navier-Stokes equation, *J. Stat. Phys.* 88,927(1997).
- [17] R. Mei, D. Yu, W. Shyy, L.-S. Luo, Force evaluation in the lattice Boltzmann method involving curved geometry, *Phys. Rev. E* 65 (2002) 041203.
- [18] J.H. Ferziger, M. Peric, *Computational Methods for Fluid Dynamics*, 3rd Edition, Springer, 2002
- [19] U. Ghia, K.N. Ghia, C.T. Shin, High-Re solutions for incompressible flow using the Navier–Stokes equations and a multigrid method, *Journal of Computational Physics* 48 (3) (1982) 387–411.
- [20] J. Shewchuk, An introduction to the Conjugate Gradient Method without the agonizing pain, Carnegie Mellon University, 1994.
- [21] Introduction to the Fractional Step Method by CTTC, UPC
- [22] A.A. Mohamad, A. Kuzmin, A critical evaluation of force term in lattice Boltzmann method, natural convection problem, *International Journal of Heat and Mass Transfer*, Volume 53, Issues 5–6, February 2010, Pages 990-996, ISSN 0017-9310, 10.1016/j.ijheatmasstransfer.2009.11.014.

- [23] P.-H. Kao, Y.-H. Chen, R.-J. Yang, Simulations of the macroscopic and mesoscopic natural convection flows within rectangular cavities, International Journal of Heat and Mass Transfer, Volume 51, Issues 15–16, 15 July 2008, Pages 3776-3793, ISSN 0017-9310, 10.1016/j.ijheatmasstransfer.2008.01.003.
- [24] de Vahl Davis G.: Natural convection of air in a square cavity: a bench mark numerical solution. Int J Numer Methods Fluids, 3:249-264, 1983.

## Anexo A

# Código fuente de LBM

```
1 #ifndef LBM_H
2 #define LBM_H
3 #include <vector.h>
4 class ComputationTime{
5     clock_t start;
6 public:
7     void Start(){
8         start = clock();
9     }
10    double Stop(){
11        return (((double) (clock() - start)) / CLOCKS_PER_SEC);
12    }
13};
14 void Swap( double &A, double &B){
15     double tmp = A;
16     A = B;
17     B = tmp;
18}
19 class LatticeModel{
20 public:
21     LatticeModel( int D, int Q, double *w, int *e_x, int *e_y, int *e_z, int *
22     opp, double cs){
23         this->D = D;
24         this->Q = Q;
25         this->cs = cs;
26         this->w = new double [ Q ];
27         this->e_x = new int [ Q ];
28         this->e_y = new int [ Q ];
29         this->opp = new int [ Q ];
30         for( int i = 0; i < Q; i++ ){
31             this->w[ i ] = w[ i ];
32             this->opp[ i ] = opp[ i ];
33             switch ( D ){
34                 case 1:
35                     this->e_x[ i ] = e_x[ i ];
36                     break;
37                 case 2:
38                     this->e_x[ i ] = e_x[ i ];
39                     this->e_y[ i ] = e_y[ i ];
40                     break;
41                 case 3:
42                     this->e_x[ i ] = e_x[ i ];
43                     this->e_y[ i ] = e_y[ i ];
44                     this->e_z[ i ] = e_z[ i ];
45             }
46         }
47     }
48 }
```

```

44             break;
45         }
46     }
47 }
48 double *w;
49 int *e_x;
50 int *e_y;
51 int *e_z;
52 int *opp;
53 double cs;
54 int D;
55 int Q;
56 };
57 class Lattice {
58 public:
59     Lattice () {
60     }
61     void SetLatticeModel ( LatticeModel *lattice_model ) {
62         this->lattice_model = lattice_model;
63         this->f = new double [ lattice_model->Q ];
64         isWet = true;
65         isBC = false;
66     }
67     inline double CalculateMacroscopicProperties ( void ) {
68         rho = 0;
69         u = 0;
70         v = 0;
71         for( int i = 0; i < lattice_model->Q; i++ ) {
72             rho += f[ i ];
73             u += f[ i ] * lattice_model->e_x[ i ];
74             v += f[ i ] * lattice_model->e_y[ i ];
75         }
76         u /= rho;
77         v /= rho;
78     }
79     double *f;
80     double rho;
81     double u;
82     double v;
83     LatticeModel *lattice_model;
84     bool isWet;
85     bool isBC;
86 };
87 class Point {
88 public:
89     Point () {
90         i = j = 0;
91     }
92     Point ( const int i, const int j ) {
93         this->i = i;
94         this->j = j;
95     }
96     int i, j;
97 };
98 enum BC_TYPE { NORTH_VELOCITY_ZOU_HE, SOUTH_VELOCITY_ZOU_HE,
EAST_VELOCITY_ZOU_HE, WEST_VELOCITY_ZOU_HE, NORTH_PRESSURE_ZOU_HE,
SOUTH_PRESSURE_ZOU_HE, EAST_PRESSURE_ZOU_HE, WEST_PRESSURE_ZOU_HE,
NORTH_WEST_CONVEX_CORNER_VELOCITY_ZOU_HE,
NORTH_EAST_CONVEX_CORNER_VELOCITY_ZOU_HE,
SOUTH_WEST_CONVEX_CORNER_VELOCITY_ZOU_HE,
SOUTH_EAST_CONVEX_CORNER_VELOCITY_ZOU_HE,
NORTH_WEST_CONCAVE_CORNER_VELOCITY_ZOU_HE,
NORTH_EAST_CONCAVE_CORNER_VELOCITY_ZOU_HE,

```

```

SOUTH_WEST_CONCAVE_CORNER_VELOCITY_ZOU_HE,
SOUTH_EAST_CONCAVE_CORNER_VELOCITY_ZOU_HE, HALF_WAY_BOUNCEBACK,
FULL_WAY_BOUNCEBACK, EAST_OPEN_BOUNDARY};

99 class BoundaryCondition {
100 public:
101     BoundaryCondition( BC_TYPE type, Point start_point, Point end_point,
102                         double u, double v, double rho){
103         this->type = type;
104         this->start_point = start_point;
105         this->end_point = end_point;
106         this->u = u;
107         this->v = v;
108         this->rho = rho;
109     }
110     BC_TYPE type;
111     Point start_point;
112     Point end_point;
113     double u;
114     double v;
115     double rho;
116 };
117 class Simulation{
118 public:
119     Simulation( int Nx, int Ny, LatticeModel *lattice_model, double omega){
120         domain = new Lattice*[ Nx ];
121         this->Nx = Nx;
122         this->Ny = Ny;
123         this->lattice_model = lattice_model;
124         time = 0;
125         this->omega = omega;
126         for( int i = 0; i < Nx; i++ ){
127             domain[ i ] = new Lattice[ Ny ];
128         }
129         for( int i = 0; i < Nx; i++ ){
130             for( int j = 0; j < Ny; j++ ){
131                 domain[ i ][ j ].SetLatticeModel( lattice_model );
132             }
133             startPointForceEvaluation.i = startPointForceEvaluation.j =
134             endPointForceEvaluation.i = endPointForceEvaluation.j = 0;
135         }
136         void InitEquilibrium( double density ) {
137             for( int i = 0; i < Nx; i++ ){
138                 for( int j = 0; j < Ny; j++ ){
139                     for( int alfa = 0; alfa < lattice_model->Q; alfa++ ){
140                         domain[ i ][ j ].f[ alfa ] = lattice_model->w[ alfa ] *
141                         density;
142                     }
143                 }
144             }
145         void StreamStep( ){
146             int ii , jj;
147             int alfa;
148             for( int j = 0; j < Ny; j++ ){
149                 for( int i = Nx-1; i >= 0; i-- ){
150                     if( !(domain[ i ][ j ].isBC == false && domain[ i ][ j ].isWet
151                         == false) ){
152                         alfa = 1;
153                         ii = i + lattice_model->e_x[ lattice_model->opp[ alfa ] ];
154                         jj = j + lattice_model->e_y[ lattice_model->opp[ alfa ] ];

```

```

154         if( IndiceLogico( ii , 0 , Nx-1) && IndiceLogico( jj , 0 , Ny
155             -1)){
156             domain[ i ][ j ].f[ alfa ] = domain[ ii ][ jj ].f[ alfa
157                 ];
158         }
159     }
160     for( int i = 0; i < Nx; i++ ){
161         if( !(domain[ i ][ j ].isBC == false && domain[ i ][ j ].isWet
162             == false) ){
163             alfa = 3;
164             ii = i + lattice_model->e_x[ lattice_model->opp[ alfa ] ];
165             jj = j + lattice_model->e_y[ lattice_model->opp[ alfa ] ];
166             if( IndiceLogico( ii , 0 , Nx-1) && IndiceLogico( jj , 0 , Ny
167                 -1)){
168                 domain[ i ][ j ].f[ alfa ] = domain[ ii ][ jj ].f[
169                     alfa ];
170             }
171         }
172     }
173     for( int j = 0; j < Ny; j++ ){
174         for( int i = 0; i < Nx; i++ ){
175             if( !(domain[ i ][ j ].isBC == false && domain[ i ][ j ].isWet
176                 == false) ){
177                 alfa = 4;
178                 ii = i + lattice_model->e_x[ lattice_model->opp[ alfa ] ];
179                 jj = j + lattice_model->e_y[ lattice_model->opp[ alfa ] ];
180                 if( IndiceLogico( ii , 0 , Nx-1) && IndiceLogico( jj , 0 , Ny
181                     -1)){
182                     domain[ i ][ j ].f[ alfa ] = domain[ ii ][ jj ].f[
183                         alfa ];
184                 }
185             }
186         }
187         for( int i = Nx-1; i >= 0; i-- ){
188             if( !(domain[ i ][ j ].isBC == false && domain[ i ][ j ].isWet
189                 == false) ){
190                 alfa = 8;
191                 ii = i + lattice_model->e_x[ lattice_model->opp[ alfa ] ];
192                 jj = j + lattice_model->e_y[ lattice_model->opp[ alfa ] ];
193                 if( IndiceLogico( ii , 0 , Nx-1) && IndiceLogico( jj , 0 , Ny
194                     -1)){
195                     domain[ i ][ j ].f[ alfa ] = domain[ ii ][ jj ].f[
196                         alfa ];
197                 }
198             }
199         }
200         for( int j = Ny-1; j >= 0; j-- ){
201             for( int i = 0; i < Nx; i++ ){
202                 if( !(domain[ i ][ j ].isBC == false && domain[ i ][ j ].isWet
203                     == false) ){
204                     alfa = 2;
205                     ii = i + lattice_model->e_x[ lattice_model->opp[ alfa ] ];

```

```

203     jj = j + lattice_model->e_y[ lattice_model->opp[ alfa ] ];
204     if( IndiceLogico( ii , 0, Nx-1) && IndiceLogico( jj , 0, Ny
205         -1)){
206         domain[ i ][ j ].f[ alfa ] = domain[ ii ][ jj ].f[
207             alfa ];
208     }
209     alfa = 6;
210     ii = i + lattice_model->e_x[ lattice_model->opp[ alfa ] ];
211     jj = j + lattice_model->e_y[ lattice_model->opp[ alfa ] ];
212     if( IndiceLogico( ii , 0, Nx-1) && IndiceLogico( jj , 0, Ny
213         -1)){
214         domain[ i ][ j ].f[ alfa ] = domain[ ii ][ jj ].f[
215             alfa ];
216     }
217     }
218     for( int i = Nx-1; i >= 0; i-- ){
219         if( !(domain[ i ][ j ].isBC == false && domain[ i ][ j ].isWet
220             == false) ){
221             alfa = 5;
222             ii = i + lattice_model->e_x[ lattice_model->opp[ alfa ] ];
223             jj = j + lattice_model->e_y[ lattice_model->opp[ alfa ] ];
224             if( IndiceLogico( ii , 0, Nx-1) && IndiceLogico( jj , 0, Ny
225                 -1)){
226                 domain[ i ][ j ].f[ alfa ] = domain[ ii ][ jj ].f[
227                     alfa ];
228             }
229         }
230     }
231     void CollisionStep( ){
232         double feq;
233         for( int i = 0; i < Nx; i++ ){
234             for( int j = 0; j < Ny; j++ ){
235                 if( domain[ i ][ j ].isWet == true ){
236                     for( int alfa = 0; alfa < lattice_model->Q; alfa++ ){
237                         feq = lattice_model->w[ alfa ]*domain[ i ][ j ].rho*(1 +
238                             3.0*(lattice_model->e_x[ alfa ]*domain[ i ][ j ].u +
239                             lattice_model->e_y[ alfa ]*domain[ i ][ j ].v) +
240                             (9.0/2.0)*pow(lattice_model->e_x[ alfa ]*domain[ i ][ j ].u +
241                               lattice_model->e_y[ alfa ]*domain[ i ][ j ].v,2.0)
242                             -(3.0/2.0)*(domain[ i ][ j ].u*domain[ i ][ j ].u + domain
243                               [ i ][ j ].v*domain[ i ][ j ].v));
244                         domain[ i ][ j ].f[ alfa ] = domain[ i ][ j ].f[ alfa
245                             ] - omega * ( domain[ i ][ j ].f[ alfa ] - feq );
246                     }
247                 }
248             }
249         }
250     void MacroscopicProperties( ){
251         double old_u, old_v, old_rho;
252         double error_u, error_v, error_rho;
253         convergence_u = convergence_v = convergence_rho = 0.0;
254         for( int i = 0; i < Nx; i++ ){
255             for( int j = 0; j < Ny; j++ ){
256                 if( domain[ i ][ j ].isWet == true ){
257                     old_u = domain[ i ][ j ].u;
258                     old_v = domain[ i ][ j ].v;
259                     old_rho = domain[ i ][ j ].rho;
260                     domain[ i ][ j ].CalculateMacroscopicProperties();
261                     error_u = fabs( old_u - domain[ i ][ j ].u );

```

```

252     error_v = fabs( old_v - domain[ i ][ j ].v );
253     error_rho = fabs( old_rho - domain[ i ][ j ].rho );
254     if( error_u > convergence_u ) convergence_u = error_u;
255     if( error_v > convergence_v ) convergence_v = error_v;
256     if( error_rho > convergence_rho ) convergence_rho =
257         error_rho;
258   }
259 }
260
261 void BoundaryConditions( ){
262   for( int i = 0; i < bc.size(); i++ ){
263     ApplySimpleBoundaryCondition( bc[ i ] );
264   }
265 }
266 void AddBoundaryCondition( BoundaryCondition newBC ) {
267   bc.push_back( newBC );
268   switch( newBC.type ){
269     case NORTH_VELOCITY_ZOU_HE:
270     case SOUTH_VELOCITY_ZOU_HE:
271     case EAST_VELOCITY_ZOU_HE:
272     case WEST_VELOCITY_ZOU_HE:
273     case NORTH_PRESSURE_ZOU_HE:
274     case SOUTH_PRESSURE_ZOU_HE:
275     case EAST_PRESSURE_ZOU_HE:
276     case WEST_PRESSURE_ZOU_HE:
277     case NORTH_WEST_CONCAVE_CORNER_VELOCITY_ZOU_HE:
278     case NORTH_EAST_CONCAVE_CORNER_VELOCITY_ZOU_HE:
279     case SOUTH_WEST_CONCAVE_CORNER_VELOCITY_ZOU_HE:
280     case SOUTH_EAST_CONCAVE_CORNER_VELOCITY_ZOU_HE:
281     case NORTH_WEST_CONVEX_CORNER_VELOCITY_ZOU_HE:
282     case NORTH_EAST_CONVEX_CORNER_VELOCITY_ZOU_HE:
283     case SOUTH_WEST_CONVEX_CORNER_VELOCITY_ZOU_HE:
284     case SOUTH_EAST_CONVEX_CORNER_VELOCITY_ZOU_HE:
285     case EAST_OPEN_BOUNDARY:
286       for( int i = newBC.start_point.i; i <= newBC.end_point.i; i++ ){
287         for( int j = newBC.start_point.j; j <= newBC.end_point.j;
288               j++ ){
289           domain[ i ][ j ].isWet = true;
290           domain[ i ][ j ].isBC = true;
291         }
292       break;
293     case HALF WAY BOUNCEBACK:
294       for( int i = newBC.start_point.i; i <= newBC.end_point.i; i++ ){
295         for( int j = newBC.start_point.j; j <= newBC.end_point.j;
296               j++ ){
297           domain[ i ][ j ].isWet = true;
298           domain[ i ][ j ].isBC = true;
299         }
300       break;
301     case FULL WAY BOUNCEBACK:
302       for( int i = newBC.start_point.i; i <= newBC.end_point.i; i++ ){
303         for( int j = newBC.start_point.j; j <= newBC.end_point.j;
304               j++ ){
305           domain[ i ][ j ].isWet = false;
306           domain[ i ][ j ].isBC = true;
307         }
308       }
309   }
310 }
```

```

308         break;
309     }
310 }
311 void AddSolidBoxBoundaryConditionZouHe( Point startPoint, Point endPoint )
312 {
313     if ( startPoint.i >= endPoint.i || startPoint.j >= endPoint.j )
314         return;
315     AddBoundaryCondition( BoundaryCondition( NORTH_VELOCITY_ZOU_HE,
316         startPoint, Point(endPoint.i,startPoint.j), 0.0, 0.0, 0.0));
317     AddBoundaryCondition( BoundaryCondition( EAST_VELOCITY_ZOU_HE,
318         startPoint, Point(startPoint.i,endPoint.j), 0.0, 0.0, 0.0));
319     AddBoundaryCondition( BoundaryCondition( SOUTH_VELOCITY_ZOU_HE, Point(
320         startPoint.i,endPoint.j), endPoint, 0.0, 0.0, 0.0));
321     AddBoundaryCondition( BoundaryCondition( WEST_VELOCITY_ZOU_HE,
322         endPoint, Point(endPoint.i,startPoint.j), 0.0, 0.0, 0.0));
323     AddBoundaryCondition( BoundaryCondition(
324         SOUTH_WEST_CONVEX_CORNER_VELOCITY_ZOU_HE, startPoint, startPoint,
325         0.0, 0.0, 0.0));
326     AddBoundaryCondition( BoundaryCondition(
327         NORTH_EAST_CONVEX_CORNER_VELOCITY_ZOU_HE, endPoint, endPoint, 0.0,
328         0.0, 0.0));
329     AddBoundaryCondition( BoundaryCondition(
330         NORTH_WEST_CONVEX_CORNER_VELOCITY_ZOU_HE, Point(startPoint.i,
331         endPoint.j), Point(startPoint.i,endPoint.j), 0.0, 0.0, 0.0));
332     AddBoundaryCondition( BoundaryCondition(
333         SOUTH_EAST_CONVEX_CORNER_VELOCITY_ZOU_HE, Point(endPoint.i,
334         startPoint.j), Point(endPoint.i,startPoint.j), 0.0, 0.0, 0.0));
335     for( int i = startPoint.i + 1; i <= endPoint.i - 1; i++ ){
336         for( int j = startPoint.j + 1; j <= endPoint.j - 1; j++ ){
337             domain[ i ][ j ].isWet = false;
338             domain[ i ][ j ].isBC = false;
339         }
340     }
341 }
342 void AddSolidBoxBoundaryConditionBBFW( Point startPoint, Point endPoint )
343 {
344     if ( startPoint.i >= endPoint.i || startPoint.j >= endPoint.j )
345         return;
346     AddBoundaryCondition( BoundaryCondition( FULL WAY BOUNCEBACK,
347         startPoint, Point(endPoint.i,startPoint.j), 0.0, 0.0, 0.0));
348     AddBoundaryCondition( BoundaryCondition( FULL WAY BOUNCEBACK,
349         startPoint, Point(startPoint.i,endPoint.j), 0.0, 0.0, 0.0));
350     AddBoundaryCondition( BoundaryCondition( FULL WAY BOUNCEBACK, Point(
351         startPoint.i,endPoint.j), endPoint, 0.0, 0.0, 0.0));
352     AddBoundaryCondition( BoundaryCondition( FULL WAY BOUNCEBACK, endPoint
353         , Point(endPoint.i,startPoint.j), 0.0, 0.0, 0.0));
354     for( int i = startPoint.i + 1; i <= endPoint.i - 1; i++ ){
355         for( int j = startPoint.j + 1; j <= endPoint.j - 1; j++ ){
356             domain[ i ][ j ].isWet = false;
357             domain[ i ][ j ].isBC = false;
358         }
359     }
360 }
361 void SetLatticeInsideBody( Point cord ) {
362     domain[ cord.i ][ cord.j ].isWet = false;
363     domain[ cord.i ][ cord.j ].isBC = false;
364 }
365 double Tick( ){
366     computation_time.Start();
367     MacroscopicProperties();
368     CalculateTotalProperties();
369     CollisionStep();

```

```

352     CalculateForce( startPointForceEvaluation , endPointForceEvaluation ,
353                     Fx1, Fy1);
354     StreamStep();
355     CalculateForce( startPointForceEvaluation , endPointForceEvaluation ,
356                     Fx2, Fy2);
357     BoundaryConditions();
358     CalculateForce( startPointForceEvaluation , endPointForceEvaluation ,
359                     Fx3, Fy3);
360     time++;
361     return computation_time.Stop();
362 }
363 Lattice **domain;
364 LatticeModel *lattice_model;
365 int Nx, Ny;
366 int time;
367 double omega;
368 vector<BoundaryCondition> bc;
369 Point startPointForceEvaluation , endPointForceEvaluation;
370 double Fx1, Fy1;
371 double Fx2, Fy2;
372 double Fx3, Fy3;
373 double total_density;
374 double total_density_u;
375 double total_density_v;
376 double convergence_u;
377 double convergence_v;
378 double convergence_rho;
379 ComputationTime computation_time;
380 bool IndiceLogico( int n, int min, int max){
381     if( n >= min && n <= max ) return true;
382     else return false;
383 }
384 void CalculateTotalProperties( void ){
385     total_density = total_density_u = total_density_v = 0.0;
386     for( int i = 0; i < Nx; i++ ){
387         for( int j = 0; j < Ny; j++ ){
388             if( domain[ i ][ j ].isWet == true ){
389                 total_density += domain[ i ][ j ].rho;
390                 total_density_u += domain[ i ][ j ].rho*domain[ i ][ j ].u;
391                 total_density_v += domain[ i ][ j ].rho*domain[ i ][ j ].v;
392             }
393         }
394     }
395 }
396 void ApplySimpleBoundaryCondition ( BoundaryCondition bc ){
397     switch (bc.type){
398         case NORTH_VELOCITY_ZOU_HE:
399             for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
400                 for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
401                     bc.rho = domain[ i ][ j ].rho = (1./(1.+bc.v))*(domain[ i ][ j ].f[0]+domain[ i ][ j ].f[1]+domain[ i ][ j ].f[3]+2.*(
402                         domain[ i ][ j ].f[2]+domain[ i ][ j ].f[6]+domain[ i ][ j ].f[5]));
403                     domain[ i ][ j ].f[4] = domain[ i ][ j ].f[2] - (2./3.)*bc.rho*bc.v;
404                     domain[ i ][ j ].f[7] = domain[ i ][ j ].f[5] + (1./2.)*((
405                         domain[ i ][ j ].f[1]-domain[ i ][ j ].f[3]) - (1./2.)*bc.rho*bc.u - (1./6.)*bc.rho*bc.v;
406                     domain[ i ][ j ].f[8] = domain[ i ][ j ].f[6] - (1./2.)*(
407                         domain[ i ][ j ].f[1]-domain[ i ][ j ].f[3]) + (1./2.)*bc.rho*bc.u - (1./6.)*bc.rho*bc.v;
408                 }
409             }
410         }
411     }

```

```

402 }
403 }
404
405 case SOUTH_VELOCITY_ZOU_HE:
406   for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
407     for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
408       bc.rho = domain[i][j].rho = (1./(1.-bc.v))*(domain[i][j].f[0]+domain[i][j].f[1]+domain[i][j].f[3]+2.*(
409         domain[i][j].f[4]+domain[i][j].f[7]+domain[i][j].f[8]));
410       domain[i][j].f[2] = domain[i][j].f[4] + (2./3.)*bc.rho*bc.v;
411       domain[i][j].f[5] = domain[i][j].f[7] - (1./2.)*(domain[i][j].f[1]-domain[i][j].f[3]) + (1./2.)*bc.rho*bc.u + (1./6.)*bc.rho*bc.v;
412       domain[i][j].f[6] = domain[i][j].f[8] + (1./2.)*(domain[i][j].f[1]-domain[i][j].f[3]) - (1./2.)*bc.rho*bc.u + (1./6.)*bc.rho*bc.v;
413     }
414   }
415   break;
416 case EAST_VELOCITY_ZOU_HE:
417   for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
418     for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
419       bc.rho = domain[i][j].rho = (1./(1.+bc.u))*(domain[i][j].f[0]+domain[i][j].f[2]+domain[i][j].f[4]+2.*(
420         domain[i][j].f[1]+domain[i][j].f[5]+domain[i][j].f[8]));
421       domain[i][j].f[3] = domain[i][j].f[1] - (2./3.)*bc.rho*bc.u;
422       domain[i][j].f[7] = domain[i][j].f[5] + (1./2.)*(domain[i][j].f[2]-domain[i][j].f[4]) - (1./6.)*bc.rho*bc.u - (1./2.)*bc.rho*bc.v;
423       domain[i][j].f[6] = domain[i][j].f[8] - (1./2.)*(domain[i][j].f[2]-domain[i][j].f[4]) - (1./6.)*bc.rho*bc.u + (1./2.)*bc.rho*bc.v;
424     }
425   }
426   break;
427 case WEST_VELOCITY_ZOU_HE:
428   for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
429     for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
430       bc.rho = domain[i][j].rho = (1./(1.-bc.u))*(domain[i][j].f[0]+domain[i][j].f[2]+domain[i][j].f[4]+2.*(
431         domain[i][j].f[3]+domain[i][j].f[6]+domain[i][j].f[7]));
432       domain[i][j].f[1] = domain[i][j].f[3] + (2./3.)*bc.rho*bc.u;
433       domain[i][j].f[5] = domain[i][j].f[7] - (1./2.)*(domain[i][j].f[2]-domain[i][j].f[4]) + (1./6.)*bc.rho*bc.u + (1./2.)*bc.rho*bc.v;
434       domain[i][j].f[8] = domain[i][j].f[6] + (1./2.)*(domain[i][j].f[2]-domain[i][j].f[4]) + (1./6.)*bc.rho*bc.u - (1./2.)*bc.rho*bc.v;
435     }
436   }
437   break;
438 case NORTH_PRESSURE_ZOU_HE:
439   for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
440     for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
441       bc.v = domain[i][j].v = -1. + (1./bc.rho)*(domain[i][j].f[0]+domain[i][j].f[1]+domain[i][j].f[3]+2.*(
442         domain[i][j].f[2]+domain[i][j].f[6]+domain[i][j].f[8]));
443     }
444   }

```

```

439         [5]);
440         domain[i][j].f[4] = domain[i][j].f[2] - (2./3.)*bc.rho
441             *bc.v;
442         domain[i][j].f[7] = domain[i][j].f[5] + (1./2.)*(
443             domain[i][j].f[1]-domain[i][j].f[3]) - (1./2.)*bc.
444             rho*bc.u - (1./6.)*bc.rho*bc.v;
445         domain[i][j].f[8] = domain[i][j].f[6] - (1./2.)*(
446             domain[i][j].f[1]-domain[i][j].f[3]) + (1./2.)*bc.
447             rho*bc.u - (1./6.)*bc.rho*bc.v;
448     }
449     break;
450 case SOUTH_PRESSURE_ZOU_HE:
451     for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
452         for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
453             bc.v = domain[i][j].v = 1. - (1./bc.rho)*(domain[i][j]
454                 .f[0]+domain[i][j].f[1]+domain[i][j].f[3]+2.*(
455                     domain[i][j].f[4]+domain[i][j].f[7]+domain[i][j].f
456                     [8]));
457             domain[i][j].f[2] = domain[i][j].f[4] + (2./3.)*bc.rho
458                 *bc.v;
459             domain[i][j].f[5] = domain[i][j].f[7] - (1./2.)*(
460                 domain[i][j].f[1]-domain[i][j].f[3]) + (1./2.)*bc.
461                 rho*bc.u + (1./6.)*bc.rho*bc.v;
462             domain[i][j].f[6] = domain[i][j].f[8] + (1./2.)*(
463                 domain[i][j].f[1]-domain[i][j].f[3]) - (1./2.)*bc.
464                 rho*bc.u + (1./6.)*bc.rho*bc.v;
465         }
466     }
467     break;
468 case EAST_PRESSURE_ZOU_HE:
469     for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
470         for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
471             bc.u = domain[i][j].u = -1. + (1./bc.rho)*(domain[i][j]
472                 .f[0]+domain[i][j].f[2]+domain[i][j].f[4]+2.*(
473                     domain[i][j].f[1]+domain[i][j].f[5]+domain[i][j].f
474                     [8]));
475             domain[i][j].f[3] = domain[i][j].f[1] - (2./3.)*bc.rho
476                 *bc.u;
477             domain[i][j].f[7] = domain[i][j].f[5] + (1./2.)*(
478                 domain[i][j].f[2]-domain[i][j].f[4]) - (1./6.)*bc.
479                 rho*bc.u - (1./2.)*bc.rho*bc.v;
480             domain[i][j].f[6] = domain[i][j].f[8] - (1./2.)*(
481                 domain[i][j].f[2]-domain[i][j].f[4]) - (1./6.)*bc.
482                 rho*bc.u + (1./2.)*bc.rho*bc.v;
483         }
484     }
485     break;
486 case WEST_PRESSURE_ZOU_HE:
487     for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
488         for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
489             bc.u = domain[i][j].u = 1. - (1./bc.rho)*(domain[i][j]
490                 .f[0]+domain[i][j].f[2]+domain[i][j].f[4]+2.*(
491                     domain[i][j].f[3]+domain[i][j].f[6]+domain[i][j].f
492                     [7]));
493             domain[i][j].f[1] = domain[i][j].f[3] + (2./3.)*bc.rho
494                 *bc.u;
495             domain[i][j].f[5] = domain[i][j].f[7] - (1./2.)*(
496                 domain[i][j].f[2]-domain[i][j].f[4]) + (1./6.)*bc.
497                 rho*bc.u + (1./2.)*bc.rho*bc.v;
498             domain[i][j].f[8] = domain[i][j].f[6] + (1./2.)*(
499                 domain[i][j].f[2]-domain[i][j].f[4]) + (1./6.)*bc.
500                 rho*bc.u - (1./2.)*bc.rho*bc.v;
501         }
502     }
503 
```

```

    }
}
break;
case NORTH_WEST_CONCAVE_CORNER_VELOCITY_ZOU_HE:
for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
    for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
        domain[i][j].f[1] = domain[i][j].f[3];
        domain[i][j].f[4] = domain[i][j].f[2];
        domain[i][j].f[8] = domain[i][j].f[6];
        domain[i][j].f[5] = domain[i][j].f[7] = 0.5*(0.5*(
            domain[i+1][j].rho+domain[i][j-1].rho)-(domain[i][
                j].f[0]+domain[i][j].f[1]+domain[i][j].f[2]+domain[
                    i][j].f[3]+domain[i][j].f[4]+domain[i][j].f[6]+
                    domain[i][j].f[8]));
    }
}
break;
case NORTH_EAST_CONCAVE_CORNER_VELOCITY_ZOU_HE:
for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
    for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
        domain[i][j].f[3] = domain[i][j].f[1];
        domain[i][j].f[4] = domain[i][j].f[2];
        domain[i][j].f[7] = domain[i][j].f[5];
        domain[i][j].f[6] = domain[i][j].f[8] = (1./2.)
            *((1./2.)*(domain[i-1][j].rho+domain[i][j-1].rho)
            -(domain[i][j].f[0]+domain[i][j].f[1]+domain[i][j].
                f[2]+domain[i][j].f[3]+domain[i][j].f[4]+domain[
                    i][j].f[5]+domain[i][j].f[7]));
    }
}
break;
case SOUTH_WEST_CONCAVE_CORNER_VELOCITY_ZOU_HE:
for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
    for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
        domain[i][j].f[5] = domain[i][j].f[7];
        domain[i][j].f[2] = domain[i][j].f[4];
        domain[i][j].f[1] = domain[i][j].f[3];
        domain[i][j].f[6] = domain[i][j].f[8] = 0.5*(0.5*(
            domain[i+1][j].rho+domain[i][j+1].rho)-(domain[i][
                j].f[0]+domain[i][j].f[1]+domain[i][j].f[2]+domain[
                    i][j].f[3]+domain[i][j].f[4]+domain[i][j].f[5]+
                    domain[i][j].f[7]));
    }
}
break;
case SOUTH_EAST_CONCAVE_CORNER_VELOCITY_ZOU_HE:
for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
    for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
        domain[i][j].f[3] = domain[i][j].f[1];
        domain[i][j].f[2] = domain[i][j].f[4];
        domain[i][j].f[6] = domain[i][j].f[8];
        domain[i][j].f[5] = domain[i][j].f[7] = 0.5*(0.5*(
            domain[i][j+1].rho+domain[i-1][j].rho)-(domain[i][
                j].f[0]+domain[i][j].f[1]+domain[i][j].f[2]+domain[
                    i][j].f[3]+domain[i][j].f[4]+domain[i][j].f[6]+
                    domain[i][j].f[8]));
    }
}
break;
case NORTH_WEST_CONVEX_CORNER_VELOCITY_ZOU_HE:
for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
    for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
        domain[i][j].f[3] = domain[i][j].f[1];

```

```

519         domain[i][j].f[2] = domain[i][j].f[4];
520         domain[i][j].f[6] = domain[i][j].f[8];
521     }
522 }
523 break;
524 case NORTH_EAST_CONVEX_CORNER_VELOCITY_ZOU_HE:
525     for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
526         for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
527             domain[i][j].f[1] = domain[i][j].f[3];
528             domain[i][j].f[2] = domain[i][j].f[4];
529             domain[i][j].f[5] = domain[i][j].f[7];
530         }
531     }
532 break;
533 case SOUTH_WEST_CONVEX_CORNER_VELOCITY_ZOU_HE:
534     for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
535         for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
536             domain[i][j].f[4] = domain[i][j].f[2];
537             domain[i][j].f[3] = domain[i][j].f[1];
538             domain[i][j].f[7] = domain[i][j].f[5];
539         }
540     }
541 break;
542 case SOUTH_EAST_CONVEX_CORNER_VELOCITY_ZOU_HE:
543     for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
544         for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
545             domain[i][j].f[1] = domain[i][j].f[3];
546             domain[i][j].f[8] = domain[i][j].f[6];
547             domain[i][j].f[4] = domain[i][j].f[2];
548         }
549     }
550 break;
551 case HALF WAY_BOUNCEBACK:
552     break;
553 case FULL WAY_BOUNCEBACK:
554     for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
555         for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
556             swap( domain[i][j].f[1], domain[i][j].f[3] );
557             swap( domain[i][j].f[2], domain[i][j].f[4] );
558             swap( domain[i][j].f[5], domain[i][j].f[7] );
559             swap( domain[i][j].f[6], domain[i][j].f[8] );
560         }
561     }
562 break;
563 case EAST_OPEN_BOUNDARY:
564     for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
565         for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
566             domain[i][j].f[3] = domain[i-1][j].f[3];
567             domain[i][j].f[6] = domain[i-1][j].f[6];
568             domain[i][j].f[7] = domain[i-1][j].f[7];
569         }
570     }
571 break;
572 }
573 void CalculateForce( Point startPoint, Point endPoint, double& Fx, double&
574 Fy) {
575     if ( startPoint.i >= endPoint.i || startPoint.j >= endPoint.j )
576         return;
577     int ii , jj ;
578     Fx = 0.0;
579     Fy = 0.0;
580     for( int i = startPoint.i; i <= endPoint.i; i++ ){

```

```

581     for( int j = startPoint.j; j <= endPoint.j; j++ ){
582         if( domain[ i ][ j ].isBC == true ){
583             for( int alfa = 1; alfa < lattice_model->Q; alfa++ ){
584                 ii = i + lattice_model->e_x[ lattice_model->opp[ alfa
585 ] ];
586                 jj = j + lattice_model->e_y[ lattice_model->opp[ alfa
587 ] ];
588                 if( domain[ ii ][ jj ].isWet == true ){
589                     Fx += lattice_model->e_x[ alfa ]*(domain[ i ][ j
590 ].f[ alfa ] + domain[ ii ][ jj ].f[
591 lattice_model->opp[ alfa ] ]);
592                     Fy += lattice_model->e_y[ alfa ]*(domain[ i ][ j
593 ].f[ alfa ] + domain[ ii ][ jj ].f[
594 lattice_model->opp[ alfa ] );
595                 }
596             }
597         }
598     }
599 }
600 #endif

```

LBM1.h

```

1 #ifndef LBM_H
2 #define LBM_H
3 #include <vector>
4 class ComputationTime{
5     clock_t start;
6 public:
7     void Start(){
8         start = clock();
9     }
10    double Stop(){
11        return (((double) (clock() - start)) / CLOCKS_PER_SEC);
12    }
13};
14 void Swap( double &A, double &B){
15     double tmp = A;
16     A = B;
17     B = tmp;
18}
19 class LatticeModel{
20 public:
21     LatticeModel( int D, int Q, double *w, int *e_x, int *e_y, int *e_z, int *
22     opp, double cs){
23         this->D = D;
24         this->Q = Q;
25         this->cs = cs;
26         this->w = new double [ Q ];
27         this->e_x = new int [ Q ];
28         this->e_y = new int [ Q ];
29         this->opp = new int [ Q ];
30         for( int i = 0; i < Q; i++ ){
31             this->w[ i ] = w[ i ];
32             this->opp[ i ] = opp[ i ];
33             switch ( D ){
34                 case 1:
35                     this->e_x[ i ] = e_x[ i ];
36                     break;
37                 case 2:
38                     this->e_x[ i ] = e_x[ i ];
39             }
40         }
41     }
42 };

```

```

38         this->e_y[ i ] = e_y[ i ];
39         break;
40     case 3:
41         this->e_x[ i ] = e_x[ i ];
42         this->e_y[ i ] = e_y[ i ];
43         this->e_z[ i ] = e_z[ i ];
44         break;
45     }
46 }
47 double *w;
48 int *e_x;
49 int *e_y;
50 int *e_z;
51 int *opp;
52 double cs;
53 int D;
54 int Q;
55 };
56 class Lattice {
57 public:
58     Lattice (){
59     }
60     void SetLatticeModel ( LatticeModel *lattice_model_momentum , LatticeModel
61     *lattice_model_energy){
62         this->lattice_model_momentum = lattice_model_momentum;
63         this->lattice_model_energy = lattice_model_energy;
64         this->f = new double [ lattice_model_momentum->Q ];
65         this->g = new double [ lattice_model_energy->Q ];
66         isWet = true;
67         isBC = false;
68     }
69     inline double CalculateMacroscopicProperties ( void ){
70         rho = 0.0;
71         u = 0.0;
72         v = 0.0;
73         T = 0.0;
74         for( int i = 0; i < lattice_model_momentum->Q; i++ ){
75             rho += f[ i ];
76             u += f[ i ] * lattice_model_momentum->e_x[ i ];
77             v += f[ i ] * lattice_model_momentum->e_y[ i ];
78         }
79         u /= rho;
80         v /= rho;
81         for( int i = 0; i < lattice_model_energy->Q; i++ ){
82             T += g[ i ];
83         }
84     }
85     double *f;
86     double *g;
87     double rho;
88     double u;
89     double v;
90     double T;
91     LatticeModel *lattice_model_momentum;
92     LatticeModel *lattice_model_energy;
93     bool isWet;
94     bool isBC;
95 };
96 class Point {
97 public:
98     Point (){
99         i = j = 0;

```

```

100    }
101    Point (const int i, const int j){
102        this->i = i;
103        this->j = j;
104    }
105    int i, j;
106};
107 enum BC_TYPE { NORTH_VELOCITY_ZOU_HE, SOUTH_VELOCITY_ZOU_HE,
108                 EAST_VELOCITY_ZOU_HE, WEST_VELOCITY_ZOU_HE, NORTH_PRESSURE_ZOU_HE,
109                 SOUTH_PRESSURE_ZOU_HE, EAST_PRESSURE_ZOU_HE, WEST_PRESSURE_ZOU_HE,
110                 NORTH_WEST_CONVEX_CORNER_VELOCITY_ZOU_HE,
111                 NORTH_EAST_CONVEX_CORNER_VELOCITY_ZOU_HE,
112                 SOUTH_WEST_CONVEX_CORNER_VELOCITY_ZOU_HE,
113                 SOUTH_EAST_CONVEX_CORNER_VELOCITY_ZOU_HE,
114                 NORTH_WEST_CONCAVE_CORNER_VELOCITY_ZOU_HE,
115                 NORTH_EAST_CONCAVE_CORNER_VELOCITY_ZOU_HE,
116                 SOUTH_WEST_CONCAVE_CORNER_VELOCITY_ZOU_HE,
117                 SOUTH_EAST_CONCAVE_CORNER_VELOCITY_ZOU_HE, HALF_WAY_BOUNCEBACK,
118                 FULL_WAY_BOUNCEBACK, EAST_OPEN_BOUNDARY, EAST_TEMPERATURE,
119                 WEST_TEMPERATURE, ADIABATIC, NORTH_ADIABATIC, SOUTH_ADIABATIC};
120 class BoundaryCondition {
121 public:
122     BoundaryCondition( BC_TYPE type, Point start_point, Point end_point,
123                         double u, double v, double rho, double T){
124         this->type = type;
125         this->start_point = start_point;
126         this->end_point = end_point;
127         this->u = u;
128         this->v = v;
129         this->rho = rho;
130         this->T = T;
131     }
132     BC_TYPE type;
133     Point start_point;
134     Point end_point;
135     double u;
136     double v;
137     double rho;
138     double T;
139 };
140 class Simulation{
141 public:
142     Simulation( int Nx, int Ny, LatticeModel *lattice_model_momentum,
143                 LatticeModel *lattice_model_energy, double omega_momentum, double
144                 omega_energy, double gravitybeta){
145         domain = new Lattice*[ Nx ];
146         this->Nx = Nx;
147         this->Ny = Ny;
148         this->lattice_model_momentum = lattice_model_momentum;
149         this->lattice_model_energy = lattice_model_energy;
150         time = 0;
151         this->omega_momentum = omega_momentum;
152         this->omega_energy = omega_energy;
153         this->gravitybeta = gravitybeta;
154         for (int i = 0; i < Nx; i++){
155             domain[ i ] = new Lattice[ Ny ];
156         }
157         for( int i = 0; i < Nx; i++ ){
158             for( int j = 0; j < Ny; j++ ){
159                 domain[ i ][ j ].SetLatticeModel( lattice_model_momentum,
160                     lattice_model_energy );
161             }
162         }
163     }

```

```

147     startPointForceEvaluation.i = startPointForceEvaluation.j =
148     endPointForceEvaluation.i = endPointForceEvaluation.j = 0;
149 }
150 void InitEquilibrium( double density, double temperature ) {
151     for( int i = 0; i < Nx; i++ ){
152         for( int j = 0; j < Ny; j++ ){
153             for( int alfa = 0; alfa < lattice_model_momentum->Q; alfa++ ){
154                 domain[ i ][ j ].f[ alfa ] = lattice_model_momentum->w[
155                     alfa ] * density;
156             }
157             domain[ i ][ j ].rho = density;
158             for( int alfa = 0; alfa < lattice_model_energy->Q; alfa++ ){
159                 domain[ i ][ j ].g[ alfa ] = lattice_model_energy->w[ alfa
160                     ] * temperature;
161             }
162         }
163     }
164     void StreamStep( ){
165         int ii , jj ;
166         int alfa ;
167         for( int j = 0; j < Ny; j++ ){
168             for( int i = Nx-1; i >= 0; i-- ){
169                 if( !(domain[ i ][ j ].isBC == false && domain[ i ][ j ].isWet
170                     == false) ){
171                     alfa = 1;
172                     ii = i + lattice_model_momentum->e_x[
173                         lattice_model_momentum->opp[ alfa ] ];
174                     jj = j + lattice_model_momentum->e_y[
175                         lattice_model_momentum->opp[ alfa ] ];
176                     if( IndiceLogico( ii , 0, Nx-1) && IndiceLogico( jj , 0, Ny
177                         -1)){
178                         domain[ i ][ j ].f[ alfa ] = domain[ ii ][ jj ].f[ alfa
179                         ];
180                         domain[ i ][ j ].g[ alfa ] = domain[ ii ][ jj ].g[ alfa
181                         ];
182                     }
183                 }
184             }
185         }
186     }
187 }
188 }
189 }
190 for( int j = 0; j < Ny; j++ ){
191     for( int i = 0; i < Nx; i++ ){
192         if( !(domain[ i ][ j ].isBC == false && domain[ i ][ j ].isWet
193             == false) ){
194             alfa = 4;

```

```

194     ii = i + lattice_model_momentum->e_x[  

195         lattice_model_momentum->opp[ alfa ] ];  

196     jj = j + lattice_model_momentum->e_y[  

197         lattice_model_momentum->opp[ alfa ] ];  

198     if( IndiceLogico( ii , 0, Nx-1) && IndiceLogico( jj , 0, Ny  

199         -1)){  

200         domain[ i ][ j ].f[ alfa ] = domain[ ii ][ jj ].f[  

201             alfa ];  

202         domain[ i ][ j ].g[ alfa ] = domain[ ii ][ jj ].g[  

203             alfa ];  

204     }  

205     alfa = 7;  

206     ii = i + lattice_model_momentum->e_x[  

207         lattice_model_momentum->opp[ alfa ] ];  

208     jj = j + lattice_model_momentum->e_y[  

209         lattice_model_momentum->opp[ alfa ] ];  

210     if( IndiceLogico( ii , 0, Nx-1) && IndiceLogico( jj , 0, Ny  

211         -1)){  

212         domain[ i ][ j ].f[ alfa ] = domain[ ii ][ jj ].f[  

213             alfa ];  

214         domain[ i ][ j ].g[ alfa ] = domain[ ii ][ jj ].g[  

215             alfa ];  

216     }  

217     }  

218   }  

219 }  

220 for( int i = Nx-1; i >= 0; i-- ){  

221   if( !(domain[ i ][ j ].isBC == false && domain[ i ][ j ].isWet  

222       == false) ){  

223     alfa = 8;  

224     ii = i + lattice_model_momentum->e_x[  

225         lattice_model_momentum->opp[ alfa ] ];  

226     jj = j + lattice_model_momentum->e_y[  

227         lattice_model_momentum->opp[ alfa ] ];  

228     if( IndiceLogico( ii , 0, Nx-1) && IndiceLogico( jj , 0, Ny  

229         -1)){  

230         domain[ i ][ j ].f[ alfa ] = domain[ ii ][ jj ].f[  

231             alfa ];  

232         domain[ i ][ j ].g[ alfa ] = domain[ ii ][ jj ].g[  

233             alfa ];  

234     }  

235     alfa = 2;  

236     ii = i + lattice_model_momentum->e_x[  

237         lattice_model_momentum->opp[ alfa ] ];  

238     jj = j + lattice_model_momentum->e_y[  

239         lattice_model_momentum->opp[ alfa ] ];  

240     if( IndiceLogico( ii , 0, Nx-1) && IndiceLogico( jj , 0, Ny  

241         -1)){  

242         domain[ i ][ j ].f[ alfa ] = domain[ ii ][ jj ].f[  

243             alfa ];  

244         domain[ i ][ j ].g[ alfa ] = domain[ ii ][ jj ].g[  

245             alfa ];  

246     }  

247     alfa = 6;  

248     ii = i + lattice_model_momentum->e_x[  

249         lattice_model_momentum->opp[ alfa ] ];

```

```

233     jj = j + lattice_model_momentum->e_y[
234         lattice_model_momentum->opp[ alfa ] ];
235     if( IndiceLogico( ii , 0, Nx-1) && IndiceLogico( jj , 0, Ny
236         -1)){
237         domain[ i ][ j ].f[ alfa ] = domain[ ii ][ jj ].f[
238             alfa ];
239         domain[ i ][ j ].g[ alfa ] = domain[ ii ][ jj ].g[
240             alfa ];
241     }
242 }
243 for( int i = Nx-1; i >= 0; i-- ){
244     if( !(domain[ i ][ j ].isBC == false && domain[ i ][ j ].isWet
245         == false ) ){
246         alfa = 5;
247         ii = i + lattice_model_momentum->e_x[
248             lattice_model_momentum->opp[ alfa ] ];
249         jj = j + lattice_model_momentum->e_y[
250             lattice_model_momentum->opp[ alfa ] ];
251         if( IndiceLogico( ii , 0, Nx-1) && IndiceLogico( jj , 0, Ny
252             -1)){
253             domain[ i ][ j ].f[ alfa ] = domain[ ii ][ jj ].f[
254                 alfa ];
255             domain[ i ][ j ].g[ alfa ] = domain[ ii ][ jj ].g[
256                 alfa ];
257         }
258     }
259 }
260 }
261 void CollisionStep( ){
262     double feq , geq;
263     for( int i = 0; i < Nx; i++ ){
264         for( int j = 0; j < Ny; j++ ){
265             if( domain[ i ][ j ].isWet == true ){
266                 for( int alfa = 0; alfa < lattice_model_momentum->Q; alfa
267                     ++ ){
268                     feq = lattice_model_momentum->w[ alfa ]*domain[ i ][ j ].rho
269                         *(1 + 3.0*(lattice_model_momentum->e_x[ alfa ]*
270                             domain[ i ][ j ].u + lattice_model_momentum->e_y[ alfa
271                             ]*domain[ i ][ j ].v) + (9.0/2.0)*pow(
272                             lattice_model_momentum->e_x[ alfa ]*domain[ i ][ j ].u +
273                             lattice_model_momentum->e_y[ alfa ]*domain[ i ][ j ].v
274                             ,2.0) -(3.0/2.0)*(domain[ i ][ j ].u*domain[ i ][ j ].u +
275                             domain[ i ][ j ].v*domain[ i ][ j ].v));
276                     domain[ i ][ j ].f[ alfa ] = domain[ i ][ j ].f[ alfa ]
277                         - omega_momentum * ( domain[ i ][ j ].f[ alfa ]
278                             - feq )
279                         + (lattice_model_momentum->w[ alfa ]/pow(
280                             lattice_model_momentum->cs ,2.0)) * (
281                             gravitybeta * domain[ i ][ j ].rho *
282                             domain[ i ][ j ].T *
283                             lattice_model_momentum->e_y[ alfa ]);
284                 }
285             }
286             for( int alfa = 0; alfa < lattice_model_energy->Q; alfa++
287                 ){
288                 geq = lattice_model_energy->w[ alfa ]*domain[ i ][ j ].T*(1
289                     + 3.0*(lattice_model_energy->e_x[ alfa ]*domain[ i ][ j
290                         ].u + lattice_model_energy->e_y[ alfa ]*domain[ i ][ j
291                         ].v) +(9.0/2.0)*pow(lattice_model_energy->e_x[ alfa
292                             ]*domain[ i ][ j ].u + lattice_model_energy->e_y[ alfa
293                             ]*domain[ i ][ j ].v,2.0) -(3.0/2.0)*(domain[ i ][ j ].u*
294                             domain[ i ][ j ].u + domain[ i ][ j ].v*domain[ i ][ j ].v));
295             }
296         }
297     }
298 }
```

```

265         domain[ i ][ j ].g[ alfa ] = domain[ i ][ j ].g[ alfa ]
266         [ ] - omega_energy * ( domain[ i ][ j ].g[ alfa ] -
267           geq );
268     }
269   }
270 }
271 void MacroscopicProperties( ){
272   double old_u, old_v, old_rho, old_T;
273   double error_u, error_v, error_rho, error_T;
274   convergence_u = convergence_v = convergence_rho = convergence_T = 0.0;
275   for( int i = 0; i < Nx; i++ ){
276     for( int j = 0; j < Ny; j++ ){
277       if( domain[ i ][ j ].isWet == true ){
278         old_u = domain[ i ][ j ].u;
279         old_v = domain[ i ][ j ].v;
280         old_rho = domain[ i ][ j ].rho;
281         old_T = domain[ i ][ j ].T;
282         domain[ i ][ j ].CalculateMacroscopicProperties();
283         error_u = fabs( old_u - domain[ i ][ j ].u );
284         error_v = fabs( old_v - domain[ i ][ j ].v );
285         error_rho = fabs( old_rho - domain[ i ][ j ].rho );
286         error_T = fabs( old_T - domain[ i ][ j ].T );
287         if( error_u > convergence_u ) convergence_u = error_u;
288         if( error_v > convergence_v ) convergence_v = error_v;
289         if( error_rho > convergence_rho ) convergence_rho =
290           error_rho;
291         if( error_T > convergence_T ) convergence_T = error_T;
292       }
293     }
294   }
295   void BoundaryConditions( ){
296     for( int i = 0; i < bc.size(); i++ ){
297       ApplySimpleBoundaryCondition( bc[ i ] );
298     }
299   }
300   void AddBoundaryCondition( BoundaryCondition newBC ) {
301     bc.push_back( newBC );
302     switch( newBC.type ){
303       case NORTH_VELOCITY_ZOU_HE:
304       case SOUTH_VELOCITY_ZOU_HE:
305       case EAST_VELOCITY_ZOU_HE:
306       case WEST_VELOCITY_ZOU_HE:
307       case NORTH_PRESSURE_ZOU_HE:
308       case SOUTH_PRESSURE_ZOU_HE:
309       case EAST_PRESSURE_ZOU_HE:
310       case WEST_PRESSURE_ZOU_HE:
311       case NORTH_WEST_CONCAVE_CORNER_VELOCITY_ZOU_HE:
312       case NORTH_EAST_CONCAVE_CORNER_VELOCITY_ZOU_HE:
313       case SOUTH_WEST_CONCAVE_CORNER_VELOCITY_ZOU_HE:
314       case SOUTH_EAST_CONCAVE_CORNER_VELOCITY_ZOU_HE:
315       case NORTH_WEST_CONVEX_CORNER_VELOCITY_ZOU_HE:
316       case NORTH_EAST_CONVEX_CORNER_VELOCITY_ZOU_HE:
317       case SOUTH_WEST_CONVEX_CORNER_VELOCITY_ZOU_HE:
318       case SOUTH_EAST_CONVEX_CORNER_VELOCITY_ZOU_HE:
319       case EAST_OPEN_BOUNDARY:
320       case EAST_TEMPERATURE:
321       case WEST_TEMPERATURE:
322       case NORTH_ADIABATIC:
323       case SOUTH_ADIABATIC:

```

```

324     for( int i = newBC.start_point.i; i <= newBC.end_point.i; i++ ){
325         for( int j = newBC.start_point.j; j <= newBC.end_point.j; j++ ){
326             domain[ i ][ j ].isWet = true;
327             domain[ i ][ j ].isBC = true;
328         }
329     }
330     break;
331 case HALF_WAY_BOUNCEBACK:
332     for( int i = newBC.start_point.i; i <= newBC.end_point.i; i++ ){
333         for( int j = newBC.start_point.j; j <= newBC.end_point.j; j++ ){
334             domain[ i ][ j ].isWet = true;
335             domain[ i ][ j ].isBC = true;
336         }
337     }
338     break;
339 case FULL_WAY_BOUNCEBACK:
340 case ADIABATIC:
341     for( int i = newBC.start_point.i; i <= newBC.end_point.i; i++ ){
342         for( int j = newBC.start_point.j; j <= newBC.end_point.j; j++ ){
343             domain[ i ][ j ].isWet = false;
344             domain[ i ][ j ].isBC = true;
345         }
346     }
347     break;
348 }
349 }
350 void AddSolidBoxBoundaryConditionZouHe( Point startPoint, Point endPoint )
{
351     if( startPoint.i >= endPoint.i || startPoint.j >= endPoint.j )
352         return;
353     AddBoundaryCondition( BoundaryCondition( NORTH_VELOCITY_ZOU_HE,
354                                         startPoint, Point( endPoint.i, startPoint.j ), 0.0, 0.0, 0.0, 0.0 ) );
355     AddBoundaryCondition( BoundaryCondition( EAST_VELOCITY_ZOU_HE,
356                                         startPoint, Point( startPoint.i, endPoint.j ), 0.0, 0.0, 0.0, 0.0 ) );
357     AddBoundaryCondition( BoundaryCondition( SOUTH_VELOCITY_ZOU_HE, Point(
358                                         startPoint.i, endPoint.j ), endPoint, 0.0, 0.0, 0.0, 0.0 ) );
359     AddBoundaryCondition( BoundaryCondition( WEST_VELOCITY_ZOU_HE,
360                                         endPoint, Point( endPoint.i, startPoint.j ), 0.0, 0.0, 0.0, 0.0 ) );
361     AddBoundaryCondition( BoundaryCondition(
362                                         SOUTH_WEST_CONVEX_CORNER_VELOCITY_ZOU_HE, startPoint, startPoint,
363                                         0.0, 0.0, 0.0, 0.0 ) );
364     AddBoundaryCondition( BoundaryCondition(
365                                         NORTH_EAST_CONVEX_CORNER_VELOCITY_ZOU_HE, endPoint, endPoint,
366                                         0.0, 0.0, 0.0, 0.0 ) );
367     AddBoundaryCondition( BoundaryCondition(
368                                         NORTH_WEST_CONVEX_CORNER_VELOCITY_ZOU_HE, Point( startPoint.i,
369                                         endPoint.j ), Point( startPoint.i, endPoint.j ), 0.0, 0.0, 0.0, 0.0 ) );
370     AddBoundaryCondition( BoundaryCondition(
371                                         SOUTH_EAST_CONVEX_CORNER_VELOCITY_ZOU_HE, Point( endPoint.i,
372                                         startPoint.j ), Point( endPoint.i, startPoint.j ), 0.0, 0.0, 0.0, 0.0 ) );
373     for( int i = startPoint.i + 1; i <= endPoint.i - 1; i++ ){
374         for( int j = startPoint.j + 1; j <= endPoint.j - 1; j++ ){
375             domain[ i ][ j ].isWet = false;
376             domain[ i ][ j ].isBC = false;
377         }
378     }
379 }
```

```

367    }
368    void AddSolidBoxBoundaryConditionBBFW( Point startPoint , Point endPoint )
369    {
370        if ( startPoint.i >= endPoint.i || startPoint.j >= endPoint.j )
371            return ;
372        AddBoundaryCondition( BoundaryCondition( FULL_WAY_BOUNCEBACK,
373            startPoint , Point(endPoint.i,startPoint.j) , 0.0 , 0.0 , 0.0 , 0.0));
374        AddBoundaryCondition( BoundaryCondition( FULL_WAY_BOUNCEBACK,
375            startPoint , Point(startPoint.i,endPoint.j) , 0.0 , 0.0 , 0.0 , 0.0));
376        AddBoundaryCondition( BoundaryCondition( FULL_WAY_BOUNCEBACK, Point(
377            startPoint.i,endPoint.j) , endPoint , 0.0 , 0.0 , 0.0 , 0.0));
378        AddBoundaryCondition( BoundaryCondition( FULL_WAY_BOUNCEBACK, endPoint
379            , Point(endPoint.i,startPoint.j) , 0.0 , 0.0 , 0.0 , 0.0));
380        for( int i = startPoint.i + 1; i <= endPoint.i - 1; i++ ){
381            for( int j = startPoint.j + 1; j <= endPoint.j - 1; j++ ){
382                domain[ i ][ j ].isWet = false ;
383                domain[ i ][ j ].isBC = false ;
384            }
385        }
386    }
387    void SetLatticeInsideBody( Point cord ) {
388        domain[ cord.i ][ cord.j ].isWet = false ;
389        domain[ cord.i ][ cord.j ].isBC = false ;
390    }
391    double Tick( ){
392        computation_time.Start();
393        MacroscopicProperties();
394        CalculateTotalProperties();
395        CollisionStep();
396        StreamStep();
397        BoundaryConditions();
398        time++;
399        return computation_time.Stop();
400    }
401    Lattice **domain;
402    LatticeModel *lattice_model_momentum;
403    LatticeModel *lattice_model_energy;
404    int Nx, Ny;
405    int time;
406    double omega_momentum;
407    double omega_energy;
408    vector<BoundaryCondition> bc;
409    Point startPointForceEvaluation , endPointForceEvaluation;
410    double Fx1, Fy1;
411    double Fx2, Fy2;
412    double Fx3, Fy3;
413    double total_density;
414    double total_density_u;
415    double total_density_v;
416    double convergence_u;
417    double convergence_v;
418    double convergence_rho;
419    double convergence_T;
420    double gravitybeta;
421    ComputationTime computation_time;
422    bool IndiceLogico( int n, int min, int max){
423        if( n >= min && n <= max ) return true;
424        else return false;
425    }
426    void CalculateTotalProperties( void ){
427        total_density = total_density_u = total_density_v = 0.0;
428        for( int i = 0; i < Nx; i++ ){
429            for( int j = 0; j < Ny; j++ ){

```

```

425     if( domain[ i ][ j ].isWet == true ){
426         total_density += domain[ i ][ j ].rho;
427         total_density_u += domain[ i ][ j ].rho*domain[ i ][ j ].u
428             ;
429         total_density_v += domain[ i ][ j ].rho*domain[ i ][ j ].v
430             ;
431     }
432 }
433 void ApplySimpleBoundaryCondition ( BoundaryCondition bc ){
434     switch (bc.type){
435         case NORTH_VELOCITY_ZOU_HE:
436             for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
437                 for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
438                     bc.rho = domain[ i ][ j ].rho = (1./(1.+bc.v))*(domain[ i ][
439                         j ].f[0]+domain[ i ][ j ].f[1]+domain[ i ][ j ].f[3]+2.*(
440                             domain[ i ][ j ].f[2]+domain[ i ][ j ].f[6]+domain[ i ][ j ].f
441                             [5]));
442                     domain[ i ][ j ].f[4] = domain[ i ][ j ].f[2] - (2./3.)*bc.rho
443                         *bc.v;
444                     domain[ i ][ j ].f[7] = domain[ i ][ j ].f[5] + (1./2.)*(
445                         domain[ i ][ j ].f[1]-domain[ i ][ j ].f[3]) - (1./2.)*bc.
446                         rho*bc.u - (1./6.)*bc.rho*bc.v;
447                     domain[ i ][ j ].f[8] = domain[ i ][ j ].f[6] - (1./2.)*(
448                         domain[ i ][ j ].f[1]-domain[ i ][ j ].f[3]) + (1./2.)*bc.
449                         rho*bc.u - (1./6.)*bc.rho*bc.v;
450                 }
451             }
452             break;
453         case SOUTH_VELOCITY_ZOU_HE:
454             for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
455                 for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
456                     bc.rho = domain[ i ][ j ].rho = (1./(1.-bc.v))*(domain[ i ][
457                         j ].f[0]+domain[ i ][ j ].f[1]+domain[ i ][ j ].f[3]+2.*(
458                             domain[ i ][ j ].f[4]+domain[ i ][ j ].f[7]+domain[ i ][ j ].f
459                             [8]));
460                     domain[ i ][ j ].f[2] = domain[ i ][ j ].f[4] + (2./3.)*bc.rho
461                         *bc.v;
462                     domain[ i ][ j ].f[5] = domain[ i ][ j ].f[7] - (1./2.)*(
463                         domain[ i ][ j ].f[1]-domain[ i ][ j ].f[3]) + (1./2.)*bc.
464                         rho*bc.u + (1./6.)*bc.rho*bc.v;
465                     domain[ i ][ j ].f[6] = domain[ i ][ j ].f[8] + (1./2.)*(
466                         domain[ i ][ j ].f[1]-domain[ i ][ j ].f[3]) - (1./2.)*bc.
467                         rho*bc.u + (1./6.)*bc.rho*bc.v;
468                 }
469             }
470             break;
471         case EAST_VELOCITY_ZOU_HE:
472             for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
473                 for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
474                     bc.rho = domain[ i ][ j ].rho = (1./(1.+bc.u))*(domain[ i ][
475                         j ].f[0]+domain[ i ][ j ].f[2]+domain[ i ][ j ].f[4]+2.*(
476                             domain[ i ][ j ].f[1]+domain[ i ][ j ].f[5]+domain[ i ][ j ].f
477                             [8]));
478                     domain[ i ][ j ].f[3] = domain[ i ][ j ].f[1] - (2./3.)*bc.rho
479                         *bc.u;
480                     domain[ i ][ j ].f[7] = domain[ i ][ j ].f[5] + (1./2.)*(
481                         domain[ i ][ j ].f[2]-domain[ i ][ j ].f[4]) - (1./6.)*bc.
482                         rho*bc.u - (1./2.)*bc.rho*bc.v;
483                     domain[ i ][ j ].f[6] = domain[ i ][ j ].f[8] - (1./2.)*(
484                         domain[ i ][ j ].f[2]-domain[ i ][ j ].f[4]) - (1./6.)*bc.
485                         rho*bc.u + (1./2.)*bc.rho*bc.v;
486                 }
487             }
488         }
489     }
490 }
```

```

462 }
463 }
464 break;
465 case WEST_VELOCITY_ZOU_HE:
466 for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
467     for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
468         bc.rho = domain[i][j].rho = (1. / (1. - bc.u)) * (domain[i][j].f[0] + domain[i][j].f[2] + domain[i][j].f[4] + 2. * (domain[i][j].f[3] + domain[i][j].f[6] + domain[i][j].f[7]));
469         domain[i][j].f[1] = domain[i][j].f[3] + (2. / 3.) * bc.rho * bc.u;
470         domain[i][j].f[5] = domain[i][j].f[7] - (1. / 2.) * (domain[i][j].f[2] - domain[i][j].f[4]) + (1. / 6.) * bc.rho * bc.u + (1. / 2.) * bc.rho * bc.v;
471         domain[i][j].f[8] = domain[i][j].f[6] + (1. / 2.) * (domain[i][j].f[2] - domain[i][j].f[4]) + (1. / 6.) * bc.rho * bc.u - (1. / 2.) * bc.rho * bc.v;
472     }
473 }
474 break;
475 case NORTH_PRESSURE_ZOU_HE:
476 for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
477     for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
478         bc.v = domain[i][j].v = -1. + (1. / bc.rho) * (domain[i][j].f[0] + domain[i][j].f[1] + domain[i][j].f[3] + 2. * (domain[i][j].f[2] + domain[i][j].f[6] + domain[i][j].f[5]));
479         domain[i][j].f[4] = domain[i][j].f[2] - (2. / 3.) * bc.rho * bc.v;
480         domain[i][j].f[7] = domain[i][j].f[5] + (1. / 2.) * (domain[i][j].f[1] - domain[i][j].f[3]) - (1. / 2.) * bc.rho * bc.u - (1. / 6.) * bc.rho * bc.v;
481         domain[i][j].f[8] = domain[i][j].f[6] - (1. / 2.) * (domain[i][j].f[1] - domain[i][j].f[3]) + (1. / 2.) * bc.rho * bc.u - (1. / 6.) * bc.rho * bc.v;
482     }
483 }
484 break;
485 case SOUTH_PRESSURE_ZOU_HE:
486 for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
487     for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
488         bc.v = domain[i][j].v = 1. - (1. / bc.rho) * (domain[i][j].f[0] + domain[i][j].f[1] + domain[i][j].f[3] + 2. * (domain[i][j].f[4] + domain[i][j].f[7] + domain[i][j].f[8]));
489         domain[i][j].f[2] = domain[i][j].f[4] + (2. / 3.) * bc.rho * bc.v;
490         domain[i][j].f[5] = domain[i][j].f[7] - (1. / 2.) * (domain[i][j].f[1] - domain[i][j].f[3]) + (1. / 2.) * bc.rho * bc.u + (1. / 6.) * bc.rho * bc.v;
491         domain[i][j].f[6] = domain[i][j].f[8] + (1. / 2.) * (domain[i][j].f[1] - domain[i][j].f[3]) - (1. / 2.) * bc.rho * bc.u + (1. / 6.) * bc.rho * bc.v;
492     }
493 }
494 break;
495 case EAST_PRESSURE_ZOU_HE:
496 for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
497     for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
498         bc.u = domain[i][j].u = -1. + (1. / bc.rho) * (domain[i][j].f[0] + domain[i][j].f[2] + domain[i][j].f[4] + 2. * (domain[i][j].f[1] + domain[i][j].f[5] + domain[i][j].f[7]));

```

```

        [8])) ;
    domain[i][j].f[3] = domain[i][j].f[1] - (2./3.)*bc.rho
    *bc.u;
    domain[i][j].f[7] = domain[i][j].f[5] + (1./2.)*(
        domain[i][j].f[2]-domain[i][j].f[4]) - (1./6.)*bc.
        rho*bc.u - (1./2.)*bc.rho*bc.v;
    domain[i][j].f[6] = domain[i][j].f[8] - (1./2.)*(
        domain[i][j].f[2]-domain[i][j].f[4]) - (1./6.)*bc.
        rho*bc.u + (1./2.)*bc.rho*bc.v;
}
}
break;
case WEST_PRESSURE_ZOU_HE:
for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
    for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
        bc.u = domain[i][j].u = 1. - (1./bc.rho)*(domain[i][j].
            f[0]+domain[i][j].f[2]+domain[i][j].f[4]+2.*(
                domain[i][j].f[3]+domain[i][j].f[6]+domain[i][j].f
                [7]));
        domain[i][j].f[1] = domain[i][j].f[3] + (2./3.)*bc.rho
        *bc.u;
        domain[i][j].f[5] = domain[i][j].f[7] - (1./2.)*(
            domain[i][j].f[2]-domain[i][j].f[4]) + (1./6.)*bc.
            rho*bc.u + (1./2.)*bc.rho*bc.v;
        domain[i][j].f[8] = domain[i][j].f[6] + (1./2.)*(
            domain[i][j].f[2]-domain[i][j].f[4]) + (1./6.)*bc.
            rho*bc.u - (1./2.)*bc.rho*bc.v;
    }
}
break;
case NORTH_WEST_CONCAVE_CORNER_VELOCITY_ZOU_HE:
for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
    for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
        domain[i][j].f[1] = domain[i][j].f[3];
        domain[i][j].f[4] = domain[i][j].f[2];
        domain[i][j].f[8] = domain[i][j].f[6];
        domain[i][j].f[5] = domain[i][j].f[7] = 0.5*(0.5*(
            domain[i+1][j].rho+domain[i][j-1].rho)-(domain[i][j].
            f[0]+domain[i][j].f[1]+domain[i][j].f[2]+domain
            [i][j].f[3]+domain[i][j].f[4]+domain[i][j].f[6]+
            domain[i][j].f[8]));
    }
}
break;
case NORTH_EAST_CONCAVE_CORNER_VELOCITY_ZOU_HE:
for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
    for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
        domain[i][j].f[3] = domain[i][j].f[1];
        domain[i][j].f[4] = domain[i][j].f[2];
        domain[i][j].f[7] = domain[i][j].f[5];
        domain[i][j].f[6] = domain[i][j].f[8] = (1./2.)
        *((1./2.)*(domain[i-1][j].rho+domain[i][j-1].rho)
        -(domain[i][j].f[0]+domain[i][j].f[1]+domain[i][j].
        f[2]+domain[i][j].f[3]+domain[i][j].f[4]+domain
        [i][j].f[5]+domain[i][j].f[7]));
    }
}
break;
case SOUTH_WEST_CONCAVE_CORNER_VELOCITY_ZOU_HE:
for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
    for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
        domain[i][j].f[5] = domain[i][j].f[7];
        domain[i][j].f[2] = domain[i][j].f[4];
    }
}

```

```

540     domain[i][j].f[1] = domain[i][j].f[3];
541     domain[i][j].f[6] = domain[i][j].f[8] = 0.5*(0.5*(
542         domain[i+1][j].rho+domain[i][j+1].rho)-(domain[i][j].f[0]+domain[i][j].f[1]+domain[i][j].f[2]+domain[i][j].f[3]+domain[i][j].f[4]+domain[i][j].f[5]+
543         domain[i][j].f[7]));
544     }
545     break;
546 case SOUTH_EAST_CONCAVE_CORNER_VELOCITY_ZOU_HE:
547     for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
548         for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
549             domain[i][j].f[3] = domain[i][j].f[1];
550             domain[i][j].f[2] = domain[i][j].f[4];
551             domain[i][j].f[6] = domain[i][j].f[8];
552             domain[i][j].f[5] = domain[i][j].f[7] = 0.5*(0.5*(
553                 domain[i][j+1].rho+domain[i-1][j].rho)-(domain[i][j].f[0]+domain[i][j].f[1]+domain[i][j].f[2]+domain[i][j].f[3]+domain[i][j].f[4]+domain[i][j].f[6]+
554                 domain[i][j].f[8]));
555             }
556             break;
557 case NORTH_WEST_CONVEX_CORNER_VELOCITY_ZOU_HE:
558     for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
559         for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
560             domain[i][j].f[3] = domain[i][j].f[1];
561             domain[i][j].f[2] = domain[i][j].f[4];
562             domain[i][j].f[6] = domain[i][j].f[8];
563             }
564             break;
565 case NORTH_EAST_CONVEX_CORNER_VELOCITY_ZOU_HE:
566     for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
567         for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
568             domain[i][j].f[1] = domain[i][j].f[3];
569             domain[i][j].f[2] = domain[i][j].f[4];
570             domain[i][j].f[5] = domain[i][j].f[7];
571             }
572             break;
573 case SOUTH_WEST_CONVEX_CORNER_VELOCITY_ZOU_HE:
574     for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
575         for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
576             domain[i][j].f[4] = domain[i][j].f[2];
577             domain[i][j].f[3] = domain[i][j].f[1];
578             domain[i][j].f[7] = domain[i][j].f[5];
579             }
580             break;
581     }
582 case SOUTH_EAST_CONVEX_CORNER_VELOCITY_ZOU_HE:
583     for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
584         for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
585             domain[i][j].f[1] = domain[i][j].f[3];
586             domain[i][j].f[8] = domain[i][j].f[6];
587             domain[i][j].f[4] = domain[i][j].f[2];
588             }
589             break;
590     }
591 case HALF WAY_BOUNCEBACK:
592     break;
593 case FULL WAY_BOUNCEBACK:
594     for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){

```

```

595     for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
596         swap( domain[i][j].f[1], domain[i][j].f[3] );
597         swap( domain[i][j].f[2], domain[i][j].f[4] );
598         swap( domain[i][j].f[5], domain[i][j].f[7] );
599         swap( domain[i][j].f[6], domain[i][j].f[8] );
600     }
601 }
602 break;
603 case EAST_OPEN_BOUNDARY:
604     for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
605         for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
606             domain[i][j].f[3] = domain[i-1][j].f[3];
607             domain[i][j].f[6] = domain[i-1][j].f[6];
608             domain[i][j].f[7] = domain[i-1][j].f[7];
609         }
610     }
611 break;
612 case EAST_TEMPERATURE:
613     for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
614         for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
615             for( int alpha = 0; alpha < lattice_model_energy->Q;
616                 alpha++ ){
617                 domain[i][j].g[alpha] = lattice_model_energy->w[
618                     alpha]*bc.T;
619             }
620         }
621     }
622 break;
623 case WEST_TEMPERATURE:
624     for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
625         for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
626             for( int alpha = 0; alpha < lattice_model_energy->Q;
627                 alpha++ ){
628                 domain[i][j].g[alpha] = lattice_model_energy->w[
629                     alpha]*bc.T;
630             }
631         }
632     }
633 break;
634 case ADIABATIC:
635     for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
636         for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
637             swap( domain[i][j].g[1], domain[i][j].g[3] );
638             swap( domain[i][j].g[0], domain[i][j].g[2] );
639         }
640     }
641 break;
642 case NORTHADIABATIC:
643     for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
644         for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
645             for( int alpha = 0; alpha < lattice_model_energy->Q;
646                 alpha++ ){
647                 domain[i][j].g[alpha] = domain[i][j-1].g[alpha];
648             }
649         }
650     }
651 break;
652 case SOUTHADIABATIC:
653     for( int i = bc.start_point.i; i <= bc.end_point.i; i++ ){
654         for( int j = bc.start_point.j; j <= bc.end_point.j; j++ ){
655             for( int alpha = 0; alpha < lattice_model_energy->Q;
656                 alpha++ ){
657                 domain[i][j].g[alpha] = domain[i][j+1].g[alpha];
658             }
659         }
660     }
661

```

```

652         }
653     }
654     break;
655   }
656 }
657 void CalculateForce( Point startPoint , Point endPoint , double& Fx, double&
658 Fy) {
659   if ( startPoint.i >= endPoint.i || startPoint.j >= endPoint.j )
660     return;
661   int ii , jj ;
662   Fx = 0.0;
663   Fy = 0.0;
664   for( int i = startPoint.i; i <= endPoint.i; i++ ){
665     for( int j = startPoint.j; j <= endPoint.j; j++ ){
666       if( domain[ i ][ j ].isBC == true ){
667         for( int alfa = 1; alfa < lattice_model_momentum->Q; alfa
668           ++ ){
669           ii = i + lattice_model_momentum->e_x[
670             lattice_model_momentum->opp[ alfa ] ];
671           jj = j + lattice_model_momentum->e_y[
672             lattice_model_momentum->opp[ alfa ] ];
673           if( domain[ ii ][ jj ].isWet == true ){
674             Fx += lattice_model_momentum->e_x[ alfa ]*(domain[
675               i ][ j ].f[ alfa ] + domain[ ii ][ jj ].f[
676               lattice_model_momentum->opp[ alfa ] ]);
677             Fy += lattice_model_momentum->e_y[ alfa ]*(domain[
678               i ][ j ].f[ alfa ] + domain[ ii ][ jj ].f[
679               lattice_model_momentum->opp[ alfa ] ]);
680           }
681         }
682       }
683     }
684   }
685 }
686 };
687 #endif

```

LBM2.h

## Anexo B

# Código fuente de FVM

```
1 #ifndef FVM_H
2 #define FVM_H
3 #include <math.h>
4 enum ENUMERATION { ZERO_MINUS_ONE_N = 1, ONE_TO_N };
5 enum MESH { UNIFORM = 1, NONUNIFORM };
6 class ScalarField{
7 public:
8     ScalarField( int Nx, int Ny, ENUMERATION enumeration_type){
9         this->enumeration_type = enumeration_type;
10        this->Nx = Nx;
11        this->Ny = Ny;
12        value = new double*[ Nx ];
13        for( int i = 0; i < Nx; i++ ){
14            value[ i ] = new double[ Ny ];
15        }
16        for( int i = 0; i < Nx; i++ ){
17            for( int j = 0; j < Ny; j++ ){
18                value[ i ][ j ] = 0.0;
19            }
20        }
21    }
22    double getValue( int i, int j){
23        if( enumeration_type == ZERO_MINUS_ONE_N )
24            return value[ i ][ j ];
25        else if( enumeration_type == ONE_TO_N )
26            return value[ i-1 ][ j-1 ];
27    }
28    void setValue( int i, int j, double v){
29        if( enumeration_type == ZERO_MINUS_ONE_N )
30            value[ i ][ j ] = v;
31        else if( enumeration_type == ONE_TO_N )
32            value[ i-1 ][ j-1 ] = v;
33    }
34    double &operator() ( int i, int j ){
35        if( enumeration_type == ZERO_MINUS_ONE_N )
36            return value[ i ][ j ];
37        else if( enumeration_type == ONE_TO_N )
38            return value[ i-1 ][ j-1 ];
39    }
40    void print(){
41        printf("\nEscalar\n");
42        for( int j = Ny-1; j >= 0; j-- ){
43            for( int i = 0; i < Nx; i++ ){
44                printf(" %15.9f", value[ i ][ j ]);
```

```

45         }
46         printf("\n");
47     }
48     printf("\n");
49 }
50 double **value;
51 int Nx, Ny;
52 ENUMERATION enumeration_type;
53 };
54 class Mesh{
55 public:
56     Mesh(double Lx, double Ly, int Nx, int Ny, MESH mesh_type){
57         this->Lx = Lx;
58         this->Ly = Ly;
59         this->Nx = Nx;
60         this->Ny = Ny;
61         this->mesh_type = mesh_type;
62         delta_xP = new ScalarField(Nx, Ny, ONE_TO_N);
63         delta_yP = new ScalarField(Nx, Ny, ONE_TO_N);
64         d_WP = new ScalarField(Nx, Ny, ONE_TO_N);
65         d_PE = new ScalarField(Nx, Ny, ONE_TO_N);
66         d_PN = new ScalarField(Nx, Ny, ONE_TO_N);
67         d_SP = new ScalarField(Nx, Ny, ONE_TO_N);
68         aP = new ScalarField(Nx, Ny, ONE_TO_N);
69         aE = new ScalarField(Nx, Ny, ONE_TO_N);
70         aW = new ScalarField(Nx, Ny, ONE_TO_N);
71         aN = new ScalarField(Nx, Ny, ONE_TO_N);
72         aS = new ScalarField(Nx, Ny, ONE_TO_N);
73         bP = new ScalarField(Nx, Ny, ONE_TO_N);
74     }
75     void CalcMesh(){
76         double delta_xE, delta_xW, delta_yN, delta_yS;
77         for( int i = 1; i <= Nx; i++ ){
78             for( int j = 1; j <= Ny; j++ ){
79                 (*d_WP)(i,j) = 0.0;
80                 (*d_PE)(i,j) = 0.0;
81                 (*d_PN)(i,j) = 0.0;
82                 (*d_SP)(i,j) = 0.0;
83                 delta_xW = delta_xE = delta_yN = delta_yS = 0.0;
84                 (*aP)(i,j) = 0.0;
85                 (*aN)(i,j) = 0.0;
86                 (*aS)(i,j) = 0.0;
87                 (*aW)(i,j) = 0.0;
88                 (*aE)(i,j) = 0.0;
89                 (*bP)(i,j) = 0.0;
90                 (*delta_xP)(i,j) = xi(i) - xi(i-1);
91                 (*delta_yP)(i,j) = yj(j) - yj(j-1);
92                 delta_xW = xi(i-1)-xi(i-2);
93                 delta_xE = xi(i+1)-xi(i);
94                 delta_yN = yj(j+1)-yj(j);
95                 delta_yS = yj(j-1)-yj(j-2);
96                 if( i != 1 ) (*d_WP)(i,j) = 0.5*(delta_xW+(*delta_xP)(i,j));
97                 else (*d_WP)(i,j) = 0.5*(*delta_xP)(i,j);
98                 if( i != Nx ) (*d_PE)(i,j) = 0.5*(delta_xE+(*delta_xP)(i,j));
99                 else (*d_PE)(i,j) = 0.5*(*delta_xP)(i,j);
100                if( j != 1 ) (*d_SP)(i,j) = 0.5*(delta_yN+(*delta_yP)(i,j));
101                else (*d_SP)(i,j) = 0.5*(*delta_yP)(i,j);
102                if( j != Ny ) (*d_PN)(i,j) = 0.5*(delta_yN+(*delta_yP)(i,j));
103                else (*d_PN)(i,j) = 0.5*(*delta_yP)(i,j);
104                if( i != 1 ) (*aW)(i,j) = -(*delta_yP)(i,j) / (*d_WP)(i,j);
105                if( i != Nx ) (*aE)(i,j) = -(*delta_yP)(i,j) / (*d_PE)(i,j);
106                if( j != 1 ) (*aS)(i,j) = -(*delta_xP)(i,j) / (*d_SP)(i,j);
107                if( j != Ny ) (*aN)(i,j) = -(*delta_xP)(i,j) / (*d_PN)(i,j);

```

```

108         (*aP)( i , j ) = (*aE)( i , j ) + (*aW)( i , j ) + (*aN)( i , j ) + (*aS)( i , j )
109         ;
110         if( i == 1 && j == 1){
111             (*aP)( i , j ) = (*aP)( i , j ) * 1.1;
112         }
113     }
114 }
115 double Lx, Ly;
116 int Nx, Ny;
117 MESH mesh_type;
118 double C;
119 ScalarField *delta_xP;
120 ScalarField *delta_yP;
121 ScalarField *d_WP, *d_PE, *d_PN, *d_SP;
122 ScalarField *aP, *aE, *aW, *aN, *aS;
123 ScalarField *bP;
124 private:
125     double xi( int i ){
126         double A;
127         if( mesh_type == UNIFORM ){
128             return (Lx*i)/(Nx);
129         } else if( mesh_type == NONUNIFORM ){
130             A = -Lx*0.5/(tanh(-C*Nx*0.5));
131             return A*tanh(C*(i-Nx*0.5))+Lx*0.5;
132         }
133     }
134     double yj( int j ){
135         double A;
136         if( mesh_type == UNIFORM ){
137             return (Ly*j)/(Ny);
138         } else if( mesh_type == NONUNIFORM ){
139             A = -Ly*0.5/(tanh(-C*Ny*0.5));
140             return A*tanh(C*(j-Ny*0.5))+Ly*0.5;
141         }
142     }
143 };
144 class VelocityField{
145     public:
146         VelocityField(int Nx, int Ny){
147             this->Nx = Nx;
148             this->Ny = Ny;
149             u = new ScalarField(Nx+1, Ny+2, ZERO_TO_MINUS_ONE_N);
150             v = new ScalarField(Nx+2, Ny+1, ZERO_TO_MINUS_ONE_N);
151         }
152         ScalarField CalcDivergence(Mesh& mesh){
153             ScalarField div(Nx, Ny, ONE_TO_N);
154             for( int i = 1; i <= Nx; i++ ){
155                 for( int j = 1; j <= Ny; j++ ){
156                     div(i, j)=(*mesh.delta_yP)(i, j)*((*u)(i, j)-(*u)(i-1,j))+(*mesh.
157                         delta_xP)(i, j)*((*v)(i, j)-(*v)(i, j-1));
158                 }
159             }
160             return div;
161         }
162         void print(){
163             printf( "\nComponente u\n" );
164             for( int j = Ny+1; j >= 0; j-- ){
165                 for( int i = 0; i <= Nx; i++ ){
166                     printf( "%15.9f", (*u)(i, j) );
167                 }
168             }
169         }
170     }
171 }
```

```

169     printf("\n");
170     printf("\nComponente v\n");
171     for( int j = Ny; j >= 0; j-- ){
172         for( int i = 0; i <= Nx+1; i++ ){
173             printf("%15.9f", (*v)(i,j));
174         }
175         printf("\n");
176     }
177     printf("\n");
178 }
179 ScalarField *u;
180 ScalarField *v;
181 int Nx, Ny;
182 };
183 void CalcMomentumConvectiveDifusiveTerm_ForcedConvection( Mesh& mesh,
184 VelocityField& u, double Re, ScalarField& Rx, ScalarField &Ry ){
185     double Rei = 1.0 / Re;
186     double difusive;
187     double convective;
188     double F_E, F_W, F_e, F_w, F_N, F_S, F_n, F_s, F_P;
189     double u_e, u_n, u_w, u_s;
190     double v_e, v_n, v_w, v_s;
191     for( int i = 1; i <= mesh.Nx-1; i++ ){
192         for( int j = 1; j <= mesh.Ny; j++ ){
193             F_P = (*u.u)(i,j) * (*mesh.delta_yP)(i,j);
194             F_E = (*u.u)(i+1,j) * (*mesh.delta_yP)(i,j);
195             F_W = (*u.u)(i-1,j) * (*mesh.delta_yP)(i,j);
196             F_e = .5 * (F_P + F_E);
197             F_w = .5 * (F_W + F_P);
198             F_n = .5 * (-(u.v)(i,j) * (*mesh.delta_xP)(i,j) + (u.v)(i+1,j) *
199                         (*mesh.delta_xP)(i+1,j));
200             F_s = .5 * (-(u.v)(i,j-1) * (*mesh.delta_xP)(i,j) + (u.v)(i+1,j-
201                         1) * (*mesh.delta_xP)(i+1,j));
202             u_e = .5 * (-(u.u)(i+1,j) + (u.u)(i,j));
203             u_w = .5 * (-(u.u)(i-1,j) + (u.u)(i,j));
204             u_n = .5 * (-(u.u)(i,j+1) + (u.u)(i,j));
205             u_s = .5 * (-(u.u)(i,j-1) + (u.u)(i,j));
206             difusive = ( - ((u.u)(i,j) - (u.u)(i-1,j)) * (*mesh.delta_yP)(i,j) /
207                         (*mesh.delta_xP)(i,j) +
208                         ((u.u)(i+1,j) - (u.u)(i,j)) * (*mesh.delta_yP)(i,j) /
209                         (*mesh.delta_xP)(i+1,j) +
210                         ((u.u)(i,j+1) - (u.u)(i,j)) * (*mesh.d_PE)(i,j) /
211                         (*mesh.d_PN)(i,j) -
212                         ((u.u)(i,j) - (u.u)(i,j-1)) * (*mesh.d_PE)(i,j) /
213                         (*mesh.d_SP)(i,j)
214                         ) * Rei;
215             convective = u_e * F_e + u_n * F_n - u_w * F_w - u_s * F_s;
216             Rx(i,j) = difusive - convective;
217             Rx(i,j) = Rx(i,j) / ( (*mesh.d_PE)(i,j) * (*mesh.delta_yP)(i,j) );
218         }
219     }
220     for( int i = 1; i <= mesh.Nx; i++ ){
221         for( int j = 1; j <= mesh.Ny-1; j++ ){
222             F_P = (*u.v)(i,j) * (*mesh.delta_xP)(i,j);
223             F_N = (*u.v)(i,j+1) * (*mesh.delta_xP)(i,j);
224             F_S = (*u.v)(i,j-1) * (*mesh.delta_xP)(i,j);
225             F_e = .5 * (-(u.u)(i,j) * (*mesh.delta_yP)(i,j) + (u.u)(i,j+1) *
226                         (*mesh.delta_yP)(i,j+1));
227             F_w = .5 * (-(u.u)(i-1,j) * (*mesh.delta_yP)(i,j) + (u.u)(i-1,j+
228                         1) * (*mesh.delta_yP)(i,j+1));
229             F_n = .5 * (F_P + F_N);
230             F_s = .5 * (F_P + F_S);
231             v_e = .5 * (-(u.v)(i+1,j) + (u.v)(i,j));

```

```

223     v_w = .5 * ( (*u.v)(i-1,j) + (*u.v)(i,j) );
224     v_n = .5 * ( (*u.v)(i,j+1) + (*u.v)(i,j) );
225     v_s = .5 * ( (*u.v)(i,j-1) + (*u.v)(i,j) );
226     difusive = ( - ((*u.v)(i,j) - (*u.v)(i-1,j)) * (*mesh.d_PN)(i,j) /
227                   (*mesh.d_WP)(i,j)
228                   + ((*u.v)(i+1,j) - (*u.v)(i,j)) * (*mesh.d_PN)(i,j) /
229                     (*mesh.d_PE)(i,j)
230                   + ((*u.v)(i,j+1) - (*u.v)(i,j)) * (*mesh.delta_xP)(i,
231                     j) / (*mesh.delta_yP)(i,j+1)
232                   - ((*u.v)(i,j) - (*u.v)(i,j-1)) * (*mesh.delta_xP)(i,
233                     j) / (*mesh.delta_yP)(i,j)
234                   ) * Rei;
235     convective = v_e * F_e + v_n * F_n - v_w * F_w - v_s * F_s;
236     Ry(i,j) = difusive - convective;
237     Ry(i,j) = Ry(i,j) / ( (*mesh.d_PN)(i,j) * (*mesh.delta_xP)(i,j) );
238   }
239 }
240 void CalcMomentumConvectiveDifusiveTerm_NaturalConvection( Mesh& mesh,
241   VelocityField& u, ScalarField& T, double Pr, double Ra, ScalarField& Rx,
242   ScalarField &Ry ){
243   double RaPr = Ra * Pr;
244   double difusive;
245   double convective;
246   double F_E, F_W, F_e, F_w, F_N, F_S, F_n, F_s, F_P;
247   double u_e, u_n, u_w, u_s;
248   double v_e, v_n, v_w, v_s;
249   for( int i = 1; i <= mesh.Nx-1; i++ ){
250     for( int j = 1; j <= mesh.Ny; j++ ){
251       F_P = (*u.u)(i,j) * (*mesh.delta_yP)(i,j);
252       F_E = (*u.u)(i+1,j) * (*mesh.delta_yP)(i,j);
253       F_W = (*u.u)(i-1,j) * (*mesh.delta_yP)(i,j);
254       F_e = .5 * (F_P + F_E);
255       F_w = .5 * (F_W + F_P);
256       F_n = .5 * ( (*u.v)(i,j) * (*mesh.delta_xP)(i,j) + (*u.v)(i+1,j) *
257                     (*mesh.delta_xP)(i+1,j) );
258       F_s = .5 * ( (*u.v)(i,j-1) * (*mesh.delta_xP)(i,j) + (*u.v)(i+1,j
259                     -1) * (*mesh.delta_xP)(i+1,j) );
260       u_e = .5 * ( (*u.u)(i+1,j) + (*u.u)(i,j) );
261       u_w = .5 * ( (*u.u)(i-1,j) + (*u.u)(i,j) );
262       u_n = .5 * ( (*u.u)(i,j+1) + (*u.u)(i,j) );
263       u_s = .5 * ( (*u.u)(i,j-1) + (*u.u)(i,j) );
264       difusive = ( - ((*u.u)(i,j) - (*u.u)(i-1,j)) * (*mesh.delta_yP)(i,
265                     j) / (*mesh.delta_xP)(i,j)
266                     + ((*u.u)(i+1,j) - (*u.u)(i,j)) * (*mesh.delta_yP)(i,
267                     j) / (*mesh.delta_xP)(i+1,j)
268                     + ((*u.u)(i,j+1) - (*u.u)(i,j)) * (*mesh.d_PE)(i,j) /
269                       (*mesh.d_PN)(i,j)
270                     - ((*u.u)(i,j) - (*u.u)(i,j-1)) * (*mesh.d_PE)(i,j) /
271                       (*mesh.d_SP)(i,j)
272                     ) * Pr;
273     convective = u_e * F_e + u_n * F_n - u_w * F_w - u_s * F_s;
274     Rx(i,j) = difusive - convective;
275     Rx(i,j) = Rx(i,j) / ( (*mesh.d_PE)(i,j) * (*mesh.delta_yP)(i,j) );
276   }
277   for( int i = 1; i <= mesh.Nx; i++ ){
278     for( int j = 1; j <= mesh.Ny-1; j++ ){
279       F_P = (*u.v)(i,j) * (*mesh.delta_xP)(i,j);
280       F_N = (*u.v)(i,j+1) * (*mesh.delta_xP)(i,j);
281       F_S = (*u.v)(i,j-1) * (*mesh.delta_xP)(i,j);
282       F_e = .5 * ( (*u.u)(i,j) * (*mesh.delta_yP)(i,j) + (*u.u)(i,j+1) *
283                     (*mesh.delta_yP)(i,j+1) );

```

```

273     F_w = .5 * ( (*u.u)(i-1,j) * (*mesh.delta_yP)(i,j) + (*u.u)(i-1,j
274         +1) * (*mesh.delta_yP)(i,j+1) );
275     F_n = .5 * (F_P + F_N);
276     F_s = .5 * (F_P + F_S);
277     v_e = .5 * ( (*u.v)(i+1,j) + (*u.v)(i,j) );
278     v_w = .5 * ( (*u.v)(i-1,j) + (*u.v)(i,j) );
279     v_n = .5 * ( (*u.v)(i,j+1) + (*u.v)(i,j) );
280     v_s = .5 * ( (*u.v)(i,j-1) + (*u.v)(i,j) );
281     difusive = ( - ((*u.v)(i,j) - (*u.v)(i-1,j)) * (*mesh.d_PN)(i,j) /
282         (*mesh.d_WP)(i,j)
283         + ((*u.v)(i+1,j) - (*u.v)(i,j)) * (*mesh.d_PN)(i,j) /
284             (*mesh.d_PE)(i,j)
285         + ((*u.v)(i,j+1) - (*u.v)(i,j)) * (*mesh.delta_xP)(i,
286             j) / (*mesh.delta_yP)(i,j+1)
287         - ((*u.v)(i,j) - (*u.v)(i,j-1)) * (*mesh.delta_xP)(i,
288             j) / (*mesh.delta_yP)(i,j)
289         ) * Pr;
290     convective = v_e * F_e + v_n * F_n - v_w * F_w - v_s * F_s;
291     Ry(i,j) = difusive - convective;
292     Ry(i,j) = Ry(i,j) / ((*mesh.d_PN)(i,j) * (*mesh.delta_xP)(i,j));
293     Ry(i,j) += RaPr * T(i,j);
294 }
295 }
296 void CalcEnergyConvectiveDifusiveTerm_NaturalConvection( Mesh& mesh,
297     VelocityField& u, ScalarField& T, ScalarField& Q){
298     double difusive;
299     double convective;
300     double F_e, F_w, F_n, F_s;
301     double T_e, T_n, T_w, T_s;
302     for( int i = 1; i <= mesh.Nx; i++ ){
303         for( int j = 1; j <= mesh.Ny; j++ ){
304             F_e = (*mesh.delta_yP)(i,j) * (*u.u)(i,j);
305             F_w = (*mesh.delta_yP)(i,j) * (*u.u)(i-1,j);
306             F_n = (*mesh.delta_xP)(i,j) * (*u.v)(i,j);
307             F_s = (*mesh.delta_xP)(i,j) * (*u.v)(i,j-1);
308             T_e = .5 * ( T(i+1,j) + T(i,j) );
309             T_w = .5 * ( T(i-1,j) + T(i,j) );
310             T_n = .5 * ( T(i,j+1) + T(i,j) );
311             T_s = .5 * ( T(i,j-1) + T(i,j) );
312             difusive = ( - (T(i,j) - T(i-1,j)) * (*mesh.delta_yP)(i,j) / (
313                 *mesh.d_WP)(i,j)
314                 + (T(i+1,j) - T(i,j)) * (*mesh.delta_yP)(i,j) / (
315                     *mesh.d_PE)(i,j)
316                 + (T(i,j+1) - T(i,j)) * (*mesh.delta_xP)(i,j) / (
317                     *mesh.d_PN)(i,j)
318                 - (T(i,j) - T(i,j-1)) * (*mesh.delta_xP)(i,j) / (
319                     *mesh.d_SP)(i,j)
320             );
321             convective = T_e * F_e + T_n * F_n - T_w * F_w - T_s * F_s;
322             Q(i,j) = difusive - convective;
323             Q(i,j) = Q(i,j) / ((*mesh.delta_xP)(i,j) * (*mesh.delta_yP)(i,j));
324         }
325     }
326     double CalcTimeInterval( Mesh& mesh, VelocityField& u, double nu, double
327         C_conv, double C_visc ){
328         double tmp;
329         double max1 = 0.0, max2 = 0.0;
330         for( int i = 1; i <= mesh.Nx-1; i++ ){
331             for( int j = 1; j <= mesh.Ny; j++ ){
332                 tmp = (fabs((*u.u)(i,j)) / (*mesh.delta_xP)(i,j) ) ;
333                 if( tmp > max1) max1 = tmp;

```

```

325         tmp = (nu / pow((*mesh.delta_xP)(i,j), 2.0));
326         if( tmp > max2 ) max2 = tmp;
327     }
328 }
329 for( int i = 1; i <= mesh.Nx; i++ ){
330     for( int j = 1; j <= mesh.Ny-1; j++ ){
331         tmp = (fabs((*u.v)(i,j)) / (*mesh.delta_yP)(i,j));
332         if( tmp > max1 ) max1 = tmp;
333         tmp = (nu / pow((*mesh.delta_yP)(i,j), 2.0));
334         if( tmp > max2 ) max2 = tmp;
335     }
336 }
337 return fmin((C_conv/max1), (C_visc/max2));
338 }
339 double CalcMaxDerivative( ScalarField& scalar_field1, ScalarField&
scalar_field2, double delta_t){
340     double max_derivative = 0.0, tmp = 0.0;
341     if( scalar_field1.enumeration_type == ZERO_TO_MINUS_ONE_N ){
342         for( int i = 0; i < scalar_field1.Nx; i++ ){
343             for( int j = 0; j < scalar_field1.Ny; j++ ){
344                 tmp = fabs((scalar_field1.getValue(i,j) - scalar_field2.
getValue(i,j)) / delta_t);
345                 if( tmp > max_derivative ) max_derivative = tmp;
346             }
347         }
348     } else if( scalar_field1.enumeration_type == ONE_TO_N ){
349         for( int i = 1; i <= scalar_field1.Nx; i++ ){
350             for( int j = 1; j <= scalar_field1.Ny; j++ ){
351                 tmp = fabs((scalar_field1.getValue(i,j) - scalar_field2.
getValue(i,j)) / delta_t);
352                 if( tmp > max_derivative ) max_derivative = tmp;
353             }
354         }
355     }
356     return max_derivative;
357 }
358 void MultiplySparseMatrixByVector_A( ScalarField& aP, ScalarField& aW,
ScalarField& aE, ScalarField& aS, ScalarField& aN, ScalarField& x,
ScalarField& y){
359     int nodoP = 1;
360     for( int j = 1; j <= aP.Ny; j++ ){
361         for( int i = 1; i <= aP.Nx; i++ ){
362             y(1, nodoP) = aP(i,j) * x(1, nodoP);
363             if( i+1 <= aP.Nx) y(1, nodoP) -= aE(i,j) * x(1, nodoP + 1);
364             if( i-1 >= 1) y(1, nodoP) -= aW(i,j) * x(1, nodoP - 1);
365             if( j+1 <= aP.Ny) y(1, nodoP) -= aN(i,j) * x(1, nodoP + aP.Nx);
366             if( j-1 >= 1) y(1, nodoP) -= aS(i,j) * x(1, nodoP - aP.Nx);
367             nodoP++;
368         }
369     }
370 }
371 void MultiplySparseMatrixByVector_B( ScalarField& aP, ScalarField& aW,
ScalarField& aE, ScalarField& aS, ScalarField& aN, ScalarField& x,
ScalarField& y){
372     for( int j = 1; j <= aP.Ny; j++ ){
373         for( int i = 1; i <= aP.Nx; i++ ){
374             y(i,j) = aP(i,j) * x(i,j);
375             if( i+1 <= aP.Nx) y(i,j) -= aE(i,j) * x(i+1,j);
376             if( i-1 >= 1) y(i,j) -= aW(i,j) * x(i-1,j);
377             if( j+1 <= aP.Ny) y(i,j) -= aN(i,j) * x(i,j+1);
378             if( j-1 >= 1) y(i,j) -= aS(i,j) * x(i,j-1);
379         }
380     }

```

```

381 }
382 int SolverConjugateGradient_A( ScalarField& aP, ScalarField& aW, ScalarField&
383 aE, ScalarField& aS, ScalarField& aN, ScalarField& bP, ScalarField& x,
384 double epsilon , ScalarField& tmp_r, ScalarField& tmp_p, ScalarField&
385 tmp_Ap){
386     int DIM = aP.Nx * aP.Ny;
387     int iteraciones = 0;
388     int i , j;
389     double rsold = 0.0 , alpha = 0.0 , rsnew = 0.0;
390     MultiplySparseMatrixByVector_A(aP, aW, aE, aS, aN, x, tmp_r);
391     for( j = 1; j <= DIM; j++ ){
392         tmp_r(1,j) = (-1)*tmp_r(1,j) + bP(1,1);
393         tmp_p(1,j) = tmp_r(1,j);
394         rsold += tmp_r(1,j) * tmp_r(1,j);
395     }
396     for( int k = 1; k <= DIM; k++ ){
397         alpha = 0.0;
398         rsnew = 0.0;
399         iteraciones++;
400         MultiplySparseMatrixByVector_A(aP, aW, aE, aS, aN, tmp_p, tmp_Ap);
401         for( j = 1; j <= DIM; j++ ){
402             alpha += tmp_p(1,j) * tmp_Ap(1,j);
403         }
404         alpha = rsold / alpha;
405         for( j = 1; j <= DIM; j++ ){
406             x(1,j) = x(1,j) + alpha * tmp_p(1,j);
407             tmp_r(1,j) = tmp_r(1,j) - alpha * tmp_Ap(1,j);
408             rsnew += tmp_r(1,j) * tmp_r(1,j);
409         }
410         if( sqrt(rsnew) < epsilon ) return iteraciones;
411         for( j = 1; j <= DIM; j++ ){
412             tmp_p(1,j) = tmp_r(1,j) + (rsnew/rsold) * tmp_p(1,j);
413         }
414         rsold = rsnew;
415     }
416     return -1;
417 }
418 int SolverConjugateGradient_B( ScalarField& aP, ScalarField& aW, ScalarField&
419 aE, ScalarField& aS, ScalarField& aN, ScalarField& bP, ScalarField& x,
420 double epsilon , ScalarField& tmp_r, ScalarField& tmp_p, ScalarField&
421 tmp_Ap){
422     int DIM = aP.Nx * aP.Ny;
423     int iteraciones = 0;
424     int i , j;
425     double rsold = 0.0 , alpha = 0.0 , rsnew = 0.0;
426     MultiplySparseMatrixByVector_B(aP, aW, aE, aS, aN, x, tmp_r);
427     for( j = 1; j <= aP.Ny; j++ ){
428         for( i = 1; i <= aP.Nx; i++ ){
429             tmp_r(i,j) = (-1)*tmp_r(i,j) + bP(i,j);
430             tmp_p(i,j) = tmp_r(i,j);
431             rsold += tmp_r(i,j) * tmp_r(i,j);
432         }
433         for( int k = 1; k <= DIM; k++ ){
434             alpha = 0.0;
435             rsnew = 0.0;
436             iteraciones++;
437             MultiplySparseMatrixByVector_B(aP, aW, aE, aS, aN, tmp_p, tmp_Ap);
438             for( j = 1; j <= aP.Ny; j++ ){
439                 for( i = 1; i <= aP.Nx; i++ ){
440                     alpha += tmp_p(i,j) * tmp_Ap(i,j);
441                 }
442             }
443         }
444     }

```

```

438     alpha = rsold / alpha;
439     for( j = 1; j <= aP.Ny; j++ ){
440         for( i = 1; i <= aP.Nx; i++ ){
441             x(i,j) = x(i,j) + alpha * tmp_p(i,j);
442             tmp_r(i,j) = tmp_r(i,j) - alpha * tmp_Ap(i,j);
443             rsnew += tmp_r(i,j) * tmp_r(i,j);
444         }
445     }
446     if( sqrt(rsnew) < epsilon ) return iteraciones;
447     for( j = 1; j <= aP.Ny; j++ ){
448         for( i = 1; i <= aP.Nx; i++ ){
449             tmp_p(i,j) = tmp_r(i,j) + (rsnew/rsold) * tmp_p(i,j);
450         }
451     }
452     rsold = rsnew;
453 }
454 return -1;
455 }
456 int CalcIncompressibleVelocityField_GaussSheidel( Mesh& mesh, VelocityField&
457 u_comp, VelocityField& u_incomp, ScalarField& p, ScalarField& p_supuesta,
458 double epsilon, double omega){
459     double max_error, error;
460     int it_count = 0;
461     double tmp;
462     for( int i = 1; i <= mesh.Nx; i++ ){
463         for( int j = 1; j <= mesh.Ny; j++ ){
464             if( !(i == 1 && j == 1) ){
465                 (*mesh.bP)(i,j) = (*mesh.delta_yP)(i,j) * ( (*u_comp.u)(i,j) - (*
466                     u_comp.u)(i-1,j) ) +
467                     (*mesh.delta_xP)(i,j) * ( (*u_comp.v)(i,j) - (*
468                         u_comp.v)(i,j-1) );
469             }
470         }
471     }
472     do{
473         max_error = 0.0;
474         it_count++;
475         for( int j = 1; j <= mesh.Ny; j++ ){
476             for( int i = 1; i <= mesh.Nx; i++ ){
477                 tmp = 0.0;
478                 if( i+1 <= mesh.Nx) tmp += (*mesh.aE)(i,j) * p_supuesta(i+1,j)
479 ;
480                 if( i-1 >= 1) tmp += (*mesh.aW)(i,j) * p(i-1,j);
481                 if( j+1 <= mesh.Ny) tmp += (*mesh.aN)(i,j) * p_supuesta(i,j+1)
482 ;
483                 if( j-1 >= 1) tmp += (*mesh.aS)(i,j) * p(i,j-1);
484                 p(i,j) = ( (*mesh.bP)(i,j) + tmp ) / (*mesh.aP)(i,j);
485                 error = fabs( p(i,j) - p_supuesta(i,j) );
486                 if( error > max_error ) max_error = error;
487             }
488         }
489     }while( max_error > epsilon );
490     for( int i = 0; i <= mesh.Nx+1; i++ ){
491         for( int j = 0; j <= mesh.Ny+1; j++ ){
492             if( i <= mesh.Nx )

```

```

494         (*u_incomp.u)(i,j) = (*u_comp.u)(i,j);
495         if( j <= mesh.Ny )
496             (*u_incomp.v)(i,j) = (*u_comp.v)(i,j);
497     }
498 }
499 for( int i = 1; i <= mesh.Nx; i++ ){
500     for( int j = 1; j <= mesh.Ny; j++ ){
501         if( i < mesh.Nx )
502             (*u_incomp.u)(i,j) = (*u_incomp.u)(i,j) - ( p(i+1,j) - p(i,j) )
503                                         / (*mesh.d_PE)(i,j);
504         if( j < mesh.Ny )
505             (*u_incomp.v)(i,j) = (*u_incomp.v)(i,j) - ( p(i,j+1) - p(i,j) )
506                                         / (*mesh.d_PN)(i,j);
507     }
508 }
509 return it_count;
510 }
511 int CalcIncompressibleVelocityField_ConjugateGradient( Mesh& mesh,
512 VelocityField& u_comp, VelocityField& u_incomp, ScalarField& p, double
513 epsilon, ScalarField& tmp_r, ScalarField& tmp_p, ScalarField& tmp_Ap){
514     int it_count = 0;
515     for( int i = 1; i <= mesh.Nx; i++ ){
516         for( int j = 1; j <= mesh.Ny; j++ ){
517             (*mesh.bP)(i,j) = (*mesh.delta_yP)(i,j) * ( (*u_comp.u)(i,j) - (*
518                 u_comp.u)(i-1,j) ) +
519                             (*mesh.delta_xP)(i,j) * ( (*u_comp.v)(i,j) - (*
520                 u_comp.v)(i,j-1) );
521         }
522     }
523     it_count = SolverConjugateGradient_B( *mesh.aP, *mesh.aW, *mesh.aE, *mesh.
524     aS, *mesh.aN, *mesh.bP, p, epsilon, tmp_r, tmp_p, tmp_Ap );
525     for( int i = 0; i <= mesh.Nx+1; i++ ){
526         for( int j = 0; j <= mesh.Ny+1; j++ ){
527             if( i <= mesh.Nx )
528                 (*u_incomp.u)(i,j) = (*u_comp.u)(i,j);
529             if( j <= mesh.Ny )
530                 (*u_incomp.v)(i,j) = (*u_comp.v)(i,j);
531         }
532     }
533     for( int i = 1; i <= mesh.Nx; i++ ){
534         for( int j = 1; j <= mesh.Ny; j++ ){
535             if( i < mesh.Nx )
536                 (*u_incomp.u)(i,j) = (*u_incomp.u)(i,j) - ( p(i+1,j) - p(i,j) )
537                                         / (*mesh.d_PE)(i,j);
538             if( j < mesh.Ny )
539                 (*u_incomp.v)(i,j) = (*u_incomp.v)(i,j) - ( p(i,j+1) - p(i,j) )
540                                         / (*mesh.d_PN)(i,j);
541         }
542     }
543     return it_count;
544 }
545 void CalcStreamFunction( Mesh& mesh, VelocityField& u, ScalarField&
546 f_corriente){
547     f_corriente(0,0) = 0.0;
548     for( int i = 1; i <= mesh.Nx; i++){
549         f_corriente(i,0) = f_corriente(i-1,0) + (*u.v)(i,0)*(*mesh.delta_xP)(i
550             ,1);
551     }
552     for( int i = 0; i <= mesh.Ny; i++){
553         for( int j = 1; j <= mesh.Ny; j++ ){
554             f_corriente(i,j) = f_corriente(i,j-1) - (*u.u)(i,j)*(*mesh.
555                 delta_yP)(1,j);
556         }
557     }

```

```
545 }  
546 }  
547 #endif
```

FVM.h

## Anexo C

# Código fuente de Driven Cavity

```
1 #include <cstdlib>
2 #include <math.h>
3 #include <stdio.h>
4 #include <iostream>
5 #include <iomanip>
6 #include <time.h>
7 #include "//FVM.h"
8 using namespace std;
9 class ComputationTime{
10     clock_t start;
11 public:
12     void Start(){
13         start = clock();
14     }
15     double Stop(){
16         return (((double) (clock() - start)) / CLOCKS_PER_SEC);
17     }
18 };
19 enum SOLVER {GAUSS_SHEIDEL = 1, CONJUGATE_GRADIENT};
20 int main(int argc, char** argv) {
21     SOLVER solver_type = CONJUGATE_GRADIENT;
22     MESH mesh_type = UNIFORM;
23     const double C = 0.015;
24     const double Lx = 1.0, Ly = 1.0;
25     const int Nx = 150, Ny = 150;
26     const double Re = 1000;
27     const double U = 1.0;
28     double time = 0.0;
29     int ciclos = 0;
30     double delta_t = 0;
31     int iteraciones = 0;
32     VelocityField un(Nx, Ny);
33     VelocityField un1(Nx, Ny);
34     VelocityField up(Nx, Ny);
35     VelocityField temp(Nx, Ny);
36     ScalarField p(Nx, Ny, ONE_TO_N);
37     ScalarField p_supuesta(Nx, Ny, ONE_TO_N);
38     ScalarField Rxn(Nx, Ny, ONE_TO_N);
39     ScalarField Ryn(Nx, Ny, ONE_TO_N);
40     ScalarField Rxn1(Nx, Ny, ONE_TO_N);
41     ScalarField Ryn1(Nx, Ny, ONE_TO_N);
42     ScalarField tmp_r(Nx, Ny, ONE_TO_N);
43     ScalarField tmp_p(Nx, Ny, ONE_TO_N);
44     ScalarField tmp_Ap(Nx, Ny, ONE_TO_N);
```

```

45 Mesh *mesh;
46 FILE *file_out_data = fopen( "Data.txt" , "w" );
47 int i , j ;
48 double convergence_u , convergence_v;
49 const double factor_relajacion_solver = 1.1;
50 const double epsilon_solver = 1E-8;
51 const double epsilon = 6E-4;
52 const double tiempo_max_simulacion = 200;
53 const double tiempo_min_simulacion = 0.05;
54 ComputationTime computation_time;
55 double ct_max = 0.0 , ct_min = 1, ct_total = 0.0;
56 double ct_temporal = 0.0;
57 mesh = new Mesh(Lx, Ly, Nx, Ny, mesh_type);
58 mesh->C = C;
59 mesh->CalcMesh();
60 for( i = 1; i <= Nx-1; i++ ){
61     for( j = 1; j <= Ny; j++ ){
62         (*un1.u)(i,j) = 0.0;
63         (*un.u)(i,j) = 0.0;
64     }
65 }
66 for( i = 1; i <= Nx; i++ ){
67     for( j = 1; j <= Ny-1; j++ ){
68         (*un1.v)(i,j) = 0.0;
69         (*un.v)(i,j) = 0.0;
70     }
71 }
72 for( i = 0; i <= Nx; i++ ){
73     (*temp.u)(i,Ny+1) = U;
74 }
75 for( i = 1; i <= Nx; i++ ){
76     for( j = 1; j <= Ny; j++ ){
77         p(i,j) = 1.0;
78         p_supuesta(i,j) = 1.0;
79     }
80 }
81 do{
82     computation_time.Start();
83     delta_t = CalcTimeInterval( *mesh, un, 1.0/Re, .35*0.4, .2*0.4 );
84     CalcMomentumConvectiveDifusiveTerm_ForcedConvection( *mesh, un, Re,
85             Rxn, Ryn );
86     CalcMomentumConvectiveDifusiveTerm_ForcedConvection( *mesh, un1, Re,
87             Rxn1, Ryn1 );
88     for( i = 1; i <= mesh->Nx; i++ ){
89         for( j = 1; j <= mesh->Ny; j++ ){
90             if( i < mesh->Nx ){
91                 (*up.u)(i,j) = (*un.u)(i,j) + delta_t * ( 1.5 * Rxn(i,j) -
92                     0.5 * Rxn1(i,j) );
93             }
94             if( j < mesh->Ny ){
95                 (*up.v)(i,j) = (*un.v)(i,j) + delta_t * ( 1.5 * Ryn(i,j) -
96                     0.5 * Ryn1(i,j) );
97             }
98         }
99     }
100    for( i = 0; i <= Nx; i++ ){
101        (*up.u)(i,Ny+1) = U;
102    }
103    for( i = 0; i <= mesh->Nx+1; i++ ){
104        for( j = 0; j <= mesh->Ny+1; j++ ){
105            if( i <= mesh->Nx ){
106                (*un1.u)(i,j) = (*un.u)(i,j);
107            }
108        }
109    }

```

```

104         if( j <= mesh->Ny ){
105             (*un1.v)(i,j) = (*un.v)(i,j);
106         }
107     }
108 }
109 if( solver_type == GAUSS_SHEIDEL ){
110     iteraciones = CalcIncompressibleVelocityField_GaussSheidel( *mesh,
111                     up, un, p, p_supuesta, epsilon_solver,
112                     factor_relajacion_solver);
113 } else if( solver_type == CONJUGATE_GRADIENT ){
114     iteraciones = CalcIncompressibleVelocityField_ConjugateGradient( *
115                     mesh, up, un, p, epsilon_solver, tmp_r, tmp_p, tmp_Ap);
116 }
117 ct_temporal = computation_time.Stop();
118 if( ct_max < ct_temporal ) ct_max = ct_temporal;
119 if( ct_min > ct_temporal ) ct_min = ct_temporal;
120 ct_total += ct_temporal;
121 printf("Tiempo = %f (%d,%f,%f)", time, iteraciones, delta_t,
122       ct_temporal);
123 convergence_u = CalcMaxDerivative((*un.u), (*un1.u), delta_t);
124 convergence_v = CalcMaxDerivative((*un.v), (*un1.v), delta_t);
125 i = j = Nx/2;
126 fprintf(file_out_data, "%d %.9f %.9f %.9f %.9f %.9f %.9f %.9f %.9f\n",
127         ciclos, time, delta_t, ct_temporal, iteraciones, convergence_u,
128         , convergence_v, (*un.u)(i,j) / U, (*un.v)(i,j) / U, p(i,j) /
129         delta_t );
130 printf(" {%.2f, %.2f}\n", convergence_u, convergence_v);
131 time += delta_t;
132 ciclos++;
133 if( convergence_u <= epsilon && convergence_v <= epsilon && time >
134     tiempo_min_simulacion || time >= tiempo_max_simulacion ){
135     FILE *file_out_u;
136     FILE *file_out_v;
137     FILE *file_out_p;
138     FILE *file_out_fi;
139     FILE *file_out_xmeshface, *file_out_xmeshnode,
140           *file_out_ymeshface, *file_out_ymeshnode;
141     file_out_u = fopen("u.txt", "w");
142     file_out_v = fopen("v.txt", "w");
143     file_out_p = fopen("p.txt", "w");
144     file_out_fi = fopen("fi.txt", "w");
145     file_out_xmeshface = fopen("x_mesh_face.txt", "w");
146     file_out_xmeshnode = fopen("x_mesh_node.txt", "w");
147     file_out_ymeshface = fopen("y_mesh_face.txt", "w");
148     file_out_ymeshnode = fopen("y_mesh_node.txt", "w");
149     for( j = 0; j <= mesh->Ny+1; j++ ){
150         for( i = 0; i <= mesh->Nx; i++ ){
151             fprintf(file_out_u, "% .9f ", (*un.u)(i,j)/U);
152         }
153         fprintf(file_out_u, "\n");
154     }
155     for( j = 0; j <= mesh->Ny; j++ ){
156         for( i = 0; i <= mesh->Nx+1; i++ ){
157             fprintf(file_out_v, "% .9f ", (*un.v)(i,j)/U);
158         }
159         fprintf(file_out_v, "\n");
160     }
161     for( j = 0; j <= mesh->Ny+1; j++ ){
162         for( i = 0; i <= mesh->Nx+1; i++ ){
163             if( i == 0 && j != 0 && j != mesh->Ny+1){
164                 fprintf(file_out_p, "% .9f ", p(i+1,j) / delta_t);
165             } else if( i == mesh->Nx+1 && j != 0 && j != mesh->Ny+1){
166                 fprintf(file_out_p, "% .9f ", p(i-1,j) / delta_t);
167             }
168         }
169     }

```

```

159         } else if( j == 0 && i != 0 && i != mesh->Nx+1){
160             fprintf(file_out_p, "%.9f ", p(i,j+1) / delta_t);
161         } else if( j == mesh->Ny+1 && i != 0 && i != mesh->Nx+1){
162             fprintf(file_out_p, "%.9f ", p(i,j-1) / delta_t);
163         } else if( i == 0 && j == 0 ){
164             fprintf(file_out_p, "%.9f ", p(1,1) / delta_t);
165         } else if( i == mesh->Nx+1 && j == 0 ){
166             fprintf(file_out_p, "%.9f ", p(mesh->Nx,1) / delta_t);
167         } else if( i == mesh->Nx+1 && j == mesh->Ny+1 ){
168             fprintf(file_out_p, "%.9f ", p(mesh->Nx,mesh->Ny) /
169                     delta_t);
170         } else if( i == 0 && j == mesh->Ny+1 ){
171             fprintf(file_out_p, "%.9f ", p(1,mesh->Ny) / delta_t);
172         } else {
173             fprintf(file_out_p, "%.9f ", p(i,j) / delta_t);
174         }
175     fprintf(file_out_p, "\n");
176 }
177 ScalarField fi(Nx+1, Ny+1, ZERO_TO_MINUS_ONE_N);
178 CalcStreamFunction( *mesh, un, fi );
179 for( j = 0; j <= mesh->Ny; j++ ){
180     for( i = 0; i <= mesh->Nx; i++ ){
181         fprintf(file_out_fi, "%.9f ", fi(i,j));
182     }
183     fprintf(file_out_fi, "\n");
184 }
185 double position;
186 position = 0.0;
187 for( i = 0; i <= mesh->Nx+1; i++ ){
188     if( i == 0 ){
189         fprintf(file_out_xmeshnode, "%.9f ", 0.0);
190     } else if( i == mesh->Nx+1 ){
191         fprintf(file_out_xmeshnode, "%.9f\n", Lx);
192     } else{
193         position += (*mesh->d_WP)(i,1);
194         fprintf(file_out_xmeshnode, "%.9f ", position);
195     }
196 }
197 position = 0.0;
198 for( i = 0; i <= mesh->Nx; i++ ){
199     if( i == 0 ){
200         fprintf(file_out_xmeshface, "%.9f ", 0.0);
201     } else if( i == mesh->Nx ){
202         fprintf(file_out_xmeshface, "%.9f\n", Lx);
203     } else{
204         position += (*mesh->delta_xP)(i,1);
205         fprintf(file_out_xmeshface, "%.9f ", position);
206     }
207 }
208 position = 0.0;
209 for( j = 0; j <= mesh->Ny+1; j++ ){
210     if( j == 0 ){
211         fprintf(file_out_ymeshnode, "%.9f ", 0.0);
212     } else if( j == mesh->Ny+1 ){
213         fprintf(file_out_ymeshnode, "%.9f\n", Ly);
214     } else{
215         position += (*mesh->d_SP)(1,j);
216         fprintf(file_out_ymeshnode, "%.9f ", position);
217     }
218 }
219 position = 0.0;
220 for( j = 0; j <= mesh->Ny; j++ ){

```

```

221     if( j == 0 ){
222         fprintf(file_out_ymeshface , " %.9f ", 0.0);
223     } else if( j == mesh->Ny ){
224         fprintf(file_out_ymeshface , " %.9f\n" , Ly);
225     } else{
226         position += (*mesh->delta_yP)(1,j);
227         fprintf(file_out_ymeshface , " %.9f " , position);
228     }
229 }
230 fclose(file_out_u);
231 fclose(file_out_v);
232 fclose(file_out_p);
233 fclose(file_out_fi);
234 fclose(file_out_data);
235 fclose(file_out_xmeshface);
236 fclose(file_out_xmeshnode);
237 fclose(file_out_ymeshface);
238 fclose(file_out_ymeshnode);
239 printf("Convergencia alcanzada o tiempo maximo de simulacion
    alcanzado.\n");
240 printf("Re: %d\nNx: %d\nNy: %d\n", Re, Nx, Ny);
241 printf("Error del solver: %e\nError en estacionario: %e\n",
    epsilon_solver , epsilon);
242 printf("Tiempo de computacion maximo por ciclo [ms]: %f\n" , ct_max)
    ;
243 printf("Tiempo de computacion minimo por ciclo [ms]: %f\n" , ct_min)
    ;
244 printf("Tiempo de computacion total [ms]: %f\n" , ct_total);
245 printf("Tiempo de computacion promedio por ciclo [ms]: %f\n",
    ct_total/(double)ciclos);
246 printf("Numero de ciclos: %d\n" , ciclos);
247 printf("Tiempo total de simulacion: %f\n" , time);
248 return 1;
249 }
250 }while( 1 );
251 return 0;
252 }
```

main\_1\_lbm.cpp

```

1 #include <cstdlib>
2 #include <math.h>
3 #include <stdio.h>
4 #include <iostream>
5 #include <iomanip>
6 #include <time.h>
7 #include "//FVM.h"
8 using namespace std;
9 class ComputationTime{
10     clock_t start;
11 public:
12     void Start(){
13         start = clock();
14     }
15     double Stop(){
16         return (((double) (clock() - start)) / CLOCKS_PER_SEC);
17     }
18 };
19 enum SOLVER {GAUSS_SHEIDEL = 1, CONJUGATE_GRADIENT};
20 int main(int argc, char** argv) {
21     SOLVER solver_type = CONJUGATE_GRADIENT;
22     MESH mesh_type = UNIFORM;
23     const double C = 0.015;
```

```

24 const double Lx = 1.0, Ly = 1.0;
25 const int Nx = 150, Ny = 150;
26 const double Re = 1000;
27 const double U = 1.0;
28 double time = 0.0;
29 int ciclos = 0;
30 double delta_t = 0;
31 int iteraciones = 0;
32 VelocityField un(Nx, Ny);
33 VelocityField un1(Nx, Ny);
34 VelocityField up(Nx, Ny);
35 VelocityField temp(Nx, Ny);
36 ScalarField p(Nx, Ny, ONE_TO_N);
37 ScalarField p_supuesta(Nx, Ny, ONE_TO_N);
38 ScalarField Rxn(Nx, Ny, ONE_TO_N);
39 ScalarField Ryn(Nx, Ny, ONE_TO_N);
40 ScalarField Rxn1(Nx, Ny, ONE_TO_N);
41 ScalarField Ryn1(Nx, Ny, ONE_TO_N);
42 ScalarField tmp_r(Nx, Ny, ONE_TO_N);
43 ScalarField tmp_p(Nx, Ny, ONE_TO_N);
44 ScalarField tmp_Ap(Nx, Ny, ONE_TO_N);
45 Mesh *mesh;
46 FILE *file_out_data = fopen("Data.txt", "w");
47 int i, j;
48 double convergence_u, convergence_v;
49 const double factor_relajacion_solver = 1.1;
50 const double epsilon_solver = 1E-8;
51 const double epsilon = 6E-4;
52 const double tiempo_max_simulacion = 200;
53 const double tiempo_min_simulacion = 0.05;
54 ComputationTime computation_time;
55 double ct_max = 0.0, ct_min = 1, ct_total = 0.0;
56 double ct_temporal = 0.0;
57 mesh = new Mesh(Lx, Ly, Nx, Ny, mesh_type);
58 mesh->C = C;
59 mesh->CalcMesh();
60 for( i = 1; i <= Nx-1; i++ ){
61     for( j = 1; j <= Ny; j++ ){
62         (*un1.u)(i,j) = 0.0;
63         (*un.u)(i,j) = 0.0;
64     }
65 }
66 for( i = 1; i <= Nx; i++ ){
67     for( j = 1; j <= Ny-1; j++ ){
68         (*un1.v)(i,j) = 0.0;
69         (*un.v)(i,j) = 0.0;
70     }
71 }
72 for( i = 0; i <= Nx; i++ ){
73     (*temp.u)(i,Ny+1) = U;
74 }
75 for( i = 1; i <= Nx; i++ ){
76     for( j = 1; j <= Ny; j++ ){
77         p(i,j) = 1.0;
78         p_supuesta(i,j) = 1.0;
79     }
80 }
81 do{
82     computation_time.Start();
83     delta_t = CalcTimeInterval( *mesh, un, 1.0/Re, .35*0.4, .2*0.4 );
84     CalcMomentumConvectiveDifusiveTerm_ForcedConvection( *mesh, un, Re,
Rxn, Ryn );

```

```

85 CalcMomentumConvectiveDifusiveTerm_ForcedConvection( *mesh, un1, Re,
86 Rxn1, Ryn1 );
87 for( i = 1; i <= mesh->Nx; i++ ){
88     for( j = 1; j <= mesh->Ny; j++ ){
89         if( i < mesh->Nx ){
90             (*up.u)(i,j) = (*un.u)(i,j) + delta_t * ( 1.5 * Rxn(i,j) -
91                                         0.5 * Rxn1(i,j) );
92         }
93         if( j < mesh->Ny ){
94             (*up.v)(i,j) = (*un.v)(i,j) + delta_t * ( 1.5 * Ryn(i,j) -
95                                         0.5 * Ryn1(i,j) );
96         }
97     }
98 }
99 for( i = 0; i <= Nx; i++ ){
100    (*up.u)(i,Ny+1) = U;
101 }
102 for( i = 0; i <= mesh->Nx+1; i++ ){
103     for( j = 0; j <= mesh->Ny+1; j++ ){
104         if( i <= mesh->Nx ){
105             (*un1.u)(i,j) = (*un.u)(i,j);
106         }
107         if( j <= mesh->Ny ){
108             (*un1.v)(i,j) = (*un.v)(i,j);
109         }
110     }
111 if( solver_type == GAUSS_SHEIDEL ){
112     iteraciones = CalcIncompressibleVelocityField_GaussSheidel( *mesh,
113                     up, un, p, p_supuesta, epsilon_solver,
114                     factor_relajacion_solver );
115 } else if( solver_type == CONJUGATE_GRADIENT ){
116     iteraciones = CalcIncompressibleVelocityField_ConjugateGradient( *
117                     mesh, up, un, p, epsilon_solver, tmp_r, tmp_p, tmp_Ap );
118 }
119 ct_temporal = computation_time.Stop();
120 if( ct_max < ct_temporal ) ct_max = ct_temporal;
121 if( ct_min > ct_temporal ) ct_min = ct_temporal;
122 ct_total += ct_temporal;
123 printf("Tiempo = %f (%d,%f,%f)", time, iteraciones, delta_t,
124           ct_temporal);
125 convergence_u = CalcMaxDerivative((*un.u), (*un1.u), delta_t);
126 convergence_v = CalcMaxDerivative((*un.v), (*un1.v), delta_t);
127 i = j = Nx/2;
128 fprintf(file_out_data, "%d %.9f %.9f %.9f %.9f %.9f %.9f %.9f\n",
129         ciclos, time, delta_t, ct_temporal, iteraciones, convergence_u,
130         convergence_v, (*un.u)(i,j) / U, (*un.v)(i,j) / U, p(i,j) /
131         delta_t );
132 printf("%f, %f}\n", convergence_u, convergence_v);
133 time += delta_t;
134 ciclos++;
135 if( convergence_u <= epsilon && convergence_v <= epsilon && time >
136     tiempo_min_simulacion || time >= tiempo_max_simulacion ){
137     FILE *file_out_u;
138     FILE *file_out_v;
139     FILE *file_out_p;
140     FILE *file_out_fi;
141     FILE *file_out_xmeshface, *file_out_xmeshnode,
142             *file_out_ymeshface, *file_out_ymeshnode;
143     file_out_u = fopen("u.txt", "w");
144     file_out_v = fopen("v.txt", "w");
145     file_out_p = fopen("p.txt", "w");
146     file_out_fi = fopen("fi.txt", "w");

```

```

137 file_out_xmeshface = fopen("x_mesh_face.txt", "w");
138 file_out_xmeshnode = fopen("x_mesh_node.txt", "w");
139 file_out_ymeshface = fopen("y_mesh_face.txt", "w");
140 file_out_ymeshnode = fopen("y_mesh_node.txt", "w");
141 for( j = 0; j <= mesh->Ny+1; j++ ){
142     for( i = 0; i <= mesh->Nx; i++ ){
143         fprintf(file_out_u, "%9f ", (*un.u)(i,j)/U);
144     }
145     fprintf(file_out_u, "\n");
146 }
147 for( j = 0; j <= mesh->Ny; j++ ){
148     for( i = 0; i <= mesh->Nx+1; i++ ){
149         fprintf(file_out_v, "%9f ", (*un.v)(i,j)/U);
150     }
151     fprintf(file_out_v, "\n");
152 }
153 for( j = 0; j <= mesh->Ny+1; j++ ){
154     for( i = 0; i <= mesh->Nx+1; i++ ){
155         if( i == 0 && j != 0 && j != mesh->Ny+1){
156             fprintf(file_out_p, "%9f ", p(i+1,j) / delta_t);
157         } else if( i == mesh->Nx+1 && j != 0 && j != mesh->Ny+1){
158             fprintf(file_out_p, "%9f ", p(i-1,j) / delta_t);
159         } else if( j == 0 && i != 0 && i != mesh->Nx+1){
160             fprintf(file_out_p, "%9f ", p(i,j+1) / delta_t);
161         } else if( j == mesh->Ny+1 && i != 0 && i != mesh->Nx+1){
162             fprintf(file_out_p, "%9f ", p(mesh->Nx,mesh->Ny) / delta_t);
163         } else if( i == 0 && j == 0 ){
164             fprintf(file_out_p, "%9f ", p(1,1) / delta_t);
165         } else if( i == mesh->Nx+1 && j == 0 ){
166             fprintf(file_out_p, "%9f ", p(mesh->Nx,1) / delta_t);
167         } else if( i == mesh->Nx+1 && j == mesh->Ny+1 ){
168             fprintf(file_out_p, "%9f ", p(mesh->Nx,mesh->Ny) / delta_t);
169         } else if( i == 0 && j == mesh->Ny+1 ){
170             fprintf(file_out_p, "%9f ", p(1,mesh->Ny) / delta_t);
171         } else {
172             fprintf(file_out_p, "%9f ", p(i,j) / delta_t);
173         }
174     }
175     fprintf(file_out_p, "\n");
176 }
177 ScalarField fi(Nx+1, Ny+1, ZERO_TO_MINUS_ONE_N);
178 CalcStreamFunction( *mesh, un, fi );
179 for( j = 0; j <= mesh->Ny; j++ ){
180     for( i = 0; i <= mesh->Nx; i++ ){
181         fprintf(file_out_fi, "%9f ", fi(i,j));
182     }
183     fprintf(file_out_fi, "\n");
184 }
185 double position;
186 position = 0.0;
187 for( i = 0; i <= mesh->Nx+1; i++ ){
188     if( i == 0 ){
189         fprintf(file_out_xmeshnode, "%9f ", 0.0);
190     } else if( i == mesh->Nx+1 ){
191         fprintf(file_out_xmeshnode, "%9f\n", Lx);
192     } else{
193         position += (*mesh->d_WP)(i,1);
194         fprintf(file_out_xmeshnode, "%9f ", position);
195     }
196 }
197 position = 0.0;
198 for( i = 0; i <= mesh->Nx; i++ ){

```

```

199     if( i == 0 ){
200         fprintf(file_out_xmeshface , " %.9f ", 0.0);
201     } else if( i == mesh->Nx ){
202         fprintf(file_out_xmeshface , " %.9f\n" , Lx);
203     } else{
204         position += (*mesh->delta_xP)(i,1);
205         fprintf(file_out_xmeshface , " %.9f " , position);
206     }
207     position = 0.0;
208     for( j = 0; j <= mesh->Ny+1; j++ ){
209         if( j == 0 ){
210             fprintf(file_out_ymeshnode , " %.9f " , 0.0);
211         } else if( j == mesh->Ny+1 ){
212             fprintf(file_out_ymeshnode , " %.9f\n" , Ly);
213         } else{
214             position += (*mesh->d_SP)(1,j);
215             fprintf(file_out_ymeshnode , " %.9f " , position);
216         }
217     }
218     position = 0.0;
219     for( j = 0; j <= mesh->Ny; j++ ){
220         if( j == 0 ){
221             fprintf(file_out_ymeshface , " %.9f " , 0.0);
222         } else if( j == mesh->Ny ){
223             fprintf(file_out_ymeshface , " %.9f\n" , Ly);
224         } else{
225             position += (*mesh->delta_yP)(1,j);
226             fprintf(file_out_ymeshface , " %.9f " , position);
227         }
228     }
229     fclose(file_out_u);
230     fclose(file_out_v);
231     fclose(file_out_p);
232     fclose(file_out_fi);
233     fclose(file_out_data);
234     fclose(file_out_xmeshface);
235     fclose(file_out_xmeshnode);
236     fclose(file_out_ymeshface);
237     fclose(file_out_ymeshnode);
238     printf("Convergencia alcanzada o tiempo maximo de simulacion
239         alcanzado.\n");
240     printf("Re: %d\nNx: %d\nNy: %d\n", Re, Nx, Ny);
241     printf("Error del solver: %e\nError en estacionario: %e\n",
242           epsilon_solver, epsilon);
243     printf("Tiempo de computacion maximo por ciclo [ms]: %f\n", ct_max)
244     ;
245     printf("Tiempo de computacion minimo por ciclo [ms]: %f\n", ct_min)
246     ;
247     printf("Tiempo de computacion total [ms]: %f\n", ct_total);
248     printf("Tiempo de computacion promedio por ciclo [ms]: %f\n",
249           ct_total/(double)ciclos);
250     printf("Numero de ciclos: %d\n", ciclos);
251     printf("Tiempo total de simulacion: %f\n", time);
252     return 1;
253 }
254 }while( 1 );
255 return 0;
}

```

main\_1\_fvm.cpp

## Anexo D

# Código fuente de Differentially Heated Cavity

```
1 #include <cstdlib>
2 #include <stdio.h>
3 #include <math.h>
4 #include <time.h>
5 #include "../LBM.h"
6 using namespace std;
7 int main(int argc, char** argv) {
8     LatticeModel lattice_model_momentum(2, 9,(double []){ 4.0/9.0, 1.0/9.0,
9         1.0/9.0, 1.0/9.0, 1.0/9.0, 1.0/36.0, 1.0/36.0, 1.0/36.0, 1.0/36.0 },
10        (int []){ 0, 1, 0, -1, 0, 1, -1, -1, 1 }, (int []){ 0, 0, 1, 0, -1, 1, 1,
11        -1, -1 }, NULL, (int []){ 0, 3, 4, 1, 2, 7, 8, 5, 6}, 1.0/sqrt(3.0));
12     LatticeModel lattice_model_energy(2, 9,(double []){ 4.0/9.0, 1.0/9.0,
13         1.0/9.0, 1.0/9.0, 1.0/9.0, 1.0/36.0, 1.0/36.0, 1.0/36.0, 1.0/36.0 },
14        (int []){ 0, 1, 0, -1, 0, 1, -1, -1, 1 }, (int []){ 0, 0, 1, 0, -1, 1, 1,
15        -1, -1 }, NULL, (int []){ 0, 3, 4, 1, 2, 7, 8, 5, 6}, 1.0/sqrt(3.0));
16     Simulation *simulation;
17     int H = 40;
18     int Nx = H, Ny = H;
19     const double Ra = 1E3;
20     const double Pr = 0.71;
21     const double Thot = 1.0;
22     const double Tcold = 0.0;
23     double nu = 0.05;
24     double a;
25     double omega_momentum, omega_energy;
26     double Ma;
27     int it_output = 1;
28     int it_movie_output = 100000;
29     int i, j, alfa;
30     double epsilon = 1E-4;
31     double U_caracteristica;
32     double U_adimensionalizacion;
33     double gravitybeta;
34     FILE *output_u = NULL;
35     FILE *output_v = NULL;
36     FILE *output_rho = NULL;
37     FILE *output_T = NULL;
38     FILE *output_pressure = NULL;
39     FILE *output_velocity = NULL;
40     FILE *output_data = NULL;
41     FILE *output_info = NULL;
```

```

36 FILE *output_Nu = NULL;
37 char output_file_name[100];
38 char keyboard;
39 double u_old, v_old, rho_old, T_old;
40 double ct_max = 0.0, ct_min = 1, ct_total = 0.0;
41 double ct_temporal = 0.0;
42 a = nu / Pr;
43 U_adimensionalizacion = a / (double) Ny;
44 U_caracteristica = U_adimensionalizacion * sqrt(Ra*Pr*(Thot-Tcold));
45 gravitybeta = pow(U_caracteristica,2.0) / ((Thot-Tcold) * Ny );
46 omega_momentum = 1.0 / ( 3.0*nu + 0.5 );
47 omega_energy = 1.0 / ( 3.0*a + 0.5 );
48 Ma = U_caracteristica / lattice_model_momentum.cs;
49 printf("Parametros de simulacion\nRa = %\nPr = %\nU caracteristica = %\n
      nU adimensionalizacion = %\nMa = %\nnu = %\na = %\nomega momentum
      = %\nomega energy = %\n",
50           Ra, Pr, U_caracteristica, U_adimensionalizacion, Ma,
51           nu, a, omega_momentum, omega_energy);
52 output_data = fopen("Data.txt", "w");
53 output_info = fopen("info.txt", "w");
54 fprintf(output_info, "%d %d %f %f %f %f %f %f %f", Nx, Ny,
55         U_caracteristica, U_adimensionalizacion, nu, a, omega_momentum,
56         omega_energy, Ra, Ma, Pr );
57 fclose(output_info);
58 simulation = new Simulation( Nx, Ny, &lattice_model_momentum, &
59     lattice_model_energy, omega_momentum, omega_energy, gravitybeta);
60 simulation->InitEquilibrium( 1.0, (Thot + Tcold)*0.5 );
61 u_old = v_old = 0;
62 rho_old = 1.0;
63 T_old = (Thot + Tcold)*0.5;
64 simulation->AddBoundaryCondition( BoundaryCondition( NORTH_VELOCITY_ZOU_HE
65     , Point(1,Ny-1), Point(Nx-2,Ny-1), 0, 0.0, 0.0,0.0));
66 simulation->AddBoundaryCondition( BoundaryCondition( SOUTH_VELOCITY_ZOU_HE
67     , Point(1,0), Point(Nx-2,0), 0.0, 0.0, 0.0,0.0));
68 simulation->AddBoundaryCondition( BoundaryCondition( WEST_VELOCITY_ZOU_HE,
69     Point(0,1), Point(0,Ny-2), 0.0, 0.0, 0.0,0.0));
70 simulation->AddBoundaryCondition( BoundaryCondition( EAST_VELOCITY_ZOU_HE,
71     Point(Nx-1,1), Point(Nx-1,Ny-2), 0.0, 0.0, 0.0,0.0));
72 simulation->AddBoundaryCondition( BoundaryCondition(
73     NORTH_WEST_CONCAVE_CORNER_VELOCITY_ZOU_HE, Point(0,Ny-1), Point(0,Ny
74     -1), 0.0, 0.0, 0.0,0.0));
75 simulation->AddBoundaryCondition( BoundaryCondition(
76     NORTH_EAST_CONCAVE_CORNER_VELOCITY_ZOU_HE, Point(Nx-1,Ny-1), Point(Nx
77     -1,Ny-1), 0.0, 0.0, 0.0,0.0));
78 simulation->AddBoundaryCondition( BoundaryCondition(
79     SOUTH_WEST_CONCAVE_CORNER_VELOCITY_ZOU_HE, Point(0,0), Point(0,0),
80     0.0, 0.0, 0.0,0.0));
81 simulation->AddBoundaryCondition( BoundaryCondition(
82     SOUTH_EAST_CONCAVE_CORNER_VELOCITY_ZOU_HE, Point(Nx-1,0), Point(Nx
83     -1,0), 0.0, 0.0, 0.0,0.0));
84 simulation->AddBoundaryCondition( BoundaryCondition( SOUTHADIABATIC,
85     Point(1,0), Point(Nx-2,0), 0.0, 0.0, 0.0,0.0));
86 simulation->AddBoundaryCondition( BoundaryCondition( NORTHADIABATIC,
87     Point(1,Ny-1), Point(Nx-2,Ny-1), 0.0, 0.0, 0.0,0.0));
88 simulation->AddBoundaryCondition( BoundaryCondition( WEST_TEMPERATURE,
89     Point(0,0), Point(0,Ny-1), 0.0, 0.0, 0.0, Thot));
90 simulation->AddBoundaryCondition( BoundaryCondition( EAST_TEMPERATURE,
91     Point(Nx-1,0), Point(Nx-1,Ny-1), 0.0, 0.0, 0.0, Tcold));
92 while ( 1 ){
93     ct_temporal = simulation->Tick();
94     if( ct_max < ct_temporal ) ct_max = ct_temporal;
95     if( ct_min > ct_temporal ) ct_min = ct_temporal;
96     ct_total += ct_temporal;

```

```

77     simulation->convergence_u == pow(Ny,2.0) / (a*U_adimensionalizacion);
78     simulation->convergence_v == pow(Ny,2.0) / (a*U_adimensionalizacion);
79     printf("Tiempo: %d (%.11f %.11f %.11f %.11f)\n", simulation->time,
80           simulation->convergence_rho, simulation->convergence_u, simulation
81           ->convergence_v, simulation->convergence_T);
82     i = j = (Nx-1)*0.5;
83     fprintf(output_data, "%d %.9f %.9f %.9f %.9f %.9f %.9f %.9f %.9f %.9f
84           %.9f %.9f %.9f %.9f %.9f %.9f %.9f %.9f %.9f %.9f\n",
85           simulation->time, (double)(simulation->time) / (pow(Ny,2.0)/a)
86           , ct_temporal, simulation->convergence_u, simulation->
87           convergence_v, simulation->convergence_rho,
88           simulation->domain[ i ][ j ].u / U_adimensionalizacion ,
89           simulation->domain[ i ][ j ].v / U_adimensionalizacion ,
90           simulation->domain[ i ][ j ].rho*pow(simulation->lattice_model_momentum->cs,2.0),
91           simulation->domain[ i ][ j ].u - u_old, simulation->domain[ i
92           ][ j ].v - v_old, simulation->domain[ i ][ j ].rho -
93           rho_old,
94           simulation->total_density, simulation->total_density_u ,
95           simulation->total_density_v );
96     u_old = simulation->domain[ i ][ j ].u;
97     v_old = simulation->domain[ i ][ j ].v;
98     rho_old = simulation->domain[ i ][ j ].rho;
99     T_old = simulation->domain[ i ][ j ].T;
100    if( (simulation->time % it_output) == 0 ){
101        output_u = fopen("u.txt","w");
102        output_v = fopen("v.txt","w");
103        output_T = fopen("T.txt","w");
104        output_velocity = fopen("velocity.txt","w");
105        output_rho = fopen("rho.txt","w");
106        output_pressure = fopen("pressure.txt","w");
107        for( j = 0; j < Ny; j++ ){
108            for( i = 0; i < Nx; i++ ){
109                fprintf(output_u, "%f ", simulation->domain[ i ][ j ].u /
110                    U_adimensionalizacion );
111                fprintf(output_v, "%f ", simulation->domain[ i ][ j ].v /
112                    U_adimensionalizacion );
113                fprintf(output_T, "%f ", simulation->domain[ i ][ j ].T);
114                fprintf(output_velocity, "%f ", sqrt(simulation->domain[ i
115                    ][ j ].u*simulation->domain[ i ][ j ].u+simulation->
116                    domain[ i ][ j ].v*simulation->domain[ i ][ j ].v) /
117                    U_adimensionalizacion );
118                fprintf(output_rho, "%f ", simulation->domain[ i ][ j ].rho );
119                fprintf(output_pressure, "%f ", simulation->domain[ i ][ j ].rho/3.0 );
120            }
121            fprintf(output_u, "\n");
122            fprintf(output_v, "\n");
123            fprintf(output_T, "\n");
124            fprintf(output_velocity, "\n");
125            fprintf(output_rho, "\n");
126            fprintf(output_pressure, "\n");
127        }
128        fclose( output_u );
129        fclose( output_v );
130        fclose( output_T );
131        fclose( output_velocity );
132        fclose( output_rho );
133        fclose( output_pressure );
134        printf("Archivos de salida generados.\n 1. Pulse 'Enter' para
135             continuar.\n 2. Pulse 'E' para salir.\n 3. Pulse 'C' para
136             cambiar el intervalo de tiempo de muestreo.\n > ");

```

```

120 do {
121     keyboard = getchar();
122     switch( keyboard ){
123         case 'E':
124             fclose( output_data );
125             return 0;
126         case 'C':
127             printf("\nEl intervalo actual de muestreo es %d, por
128                 favor introduzca el nuevo: ", it_output);
129             scanf("%d", &it_output );
130             break;
131     } while ( keyboard != '\n' && keyboard != 'C' );
132 }
133 if( (simulation->time % it_movie_output) == 0 ){
134     sprintf(output_file_name, "movie//u%d.txt", simulation->time );
135     output_u = fopen(output_file_name, "w");
136     sprintf(output_file_name, "movie//v%d.txt", simulation->time );
137     output_v = fopen(output_file_name, "w");
138     sprintf(output_file_name, "movie//rho%d.txt", simulation->time );
139     output_rho = fopen(output_file_name, "w");
140     sprintf(output_file_name, "movie//T%d.txt", simulation->time );
141     output_T = fopen(output_file_name, "w");
142     for( j = 0; j < Ny; j++){
143         for( i = 0; i < Nx; i++){
144             fprintf(output_u, "%f ", simulation->domain[ i ][ j ].u /
145                 U_adimensionalizacion );
146             fprintf(output_v, "%f ", simulation->domain[ i ][ j ].v /
147                 U_adimensionalizacion );
148             fprintf(output_rho, "%f ", simulation->domain[ i ][ j ].rho );
149             fprintf(output_T, "%f ", simulation->domain[ i ][ j ].T );
150         }
151         fprintf(output_u, "\n");
152         fprintf(output_v, "\n");
153         fprintf(output_rho, "\n");
154         fprintf(output_T, "\n");
155     }
156     fclose( output_u );
157     fclose( output_v );
158     fclose( output_rho );
159     fclose( output_T );
160 }
161 if( simulation->time >= 3*H && simulation->convergence_u <= epsilon &&
162     simulation->convergence_v <= epsilon ){
163     printf("Convergencia alcanzada.\n");
164     printf("Ra: %f\nPr: %f\nNx: %d\nNy: %d\n", Ra, Pr, Nx, Ny);
165     printf("Error en estacionario: %f\n", epsilon);
166     printf("U caracteristica: %f\nU adimensionalizacion: %f\nMa: %f\n
167         nnu: %f\na: %f\nomega momentum: %f\nomega energy: %f\n",
168         U_caracteristica, U_adimensionalizacion, Ma, nu, a,
169         omega_momentum, omega_energy);
170     printf("Tiempo de computacion maximo por ciclo [ms]: %f\n", ct_max)
171     ;
172     printf("Tiempo de computacion minimo por ciclo [ms]: %f\n", ct_min)
173     ;
174     printf("Tiempo de computacion total [ms]: %f\n", ct_total);
175     printf("Tiempo de computacion promedio por ciclo [ms]: %f\n",
176         ct_total/(double)simulation->time);
177     printf("Numero de ciclos: %d\n", simulation->time);
178     printf("Tiempo de simulacion: %f\n", (double)(simulation->time) /
179         (pow(Ny,2.0)/a));
180     output_u = fopen("u.txt", "w");

```

```

171     output_v = fopen("v.txt", "w");
172     output_T = fopen("T.txt", "w");
173     output_velocity = fopen("velocity.txt", "w");
174     output_rho = fopen("rho.txt", "w");
175     output_pressure = fopen("pressure.txt", "w");
176     output_Nu = fopen("Nu.txt", "w");
177     for( j = 0; j < Ny; j++ ){
178         for( i = 0; i < Nx; i++ ){
179             fprintf(output_u, "%f ", simulation->domain[ i ][ j ].u /
180                 U_adimensionalizacion );
181             fprintf(output_v, "%f ", simulation->domain[ i ][ j ].v /
182                 U_adimensionalizacion );
183             fprintf(output_T, "%f ", simulation->domain[ i ][ j ].T );
184             fprintf(output_velocity, "%f ", sqrt(simulation->domain[ i ][ j ].u*simulation->
185                 domain[ i ][ j ].v+simulation->domain[ i ][ j ].v*simulation->
186                 domain[ i ][ j ].v) / U_adimensionalizacion );
187             fprintf(output_rho, "%f ", simulation->domain[ i ][ j ].rho );
188             fprintf(output_pressure, "%f ", simulation->domain[ i ][ j ].rho/3.0 );
189         }
190         fprintf(output_u, "\n");
191         fprintf(output_v, "\n");
192         fprintf(output_T, "\n");
193         fprintf(output_velocity, "\n");
194         fprintf(output_rho, "\n");
195         fprintf(output_pressure, "\n");
196     }
197     fclose( output_u );
198     fclose( output_v );
199     fclose( output_T );
200     fclose( output_velocity );
201     fclose( output_rho );
202     fclose( output_pressure );
203     fclose( output_data );
204     double Nu_medio = 0.0;
205     double Nu_i;
206     double Nu_min_hot, Nu_min_cold;
207     double Nu_min_hot_y, Nu_min_cold_y;
208     double Nu_max_hot, Nu_max_cold;
209     double Nu_max_hot_y, Nu_max_cold_y;
210     double position;
211     double Nu_local;
212     double Dx,Dy;
213     Dx = 1.0 / ((double)Nx-1.0);
214     Dy = 1.0 / ((double)Ny-1.0);
215     for( i = 1; i <= Nx-3; i++ ){
216         Nu_i = 0.0;
217         if( i == 1 ){
218             Nu_min_hot = 1000;
219             Nu_max_hot = -1000;
220             position = 0.0;
221             for( j = 0; j <= Ny-1; j++ ){
222                 Nu_local = (-1) * (simulation->domain[ i+1 ][ j ].T-
223                     simulation->domain[ i ][ j ].T)*((double)Nx-1.0) +
224                     0.5*(simulation->domain[ i+1 ][ j ].u+simulation-
225                         >domain[ i ][ j ].u)*0.5*(simulation->domain[ i+1 ]
226                         ][ j ].T+simulation->domain[ i ][ j ].T) /
227                         U_adimensionalizacion ;
228             if( Nu_local > Nu_max_hot){
229                 Nu_max_hot = Nu_local;
230                 Nu_max_hot_y = position;

```

```

222 }
223 if( Nu_local < Nu_min_hot){
224     Nu_min_hot = Nu_local;
225     Nu_min_hot_y = position;
226 }
227 position += Dy;
228 Nu_i += Nu_local;
229 }
230 }else if( i == (Nx-3) ){
231     Nu_min_cold = 1000;
232     Nu_max_cold = -1000;
233     position = 0.0;
234     for( j = 0; j <= Ny-1; j++ ){
235         Nu_local = (-1)* ( simulation->domain[ i+1 ][ j ].T-
236             simulation->domain[ i ][ j ].T)*((double)Nx-1.0) +
237             0.5*( simulation->domain[ i+1 ][ j ].u+simulation
238             ->domain[ i ][ j ].u)*0.5*( simulation->domain[ i+1
239             ][ j ].T+simulation->domain[ i ][ j ].T) /
240             U_adimensionalizacion;
241         if( Nu_local > Nu_max_cold){
242             Nu_max_cold = Nu_local;
243             Nu_max_cold_y = position;
244         }
245         position += Dy;
246         Nu_i += Nu_local;
247     }
248 }else {
249     for( j = 0; j <= Ny-1; j++ ){
250         Nu_local = (-1) * (simulation->domain[ i+1 ][ j ].T-
251             simulation->domain[ i ][ j ].T)*((double)Nx-1.0) +
252             0.5*( simulation->domain[ i+1 ][ j ].u+simulation
253             ->domain[ i ][ j ].u)*0.5*( simulation->domain[ i+1
254             ][ j ].T+simulation->domain[ i ][ j ].T) /
255             U_adimensionalizacion;
256         Nu_i += Nu_local;
257     }
258     Nu_i /= (double)Ny;
259     Nu_medio += Nu_i;
260     printf("Nu(%d)=%.6f\n", i, Nu_i);
261     fprintf(output_Nu, "%.9f\n", Nu_i);
262 }
263 printf("Nu medio=%.6f\n", Nu_medio/(Nx-3));
264 printf("Para la pared caliente: \n");
265 printf("Nu max = %.6f (y=%f)\n", Nu_max_hot, Nu_max_hot_y);
266 printf("Nu min = %.6f (y=%f)\n", Nu_min_hot, Nu_min_hot_y);
267 printf("Para la pared fria: \n");
268 printf("Nu max = %.6f (y=%f)\n", Nu_max_cold, Nu_max_cold_y);
269 printf("Nu min = %.6f (y=%f)\n", Nu_min_cold, Nu_min_cold_y);
270 double max_u = -1000, max_v = -1000, max_u_y, max_v_x;
271 position = 0.0;
272 for( j = 0; j < Ny; j++ ){
273     for( i = 0; i < Nx; i++ ){
274         if( (simulation->domain[ i ][ j ].u /
275             U_adimensionalizacion) > max_u ){
276             max_u = simulation->domain[ i ][ j ].u /
277                 U_adimensionalizacion;
278             max_u_y = position;
279     }

```

```

273         }
274         position += Dx;
275     }
276     position = 0.0;
277     for( i = 0; i < Nx; i++ ){
278         for( j = 0; j < Ny; j++ ){
279             if( (simulation->domain[ i ][ j ].v /
280                 U_adimensionalizacion) > max_v ){
281                 max_v = simulation->domain[ i ][ j ].v /
282                     U_adimensionalizacion;
283                 max_v_x = position;
284             }
285             position += Dx;
286         }
287         printf("\nu max = %.6f (y=%d)\n", max_u, max_u_y);
288         printf("v max = %.6f (x=%d)\n", max_v, max_v_x);
289         fclose(output_Nu);
290         return 1;
291     }
292     return 0;
293 }
```

main\_2\_llbm.cpp

```

1 #include <cstdlib>
2 #include <math.h>
3 #include <stdio.h>
4 #include <iostream>
5 #include <iomanip>
6 #include <time.h>
7 #include "//FVM.h"
8 using namespace std;
9 class ComputationTime{
10     clock_t start;
11 public:
12     void Start(){
13         start = clock();
14     }
15     double Stop(){
16         return (((double) (clock() - start)) / CLOCKS_PER_SEC);
17     }
18 };
19 enum SOLVER {GAUSS_SHEIDEL = 1, CONJUGATE_GRADIENT};
20 int main(int argc, char** argv) {
21     SOLVER solver_type = CONJUGATE_GRADIENT;
22     MESH mesh_type = UNIFORM;
23     const double C = 0.06;
24     const double Lx = 1.0, Ly = 1.0;
25     const int Nx = 40, Ny = 40;
26     const double Pr = 0.71;
27     const double Ra = 1E3;
28     const double T_H = 1;
29     const double T_C = 0;
30     double time = 0.0;
31     int ciclos = 0;
32     double delta_t = 0;
33     int iteraciones = 0;
34     double Nu_i, Nu_medio;
35     VelocityField un(Nx, Ny);
36     VelocityField un1(Nx, Ny);
37     VelocityField up(Nx, Ny);
```

```

38 VelocityField temp(Nx, Ny);
39 ScalarField p(Nx, Ny, ONE_TO_N);
40 ScalarField p_supuesta(Nx, Ny, ONE_TO_N);
41 ScalarField Rxn(Nx, Ny, ONE_TO_N);
42 ScalarField Ryn(Nx, Ny, ONE_TO_N);
43 ScalarField Rxn1(Nx, Ny, ONE_TO_N);
44 ScalarField Ryn1(Nx, Ny, ONE_TO_N);
45 ScalarField Qn(Nx, Ny, ONE_TO_N);
46 ScalarField Qn1(Nx, Ny, ONE_TO_N);
47 ScalarField Tn(Nx+2, Ny+2, ZERO_MINUS_ONE_N);
48 ScalarField Tn1(Nx+2, Ny+2, ZERO_MINUS_ONE_N);
49 ScalarField tmp_r(Nx, Ny, ONE_TO_N);
50 ScalarField tmp_p(Nx, Ny, ONE_TO_N);
51 ScalarField tmp_Ap(Nx, Ny, ONE_TO_N);
52 Mesh *mesh;
53 FILE *file_out_data = fopen("Data.txt", "w");
54 int i, j;
55 double convergence_u, convergence_v, convergence_T;
56 const double factor_relajacion_solver = 1.1;
57 const double epsilon_solver = 1E-8;
58 const double epsilon = 1E-4;
59 const double tiempo_max_simulacion = 10;
60 const double tiempo_min_simulacion = 0.01;
61 ComputationTime computation_time;
62 double ct_max = 0.0, ct_min = 1, ct_total = 0.0;
63 double ct_temporal = 0.0;
64 mesh = new Mesh(Lx, Ly, Nx, Ny, mesh_type);
65 mesh->C = C;
66 mesh->CalcMesh();
67 for( i = 1; i <= Nx-1; i++ ){
68     for( j = 1; j <= Ny; j++ ){
69         (*un1.u)(i,j) = 0.0;
70         (*un.u)(i,j) = 0.0;
71     }
72 }
73 for( i = 1; i <= Nx; i++ ){
74     for( j = 1; j <= Ny-1; j++ ){
75         (*un1.v)(i,j) = 0.0;
76         (*un.v)(i,j) = 0.0;
77     }
78 }
79 for( i = 1; i <= Nx; i++ ){
80     for( j = 1; j <= Ny; j++ ){
81         p(i,j) = 1.0;
82         p_supuesta(i,j) = 1.0;
83     }
84 }
85 for( i = 0; i <= Nx+1; i++ ){
86     for( j = 0; j <= Ny+1; j++ ){
87         Tn(i,j) = 0.5;
88         Tn1(i,j) = 0.5;
89     }
90 }
91 for( i = 1; i <= mesh->Nx; i++ ){
92     Tn(i,0) = Tn(i,1);
93     Tn1(i,0) = Tn1(i,1);
94     Tn(i,mesh->Ny+1) = Tn(i,mesh->Ny);
95     Tn1(i,mesh->Ny+1) = Tn1(i,mesh->Ny);
96 }
97 for( j = 0; j <= mesh->Ny+1; j++ ){
98     Tn(0,j) = T_H;
99     Tn1(0,j) = T_H;
100    Tn(Nx+1,j) = T_C;

```

```

101     Tn1(Nx+1,j) = T_C;
102 }
103 do{
104     computation_time.Start();
105     delta_t = CalcTimeInterval( *mesh, un, Pr, .35*0.4, .2*0.4 );
106     CalcMomentumConvectiveDifusiveTerm_NaturalConvection( *mesh, un, Tn,
107         Pr, Ra, Rxn, Ryn );
108     CalcMomentumConvectiveDifusiveTerm_NaturalConvection( *mesh, un1, Tn1,
109         Pr, Ra, Rxn1, Ryn1 );
110     CalcEnergyConvectiveDifusiveTerm_NaturalConvection( *mesh, un, Tn, Qn
111         );
112     CalcEnergyConvectiveDifusiveTerm_NaturalConvection( *mesh, un1, Tn1,
113         Qn1 );
114     for( i = 1; i <= mesh->Nx; i++ ){
115         for( j = 1; j <= mesh->Ny; j++ ){
116             if( i < mesh->Nx ){
117                 (*up.u)(i,j) = (*un.u)(i,j) + delta_t * ( 1.5 * Rxn(i,j) -
118                     0.5 * Rxn1(i,j) );
119             }
120             if( j < mesh->Ny ){
121                 (*up.v)(i,j) = (*un.v)(i,j) + delta_t * ( 1.5 * Ryn(i,j) -
122                     0.5 * Ryn1(i,j) );
123             }
124         }
125     }
126     for( i = 0; i <= mesh->Nx+1; i++ ){
127         for( j = 0; j <= mesh->Ny+1; j++ ){
128             if( i <= mesh->Nx ){
129                 (*un1.u)(i,j) = (*un.u)(i,j);
130             }
131             if( j <= mesh->Ny ){
132                 (*un1.v)(i,j) = (*un.v)(i,j);
133             }
134         }
135     }
136     if( solver_type == GAUSS_SHEIDEL ){
137         iteraciones = CalcIncompressibleVelocityField_GaussSheidel( *mesh,
138             up, un, p, p_supuesta, epsilon_solver,
139             factor_relajacion_solver );
140     } else if( solver_type == CONJUGATE_GRADIENT ){
141         iteraciones = CalcIncompressibleVelocityField_ConjugateGradient( *
142             mesh, up, un, p, epsilon_solver, tmp_r, tmp_p, tmp_Ap );
143     }
144     for( i = 1; i <= mesh->Nx; i++ ){
145         for( j = 1; j <= mesh->Ny; j++ ){
146             Tn(i,j) = Tn1(i,j) + delta_t * ( 1.5 * Qn(i,j) - 0.5 * Qn1(i,j)
147                 );
148         }
149     }
150     for( i = 1; i <= mesh->Nx; i++ ){
151         Tn(i,0) = Tn(i,1);
152         Tn(i,mesh->Ny+1) = Tn(i,mesh->Ny);
153     }
154     for( j = 0; j <= mesh->Ny+1; j++ ){
155         Tn(0,j) = T_H;
156         Tn(Nx+1,j) = T_C;
157     }
158     ct_temporal = computation_time.Stop();

```

```

154     if( ct_max < ct_temporal ) ct_max = ct_temporal;
155     if( ct_min > ct_temporal ) ct_min = ct_temporal;
156     ct_total += ct_temporal;
157     printf("Tiempo = %f (%d,%f,%f)", time, iteraciones, delta_t,
158           ct_temporal);
159     convergence_u = CalcMaxDerivative((*un.u), (*un1.u), delta_t);
160     convergence_v = CalcMaxDerivative((*un.v), (*un1.v), delta_t);
161     convergence_T = CalcMaxDerivative(Tn, Tn1, delta_t);
162     i = j = Nx/2;
163     fprintf(file_out_data, "%d %.9f %.9f %.9f %d %.9f %.9f %.9f %.9f
164           %.9f\n",
165           ciclos, time, delta_t, ct_temporal, iteraciones, convergence_u
166           ,
167           convergence_v, convergence_T, (*un.u)(i,j), (*un.v)(i,j), p(i,
168           j) / delta_t );
169     printf(" {%, %f, %f}\n", convergence_u, convergence_v, convergence_T)
170     ;
171     time += delta_t;
172     ciclos++;
173     if( convergence_u <= epsilon && convergence_v <= epsilon && time >
174         tiempo_min_simulacion || time >= tiempo_max_simulacion ){
175         FILE *file_out_u;
176         FILE *file_out_v;
177         FILE *file_out_p;
178         FILE *file_out_T;
179         FILE *file_out_Nu;
180         FILE *file_out_fi;
181         FILE *file_out_xmeshface, *file_out_xmeshnode,
182             *file_out_ymeshface, *file_out_ymeshnode;
183         file_out_u = fopen("u.txt", "w");
184         file_out_v = fopen("v.txt", "w");
185         file_out_p = fopen("p.txt", "w");
186         file_out_T = fopen("T.txt", "w");
187         file_out_Nu = fopen("Nu.txt", "w");
188         file_out_fi = fopen("fi.txt", "w");
189         file_out_xmeshface = fopen("x_mesh_face.txt", "w");
190         file_out_xmeshnode = fopen("x_mesh_node.txt", "w");
191         file_out_ymeshface = fopen("y_mesh_face.txt", "w");
192         file_out_ymeshnode = fopen("y_mesh_node.txt", "w");
193         for( j = 0; j <= mesh->Ny+1; j++ ){
194             for( i = 0; i <= mesh->Nx; i++ ){
195                 fprintf(file_out_u, "% .9f ", (*un.u)(i,j));
196             }
197             fprintf(file_out_u, "\n");
198         }
199         for( j = 0; j <= mesh->Ny; j++ ){
200             for( i = 0; i <= mesh->Nx+1; i++ ){
201                 fprintf(file_out_v, "% .9f ", (*un.v)(i,j));
202             }
203             fprintf(file_out_v, "\n");
204         }
205         for( j = 0; j <= mesh->Ny+1; j++ ){
206             for( i = 0; i <= mesh->Nx+1; i++ ){
207                 if( i == 0 && j != 0 && j != mesh->Ny+1){
208                     fprintf(file_out_p, "% .9f ", p(i+1,j) / delta_t);
209                 } else if( i == mesh->Nx+1 && j != 0 && j != mesh->Ny+1){
210                     fprintf(file_out_p, "% .9f ", p(i-1,j) / delta_t);
211                 } else if( j == 0 && i != 0 && i != mesh->Nx+1){
212                     fprintf(file_out_p, "% .9f ", p(i,j+1) / delta_t);
213                 } else if( j == mesh->Ny+1 && i != 0 && i != mesh->Nx+1){
214                     fprintf(file_out_p, "% .9f ", p(i,j-1) / delta_t);
215                 } else if( i == 0 && j == 0 ){
216                     fprintf(file_out_p, "% .9f ", p(1,1) / delta_t);
217                 }
218             }
219         }
220     }

```

```

211 } else if( i == mesh->Nx+1 && j == 0 ){
212     fprintf(file_out_p , "%.9f ", p(mesh->Nx,1) / delta_t);
213 } else if( i == mesh->Nx+1 && j == mesh->Ny+1 ){
214     fprintf(file_out_p , "%.9f ", p(mesh->Nx,mesh->Ny) / delta_t);
215 } else if( i == 0 && j == mesh->Ny+1 ){
216     fprintf(file_out_p , "%.9f ", p(1,mesh->Ny) / delta_t);
217 } else {
218     fprintf(file_out_p , "%.9f ", p(i,j) / delta_t);
219 }
220 }
221 fprintf(file_out_p , "\n");
222 }
223 for( j = 0; j <= mesh->Ny+1; j++ ){
224     for( i = 0; i <= mesh->Nx+1; i++ ){
225         fprintf(file_out_T , "%.9f ", Tn(i,j));
226     }
227     fprintf(file_out_T , "\n");
228 }
229 ScalarField fi(Nx+1, Ny+1, ZERO_TO_MINUS_ONE_N);
230 CalcStreamFunction( *mesh, un, fi );
231 for( j = 0; j <= mesh->Ny; j++ ){
232     for( i = 0; i <= mesh->Nx; i++ ){
233         fprintf(file_out_fi , "%.9f ", fi(i,j));
234     }
235     fprintf(file_out_fi , "\n");
236 }
237 double position;
238 position = 0.0;
239 for( i = 0; i <= mesh->Nx+1; i++ ){
240     if( i == 0 ){
241         fprintf(file_out_xmeshnode , "%.9f ", 0.0);
242     } else if( i == mesh->Nx+1 ){
243         fprintf(file_out_xmeshnode , "%.9f\n", Lx);
244     } else{
245         position += (*mesh->d_WP)(i,1);
246         fprintf(file_out_xmeshnode , "%.9f ", position);
247     }
248 }
249 position = 0.0;
250 for( i = 0; i <= mesh->Nx; i++ ){
251     if( i == 0 ){
252         fprintf(file_out_xmeshface , "%.9f ", 0.0);
253     } else if( i == mesh->Nx ){
254         fprintf(file_out_xmeshface , "%.9f\n", Lx);
255     } else{
256         position += (*mesh->delta_xP)(i,1);
257         fprintf(file_out_xmeshface , "%.9f ", position);
258     }
259 }
260 position = 0.0;
261 for( j = 0; j <= mesh->Ny+1; j++ ){
262     if( j == 0 ){
263         fprintf(file_out_ymeshnode , "%.9f ", 0.0);
264     } else if( j == mesh->Ny+1 ){
265         fprintf(file_out_ymeshnode , "%.9f\n", Ly);
266     } else{
267         position += (*mesh->d_SP)(1,j);
268         fprintf(file_out_ymeshnode , "%.9f ", position);
269     }
270 }
271 position = 0.0;
272 for( j = 0; j <= mesh->Ny; j++ ){

```

```

273     if( j == 0 ){
274         fprintf(file_out_ymeshface , " %.9f ", 0.0);
275     } else if( j == mesh->Ny ){
276         fprintf(file_out_ymeshface , " %.9f\n" , Ly);
277     } else{
278         position += (*mesh->delta_yP)(1,j);
279         fprintf(file_out_ymeshface , " %.9f " , position);
280     }
281     Nu_medio = 0.0;
282     double Nu_min_hot, Nu_min_cold;
283     double Nu_min_hot_y, Nu_min_cold_y;
284     double Nu_max_hot, Nu_max_cold;
285     double Nu_max_hot_y, Nu_max_cold_y;
286     double Nu_local;
287     for( i = 0; i <= mesh->Nx; i++ ){
288         Nu_i = 0.0;
289         if( i == 0 ){
290             Nu_min_hot = 1000;
291             Nu_max_hot = -1000;
292             position = 0.5 * (*mesh->delta_yP)(1,1);
293             for( j = 1; j <= mesh->Ny; j++ ){
294                 Nu_local = (-1) * (Tn(i+1,j)-Tn(i,j))/(0.5*(*mesh->
295                     delta_xP)(1,j));
296                 if( Nu_local > Nu_max_hot){
297                     Nu_max_hot = Nu_local;
298                     Nu_max_hot_y = position;
299                 }
300                 if( Nu_local < Nu_min_hot){
301                     Nu_min_hot = Nu_local;
302                     Nu_min_hot_y = position;
303                 }
304                 position += (*mesh->d_PN)(1,j);
305                 Nu_i += (*mesh->delta_yP)(1,j) * Nu_local ;
306             }
307             Nu_medio += Nu_i * 0.5 * (*mesh->delta_xP)(1,1);
308         } else if( i == mesh->Nx ){
309             Nu_min_cold = 1000;
310             Nu_max_cold = -1000;
311             position = 0.5 * (*mesh->delta_yP)(mesh->Nx,1);
312             for( j = 1; j <= mesh->Ny; j++ ){
313                 Nu_local = (-1)* (Tn(i+1,j)-Tn(i,j))/(0.5*(*mesh->
314                     delta_xP)(mesh->Nx,j));
315                 if( Nu_local > Nu_max_cold){
316                     Nu_max_cold = Nu_local;
317                     Nu_max_cold_y = position;
318                 }
319                 if( Nu_local < Nu_min_cold){
320                     Nu_min_cold = Nu_local;
321                     Nu_min_cold_y = position;
322                 }
323                 position += (*mesh->d_PN)(mesh->Nx,j);
324                 Nu_i += (*mesh->delta_yP)(mesh->Nx,j) * Nu_local ;
325             }
326             Nu_medio += Nu_i * 0.5* (*mesh->delta_xP)(mesh->Nx,1);
327         } else {
328             for( j = 1; j <= mesh->Ny; j++ ){
329                 Nu_local = (-1) * ((Tn(i+1,j)-Tn(i,j))/(*mesh->d_PE)(i,
330                     j)) + (*un.u)(i,j)*(Tn(i+1,j)+Tn(i,j))*0.5;
331                 Nu_i += Nu_local * (*mesh->delta_yP)(i,j);
332             }
333             Nu_medio += Nu_i * (*mesh->d_PE)(i,1);
334         }

```

```

333     printf("Nu(%d)=%.6f\n", i, Nu_i);
334     fprintf(file_out_Nu, "%.9f\n", Nu_i);
335 }
336 printf("Nu medio=%.6f\n", Nu_medio);
337 printf("Para la pared caliente: \n");
338 printf("Nu max = %.6f (y=%f)\n", Nu_max_hot, Nu_max_hot_y);
339 printf("Nu min = %.6f (y=%f)\n", Nu_min_hot, Nu_min_hot_y);
340 printf("Para la pared fria: \n");
341 printf("Nu max = %.6f (y=%f)\n", Nu_max_cold, Nu_max_cold_y);
342 printf("Nu min = %.6f (y=%f)\n", Nu_min_cold, Nu_min_cold_y);
343 double max_u = -1000, max_v = -1000, max_u_y, max_v_x;
344 position = 0.0;
345 for( j = 0; j <= mesh->Ny+1; j++ ){
346     for( i = 0; i <= mesh->Nx; i++ ){
347         if( (*un.u)(i,j) > max_u ){
348             max_u = (*un.u)(i,j);
349             max_u_y = position;
350         }
351     }
352     if( j == 0 ){
353         position += 0.5 * (*mesh->delta_yP)(1,1);
354     } else if( j == mesh->Ny+1 ){
355         position += 0.5 * (*mesh->delta_yP)(1,mesh->Ny);
356     } else {
357         position += (*mesh->d_PN)(1,j);
358     }
359 }
360 position = 0.0;
361 for( i = 0; i <= mesh->Nx+1; i++ ){
362     for( j = 0; j <= mesh->Ny; j++ ){
363         if( (*un.v)(i,j) > max_v ){
364             max_v = (*un.v)(i,j);
365             max_v_x = position;
366         }
367     }
368     if( i == 0 ){
369         position += 0.5 * (*mesh->delta_xP)(1,1);
370     } else if( i == mesh->Nx+1 ){
371         position += 0.5 * (*mesh->delta_xP)(mesh->Nx,1);
372     } else {
373         position += (*mesh->d_PE)(i,1);
374     }
375 }
376 printf("\nu max = %.6f (y=%f)\n", max_u, max_u_y);
377 printf("v max = %.6f (x=%f)\n", max_v, max_v_x);
378 fclose(file_out_Nu);
379 fclose(file_out_u);
380 fclose(file_out_v);
381 fclose(file_out_p);
382 fclose(file_out_T);
383 fclose(file_out_data);
384 fclose(file_out_fi);
385 fclose(file_out_data);
386 fclose(file_out_xmeshface);
387 fclose(file_out_xmeshnode);
388 fclose(file_out_ymeshface);
389 fclose(file_out_ymeshnode);
390 printf("Convergencia alcanzada o tiempo maximo de simulacion
391 alcanzado.\n");
392 printf("Pr: %f\nRa: %f\nNx: %d\nNy: %d\n", Pr, Ra, Nx, Ny);
393 printf("Error del solver: %e\nError en estacionario: %e\n",
394 epsilon_solver, epsilon);

```

```

393     printf("Tiempo de computacion maximo por ciclo [ms]: %f\n", ct_max)
394     ;
395     printf("Tiempo de computacion minimo por ciclo [ms]: %f\n", ct_min)
396     ;
397     printf("Tiempo de computacion total [ms]: %f\n", ct_total);
398     printf("Tiempo de computacion promedio por ciclo [ms]: %f\n",
399            ct_total/(double)ciclos);
400     printf("Numero de ciclos: %d\n", ciclos);
401     printf("Tiempo total de simulacion: %f\n", time);
402     return 1;
403 }

```

main\_2\_fvm.cpp