4.evolution.h.tex

Chen Peng Chung

February 1, 2026

Contents

一　函數式匯總

本檔案一共包含下列函數，及其估能與意義羅列如下：

函數名稱	功能說明
ModifydRho_F378	(f_3, f_7, f_8 與 f_4, f_9, f_{10}) 正 y 方向動量修正 (平面部分)
ModifydRho_F4910	(f_4, f_9, f_{10} 與 f_3, f_7, f_8) 負 y 方向動量修正 (平面部分)
ModifydRho_F15	(f_{15} 與 f_{18}) 正 y 方向動量修正 (立體部分)
ModifydRho_F16	(f_{16} 與 f_{17}) 負 y 方向動量修正 (立體部分)
dRhoglobal	該點上的密度 ρ 變化
stream_collide_Buffer	buffer layer 提供「邊界外的必要資料」，邊界點則用它完成 LBM 的 streaming/BC/collision。
stream_collide	執行 LBM 演算法的核心 (1. 遷移步 2. 計算宏觀參數 3. 碰撞步 4. 更新賦值)
periodicUD	建立 Up-Down 的週期性邊界條件
periodicSW	建立 Stream-Wise 方向的週期性邊界條件
periodicNML	建立 NormaL(SpanWise) 方向的週期性邊界條件
ccumulateUbulk	計算主流方向速度場在截面上的空間平均 $U_{average}$
Launch_CollisionStreaming	整個 LBM 的主控程序，調用函數式: stream_collide_Buffer, AccumulateUbulk, stream_collide, periodicSW
Launch_ModifyForcingTerm	利用郭老師的外力格式引入 Force Term

二　函數式: ModifydRho_F378 程式碼說明

```
__device__ double ModifydRho_F378(double F3_in,double F7_in,double F8_in,double
    f4_old,double f9_old,double f10_old){
  double drho = F3_in + F7_in + F8_in - f4_old - f9_old - f10_old ;
  return drho ;
}
```

Listing 1: ModifydRho_F378

意義: 以 $\vec{e_y}$ 為外法向量的壁面的質量流率 (mass flow rate) : (可以把壁面面積當成 1) 來理解 F3_in + F7_in + F8_in - f4_old - f9_old - f10_old;
此函數上式在哪裡被用到:

```
//F0_in = F0_in + Modify_dRhoF378(F3_in,F7_in,F8_in,f4_old[index],f9_old[index
    ],f10_old[index])
```

Listing 2: stream_collid_378

三　函數式: ModifydRho_F4910 程式碼說明

```
__device__ double ModifydRho_F4910(double F4_in,double F9_in,double F10_in,
    double f3_old,double f7_old,double f8_old){
    double drho = F4_in+F9_in+F10_in-f3_old-f7_old-f8_old ;
    return drho ;
}
```

Listing 3: ModifydRho_F4910

意義: 以 $-\vec{e_y}$ 方向為外法向量的壁面的單位質量流率。
當壁面面積為 1 時，F4_in+F9_in+F10_in-f3_old-f7_old-f8_old 作為朝 $-\vec{e_y}$ 方向之動量，也可以視為穿過壁面之單位質量流率。
此函數上式在哪裡被用到:

```
    //F0_in=F0_in+ModifydRho_F4910(F4_in,sF9_in,F10_in,f3_old[index],f7_old[
    index],f8_old[index])
```

Listing 4: stream_collid_491

四　函數式: ModifydRho_F15 程式碼說明

```
__device__ double ModifydRho_F15(double F15_in,double f18_old){
  double drho = F15_in - f18_old ;
  return drho ;
}
```

Listing 5: ModifydRho_F15

此程式碼為驗證 yz 方向是否質量守恆，為質量守恆與否的判據。如果是壁面 half-way bounce back 條件，則在此函數的計算下，dRho = 0。
此程式碼會用在哪裡?

```
//F0_in=F0_in+ModifydRho_F15(F15_in,f18_old[index]) ;
```

Listing 6: stream_collid_501

五　函數式: ModifydRho_F16 程式碼說明

```
__device__ double ModifydRho_F16(double F16_in ,double f17_old){
  double drho = F16_in - f17_old ;
  return drho ;
}
```

Listing 7: ModifydRho_F16

此程式碼為驗證 (-y)z 方向是否質量守恆，為質量守恆的判據。如果在邊界上採用 half-way bounce back condition，則用此函數進行量測時，drho = 0
此程式碼用在哪裡?

```
//F0_in=F0_in+ModifydRho_F16(F16_in,f17_old[index]);
```

Listing 8: stream_collid_516

六　函數式: dRhoglobal 程式碼說明

```
__device__ double dRhoglobal(
double F1_in,double F2_in,double F3_in,double F4_in,double F5_in,
double F6_in,double F7_in,double F8_in,double F9_in,double F10_in,
double F11_in,double F12_in,double F13_in,double F14_in,double F15_in,
double F16_in,double F17_in,double F18_in,
double f1_old,double f2_old,double f3_old,double f4_old,double f5_old,
double f6_old,double f7_old,double f8_old,double f9_old,double f10_old,
double f11_old,double f12_old,double f13_old,double f14_old,double f15_old,
double f16_old,double f17_old,double f18_old){double globaldrho =
(F1_in- f1_old)+(F2_in- f2_old)+(F3_in- f3_old)+(F4_in- f4_old)+(F5_in- f5
  _old)
+(F6_in- f6_old)+(F7_in- f7_old)+(F8_in- f8_old)+(F9_in- f9_old)+(F10_in- f10
  _old)
+(F11_in- f11_old)+(F12_in- f12_old)+(F13_in- f13_old)+(F14_in- f14_old)+(F15
  _in- f15_old)
+(F16_in- f16_old)+(F17_in- f17_old)+(F18_in- f18_old) ;
return globaldrho ;
}

```

Listing 9: dRhoglobal

此程式碼為單一格子點的質量守恆計算，其中，求和 F1_in+F2_in..... 為空間點的密度新值。求和 f1_old+f2_old..... 為空間點的密度舊值。則此程式碼為時間前後密度變化之判斷，猜測可以用在全場之後的加總，判斷邊界條件使否合理之判據。

七　函數式: stream_collide 程式碼說明

本文件專為提供的 stream_collide CUDA 核心撰寫，逐一說明每個變數的角色與用途，並概述計算流程。該核心完成 Lattice Boltzmann Method (LBM) 中的 *streaming* 與 *collision*，採用 D3Q19 速度模型 (partical velocity model) 並包含多鬆弛時間矩陣 (MRT) 與無滑移邊界條件的插值反彈格式 (Interpolation Bounce-Back Method) 處理。

七、1　函數內部宣告之變數

Table 1: Stream-Collide 變數符號對照表

變數符號		編號	索引						
			0	**1**	**2**	**3**	**4**	**5**	**6**
Xi	F	3	XiF3_0	XiF3_1	XiF3_2	XiF3_3	XiF3_4	XiF3_5	XiF3_6
		4	XiF4_0	XiF4_1	XiF4_2	XiF4_3	XiF4_4	XiF4_5	XiF4_6
		5	XiF5_0	XiF5_1	XiF5_2	XiF5_3	XiF5_4	XiF5_5	XiF5_6
		6	XiF6_0	XiF6_1	XiF6_2	XiF6_3	XiF6_4	XiF6_5	XiF6_6
		15	XiF15_0	XiF15_1	XiF15_2	XiF15_3	XiF15_4	XiF15_5	XiF15_6
		16	XiF16_0	XiF16_1	XiF16_2	XiF16_3	XiF16_4	XiF16_5	XiF16_6
		17	XiF17_0	XiF17_1	XiF17_2	XiF17_3	XiF17_4	XiF17_5	XiF17_6
		18	XiF18_0	XiF18_1	XiF18_2	XiF18_3	XiF18_4	XiF18_5	XiF18_6
XBFL	f	37	XBFLf37_0	XBFLf37_1	XBFLf37_2	XBFLf37_3	XBFLf37_4	XBFLf37_5	XBFLf37_6
		38	XBFLf38_0	XBFLf38_1	XBFLf38_2	XBFLf38_3	XBFLf38_4	XBFLf38_5	XBFLf38_6
		49	XBFLf49_0	XBFLf49_1	XBFLf49_2	XBFLf49_3	XBFLf49_4	XBFLf49_5	XBFLf49_6
		410	XBFLf410_0	XBFLf410_1	XBFLf410_2	XBFLf410_3	XBFLf410_4	XBFLf410_5	XBFLf410_6
XiBFL	f	3	XiBFLf3_0	XiBFLf3_1	XiBFLf3_2	XiBFLf3_3	XiBFLf3_4	XiBFLf3_5	XiBFLf3_6
		4	XiBFLf4_0	XiBFLf4_1	XiBFLf4_2	XiBFLf4_3	XiBFLf4_4	XiBFLf4_5	XiBFLf4_6
		15	XiBFLf15_0	XiBFLf15_1	XiBFLf15_2	XiBFLf15_3	XiBFLf15_4	XiBFLf15_5	XiBFLf15_6
		16	XiBFLf16_0	XiBFLf16_1	XiBFLf16_2	XiBFLf16_3	XiBFLf16_4	XiBFLf16_5	XiBFLf16_6
YBFL	f	3	YBFLf3_0	YBFLf3_1	YBFLf3_2	YBFLf3_3	YBFLf3_4	YBFLf3_5	YBFLf3_6
		4	YBFLf4_0	YBFLf4_1	YBFLf4_2	YBFLf4_3	YBFLf4_4	YBFLf4_5	YBFLf4_6
		15	YBFLf15_0	YBFLf15_1	YBFLf15_2	YBFLf15_3	YBFLf15_4	YBFLf15_5	YBFLf15_6
		16	YBFLf16_0	YBFLf16_1	YBFLf16_2	YBFLf16_3	YBFLf16_4	YBFLf16_5	YBFLf16_6

七、2　線程 (Thread), 區塊 (Block), 與網格 (Grid) 索引

```
1 const int i = blockIdx.x*blockDim.x + threadIdx.x;
2 const int j = blockIdx.y*blockDim.y + threadIdx.y;
3 const int k = blockIdx.z*blockDim.z + threadIdx.z;
```

Listing 10: Kernel index computation

如程式碼 Kernel index computation 所示，有 3 個變數在 GPU 中特別重要，分別是: blockIdx, threadIdx, blockDim。

- Grid：由同一個 kernel 所產生的線程 (thread) 集合，或稱線程網格，特性是”對應同一個 Grid 的所有 thread 共享記憶體”。

- Block：Grid 的組成單元，一個 Block 包含一組線程。特色是，”對應同一個 BLock 的所有 thread 同步，且共享記憶體”。
- 變數名稱: blockIdx： block(線程塊) 在 Grid(線程網格) 中之索引編號；threadIdx：Thread(線程) 在 Block(線程塊) 中的索引編號。

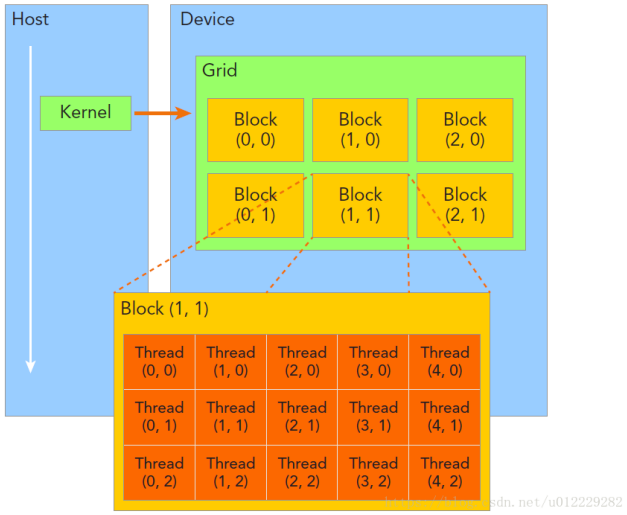

Figure 七.1: threads, block, grid, 觀念

Source: https://blog.csdn.net/u012229282/article/details/79972014

所以 blockDim 是每一個 block 的大小或者維度，由內部所塞滿的 threads 所決定。而 blockIdx 是 block 相對於 grid 的索引編號。同理，threadIdx 是 thread 相對於 block 的索引編號。所以針對符號 i,j,k 有如下解釋:

1. i：thread 在一個 Grid 中的 x 方向”固定”索引編號。
2. j：thread 在一個 Grid 中的 y 方向”固定”索引編號。
3. k：thread 在一個 Grid 中的 z 方向”固定”索引編號。

如上述可以知道，對於 i,j,k，是固定的 threads 索引編號，對應物理空間的計算網格，一個 thread 對應到一個真實物理空間計算點，因此，採用 const int 來宣告 htreads 的全局索引，一個物理空間算點只由一個 thread 來執行。因此:

定義了 i,j,k 作為索引編號，固定，是為了穩穩對應物理空間計算網格。

概念	說明
Threads 索引編號	在 GPU 端上執行程式碼的工作單元 (工號)
物理網格點索引編號	流體區域的空間排序 (x,y,z)
(i,j,k) 的定義目的	建立 threads ↔ 物理網格點的穩定對應
如果 i(i,j,k) 不固定	讀寫不同物理間計算點，計算錯誤。

七、3 規範要處理的物理計算點

```
if( i <= 2 || i >= NX6-3 || j <= 6 || j >= NYD6-7 || k <= 2 || k >= NZ6-3 )
    return;
```

Listing 11: evolution.h369

限制物理間計算點之範圍，避免 Halo 層被誤用於計算。其中，y 方向缺少的計算點 j = [3,6] ，[NYD6-7,NYD6-4] 則交由 stream_collid_Buffer 處理。

需要注意的是: Kernel 內的判斷式並不會影響 Halo 層的使用，我指的是 X 方向做邊界點附近的插值行為時，仍然會用到 buffer lasyers 的分佈函數，換言之，做 *span-wise* 方向的週期性邊界條件時，仍然取用規範之外的 threads 所算的值。

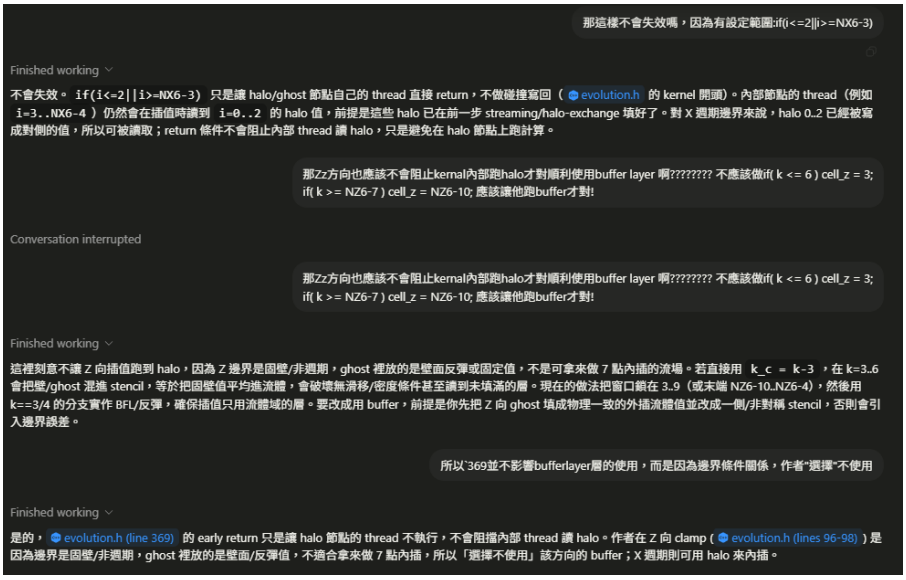

Figure 七.2: 不會阻擋 Halo 層的讀取

七、4 物理網格之座標設定

此程式碼為降維索引編號，為三維座雕轉換唯一維度索引編號。

```
const int index = j*NX6*NZ6 + k*NX6 + i;
      int idx;
      int idx_xi = j*NZ6 + k;//用xi來代替變換後的格點座標
const int nface = NX6*NZ6;
const int nline = NX6;
```

Listing 12: evolution.h365-367

說明:

1. const int index = j*NX6*NZ6 + k*NX6 + i; ：為物理網格之索引編號，將三維空間座標 (i,j,k) 由 x 方向 (*span-wise*) 從 0 開始編號，再往 z 方向 (*normal*) 編號，最後往 y 方向 (*stream-wise*) 編號，形成唯一維度索引編號 index。
2. nline : 在同一個 x 軸上構成的網格連線
3. nface : 在同一個 x-z 平面上構成的網格平面
4. Idx_xi : 為某一塊 y-xi 平面上的網格系統，先將 z 軸之非均勻網格映射為均勻網格，形成 xi 軸。Idx_xi 為 y-xi 平面網格系統之一維網格編號。注意:xi 軸與 z 軸共用一套編號系統，因為 xi 軸與 z 軸一對一對應。

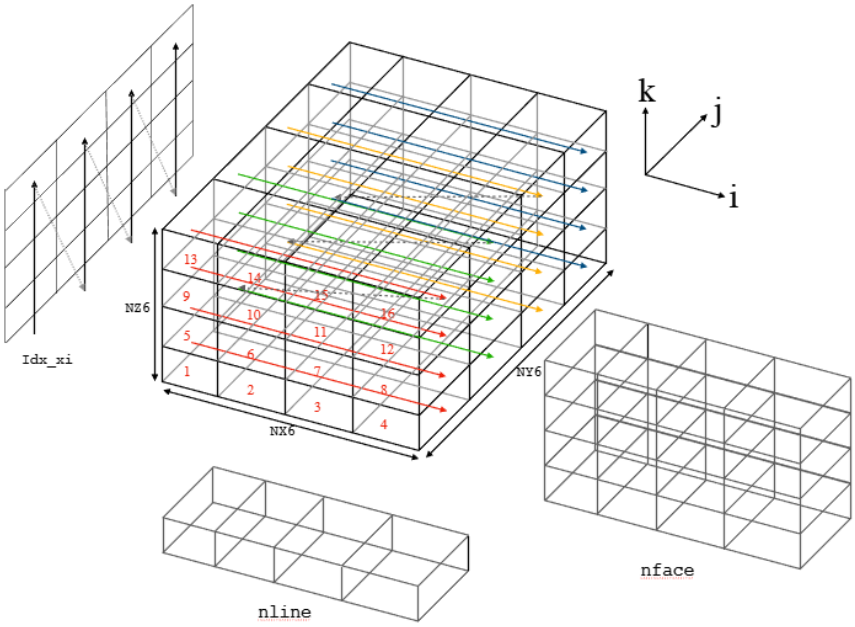

Figure 七.3: 程式碼中座標編號可視化: nline,nface,index

七、5 碰撞後一般態分佈函數

```
double F0_in,F1_in,F2_in,F3_in,F4_in,F5_in,F6_in,F7_in,F8_in,F9_in,F10_in
double F11_in,F12_in,F13_in,F14_in,F15_in,F16_in,F17_in,F18_in ;
```

Listing 13: evolution.h378

F0-18_in 是 (k(A 非物理空間計算點) 的 (碰撞後一般態分佈函數)) 。

七、6 MRT Variable

七、7 xi_h[NZ6*NYD6] ：buffer layer

```
int cell_z = k-3 ;
if(k<=6) cell_z = 3 ;//強制採用固定點做內插
if(k>=NZ6-7) cell_z = NZ6-10 ; //強制採用定點做內插
```

Listing 14: evolution.h387

在後續使用 ξ 方向之內插時，會用到變量 k_c，此時，會以如上定義的 cell_z 作為內插成員的起始點，往右列出 7 個位置作為內插點。如下定義與實際使用概況:

```
//定義 :
#define F4_Intrpl7(f,Fi,j,k,i_c,j_c,k_c,i,j,idx_xi,y_0,y_1,....xi_0,xi_1,...)
//使用 :
F4_Intrpl7(f4_old ,i,j,k ,(i-3),(j-3),cell_z ,i,j,idx_xi
,Y2_0,Y2_1,Y2_2,Y2_3,Y2_4,Y2_5,Y2_6
,XiF4_0,XiF4_1,XiF4_2,XiF4_3,XiF4_4,XiF4_5,XiF4_6
)//一共24個自變數要填入
```

Listing 15: 進行七點內插公式的定義與使用

1. xi_h：(不含山丘) 離散化無因次化 z 座標 xi_h 的陣列尺寸為 NZ6，實際寫入範圍為 xi_h[3:NZ6-4]

   xi_h 配置記憶體：

```
nBytes = NZ6 * sizeof(double);
CHECK_CUDA( cudaMallocHost( (void**)&xi_h, nBytes ) );
CHECK_CUDA( cudaMalloc( &xi_d, nBytes ) );
```

   評述：離散化無因次化座標 xi_h 有設置 buffer layer，但是沒有寫進東西，後續可以避免越界，但是有用到 buffer layer 者沒有意義，因為沒有寫入。

Table 2: xi_h 區域分類

區域	範圍	說明
k = buffer	k=[0,2]or[NZ6-3,NZ6-1]	沒有寫入值 | 沒有做使用
k = interrior	k=[3,NZ6-4]	有寫入值 | 有做使用

   xi_h 寫入值：

```
for( int k = bfr; k < NZ6-bfr; k++ ){
  xi_h[k] = tanhFunction( LXi, minSize, a, (k-3), (NZ6-7) ) - minSize/2.0;
}
```

   xi_h[0,1,2]、xi_h[NZ6-3,2,1] 沒有被寫入 buffer layer 的值。

   xi_h 使用情況：

```
void GetXiParameter(
    double *XiPara_h[7],    double pos_z,       double pos_y,
    double *Pos_xi,         int IdxToStore,     int k  ) //IdxToStore =
    now ; k = start
{...//其中的Pos_xi = xi_h
if( k >= 3 && k <= 6 ){
        GetParameter_6th( XiPara_h, pos_xi, Pos_xi, IdxToStore, 3 );
    } else if ( k >= NZ6-7 && k <= NZ6-4 ) {
        GetParameter_6th( XiPara_h, pos_xi, Pos_xi, IdxToStore, NZ6-10 );
    } else {
        GetParameter_6th( XiPara_h, pos_xi, Pos_xi, IdxToStore, k-3 );
    }}//XiParaF3_d被特殊寫入導致k=3,4,5以及k=NZ6-4,-5,-6有被寫入有被使用
```

   xi_h[0,1,2]、xi_h[NZ6-3,2,1] 沒有被使用。

2. XiParaF3_d：分三個區域分類作為 Z 方向預配置連乘權重一維連續記憶體 (分配到各個位置點為二維連續記憶體) 尺寸大小為 ZNZ6*j+NYD6。

   XiParaF3_d 配置記憶體：

```
nBytes = NYD6 * NZ6 * sizeof(double);
....
for( int i = 0; i < 7; i++ ){
```

```
  CHECK_CUDA( cudaMallocHost( (void**)&XiParaF3_h[i], nBytes ) );
....
  CHECK_CUDA( cudaMalloc( &XiParaF3_d[i], nBytes ) );....
}
```

Table 3: XiParaF3_d 區域分類

區域	範圍	說明
k = buffer	k=[0,2]or[NZ6-3,NZ6-1]	沒有寫入值 | 沒有做使用
k = boundary layer	k=[3,5]or[NZ6-6,NZ6-4]	有特殊寫入值 | 有做使用
k = interrior	k=[6,NZ6-7]	有寫入值 | 有做使用

XiParaF3_d 寫入值 (xi_h 使用情況 2)：GetXiParameter 定義寫在1

```
void GetIntrplParameter_Xi() {
for(int j = 3 ; j <= NYD6-4 ; j++){
  for(int k = 3 ; k <= NZ6-4 ; k++){
    GetXiParameter(XiParaF3_d,z_h[NZ6*j+k],y_h[j]-minSize,xi_h,NZ6*j+k,k)
  }}}
```

3. F3_Intrpl7：XiParaF3_d 使用情況：

```
int cell_z = k-3;
if( k <= 6 ) cell_z = 3;
if( k >= NZ6-7 ) cell_z = NZ6-10;//....
F3_Intrpl7(..cell_z, .., idx_xi, .., XiF3_0, XiF3_1, XiF3_2, ..);
//在Launch_CollisionStreaming中簽套為
F3_Intrpl7(..cell_z, .., idx_xi, .., XiParaF3_d[0], XiParaF3_d[1], XiParaF
    3_d[2], ..);
```

所以說，在 k = 3,4,5 以及 k = NZ6-6,-5,-4，都沒有被使用到該點相應的元素。

七、8 Matrix

七、9 下邊界條件與上邊界條件 (Up and Down Direction)

```
if( k == 3 ){//physical boundary//下邊界邊界條件
  F5_in  = f6_old[index]; //上 = 下
  F11_in = f14_old[index];//正x上 = 負x下
  F12_in = f13_old[index];//負x上 = 正x下
  F15_in = f18_old[index];//正y上 = 負y下
  F16_in = f17_old[index];//負y上 = 正y下
}
if( k == NZ6-4 ){//physical boundary //上邊界條件
  F6_in  = f5_old[index]; //下 = 上
  F13_in = f12_old[index];//正x下 = 負x上
```

```
    F14_in = f11_old[index];//負x下 = 正x上
    F17_in = f16_old[index];//正y下 = 負y上
    F18_in = f15_old[index];//負y下 = 正y上
}
```

下邊界條件 (Down direction B.C.) 與上邊界條件 (Up direction B.C.) 為採用：Link-wise 計算點佈局的平面無滑移邊界條件的 Half-way Bounce Back 格式。公式為：

Table 4: 邊界處理方法比較

節點佈局	宏觀邊界條件	**LBM** 邊界處理格式
Link-wise boundary	Non-slip Boundary Condition at Straight Wall	Full Way Bounce Back Method
		Half Way Bounce Back Method
	Non-slip Boundary Condition at Curvilinear Wall	Interpolation Bounce Back Method
	Velocity Open Boundary Condition	Half Way Bounce Back Method
	Pressure Open Boundary Condition	Anti Half Way Bounce Back Method
Wet-node boundary	Non-slip Boundary Condition	Equilibrium Scheme
		Non-equilibrium Bounce Back Scheme
		Non-equilibrium Extrapolation Scheme
	Velocity Open Boundary Condition	–
	Pressure Open Boundary Condition	–

Half-way Bounce Back Method 的一般表達式為：

$$f_{\hat{i}}^{Post-streaming}(\vec{r_b}, t+\Delta t) = f_i^{\star}(\vec{r_b}, t) + 2w_i\rho_w\frac{\vec{e_i}\cdot\vec{u}_w}{c_s^2} \tag{七.1}$$

從學長的 (馮) 論文可以知道，有關於上邊界與下邊界 (山丘部分略為複雜)，由於宏觀條件均為”平面無滑移條件”，因此均採用 Half-way Bounce Back Method 來處理。

七、10　曲面邊界條件的處理: Interpolation Bounce Back Method

在處理邊界條件時，有幾個要點需要注意，最後一步更新的對象必然為 Boundary node 的一般態分佈函數，以及邊界條件的處理應該視為遷移步的一種特例，對於 Half-way Bounce Back 以及 Interpolation Bounce Back Method 而言。

七、10.1　Bouzidi 插值反彈法的物理意義

Bouzidi 等人提出的插值反彈格式 (Interpolation Bounce-Back Scheme) 用於處理曲面邊界上的無滑移條件。其核心思想是根據壁面交點位置 q 的不同，採用不同階數的插值公式：

- 當 $q > 0.5$ 時：壁面交點靠近流體節點，採用線性插值，很遠的時候

- 當 $q < 0.5$ 時：壁面交點靠近固體內部，採用二次插值，很近的時候（需 Lagrange 7 點內插）

其中，q 定義為流體節點到壁面交點的無量綱距離。對於無滑移邊界條件，Bouzidi 格式的一般形式為：

$$f_{\hat{i}}(\vec{x}_f, t+\Delta t) = \begin{cases} \dfrac{1}{2q} f_i(\vec{x}_f, t) + \dfrac{2q-1}{2q} f_{\hat{i}}(\vec{x}_f, t) & q > 0.5 \text{ (線性)} \\ \text{Lagrange 7-point interpolation} & q < 0.5 \text{ (高階)} \end{cases} \tag{七.2}$$

其中，f_i 為入射方向分佈函數，$f_{\hat{i}}$ 為反射方向分佈函數。

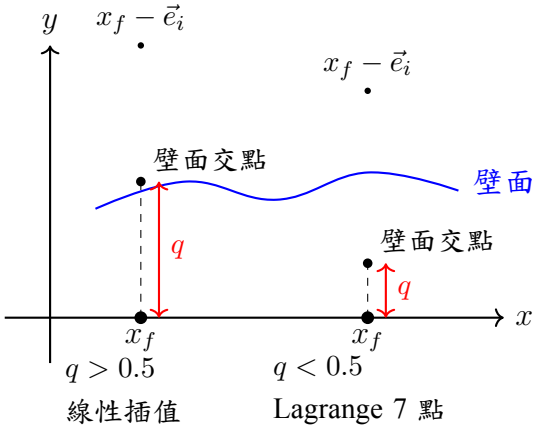

Figure 七.4: Bouzidi 插值反彈法示意圖：根據 q 值選擇不同插值策略

七、10.2 程式碼實現

以下程式碼實現了針對 D3Q19 速度模型中不同方向的 Bouzidi 插值反彈法。為提高可讀性，對重複部分予以省略：

```
//BFL Linear - 處理 boundary fitted layer 的線性插值
if( k == 3 || k == 4 ) {
    idx_xi = (k-3)*NYD6+j;

    // ========== 第一組：處理 F3, F7, F8 (正 y 方向平面) ==========
    if ( BFLReqF3_d[(k-3)*NYD6+j] == 1 ) {
        if(Q3_d[(k-3)*NYD6+j] > 0.5) {  // 線性插值
            F3_in = (1/(2*Q3_d[(k-3)*NYD6+j]))*f4_old[index] +
                    ((2*Q3_d[(k-3)*NYD6+j]-1)/(2*Q3_d[(k-3)*NYD6+j]))*f3_old[index];
            F7_in = (1/(2*Q3_d[(k-3)*NYD6+j]))*f10_old[index] +
                    ((2*Q3_d[(k-3)*NYD6+j]-1)/(2*Q3_d[(k-3)*NYD6+j]))*f7_old[index];
            F8_in = (1/(2*Q3_d[(k-3)*NYD6+j]))*f9_old[index] +
                    ((2*Q3_d[(k-3)*NYD6+j]-1)/(2*Q3_d[(k-3)*NYD6+j]))*f8_old[index];
        }
        if(Q3_d[(k-3)*NYD6+j] < 0.5) {  // Lagrange 7點插值
            Y_XI_Intrpl7(f4_old, F3_in, i,j,k, (i-3),(j-3),3, i,j, idx_xi,
                         YBFLf3_0, ..., YBFLf3_6, XiBFLf3_0, ..., XiBFLf3_6);
            X_Y_XI_Intrpl7(f10_old, F7_in, i,j,k, (i-3),(j-3),3, i,j, idx_xi,
                           XBFLf37_0,...,XBFLf37_6, YBFLf3_0,...,YBFLf3_6,
                           XiBFLf3_0,...,XiBFLf3_6);
            X_Y_XI_Intrpl7(f9_old, F8_in, ...);  // 結構同上，參數省略
        }
        //F0_in = F0_in + ModifydRho_F378(...);  // 密度修正（已註解）
    }

    // ========== 第二組：處理 F4, F9, F10 (負 y 方向平面) ==========
```

```
    if ( BFLReqF4_d[(k-3)*NYD6+j] == 1 ) {
        if(Q4_d[(k-3)*NYD6+j] > 0.5) {  // 線性插值（公式同第一組）
            F4_in  = (1/(2*Q4_d[(k-3)*NYD6+j]))*f3_old[index] +
                     ((2*Q4_d[(k-3)*NYD6+j]-1)/(2*Q4_d[(k-3)*NYD6+j]))*f4_old[index];
            F9_in  = (1/(2*Q4_d[(k-3)*NYD6+j]))*f8_old[index] +
                     ((2*Q4_d[(k-3)*NYD6+j]-1)/(2*Q4_d[(k-3)*NYD6+j]))*f9_old[index];
            F10_in = (1/(2*Q4_d[(k-3)*NYD6+j]))*f7_old[index] +
                     ((2*Q4_d[(k-3)*NYD6+j]-1)/(2*Q4_d[(k-3)*NYD6+j]))*f10_old[index];
        }
        if(Q4_d[(k-3)*NYD6+j] < 0.5) {  // Lagrange 7點插值（結構同第一組）
            Y_XI_Intrpl7(f3_old, F4_in, ...);    // 參數結構與第一組相同
            X_Y_XI_Intrpl7(f8_old, F9_in, ...);  // 僅分佈函數編號不同
            X_Y_XI_Intrpl7(f7_old, F10_in, ...);
        }
        //F0_in = F0_in + ModifydRho_F4910(...);  // 密度修正（已註解）
    }

    // ========== 第三組：處理 F15（正 y 方向立體）==========
    if ( BFLReqF15_d[(k-3)*NYD6+j] == 1 ) {
        if(Q15_d[(k-3)*NYD6+j] > 0.5) {  // 線性插值
            F15_in = (1/(2*Q15_d[(k-3)*NYD6+j]))*f18_old[index] +
                     ((2*Q15_d[(k-3)*NYD6+j]-1)/(2*Q15_d[(k-3)*NYD6+j]))*f15_old[index];
        }
        if(Q15_d[(k-3)*NYD6+j] < 0.5) {  // Lagrange 7點插值
            Y_XI_Intrpl7(f18_old, F15_in, i,j,k, (i-3),(j-3),3, i,j, idx_xi,
                         YBFLf15_0,...,YBFLf15_6, XiBFLf15_0,...,XiBFLf15_6);
        }
        //F0_in = F0_in + ModifydRho_F15(...);  // 密度修正（已註解）
    }

    // ========== 第四組：處理 F16（負 y 方向立體）==========
    if ( BFLReqF16_d[(k-3)*NYD6+j] == 1 ) {
        if(Q16_d[(k-3)*NYD6+j] > 0.5) {  // 線性插值（結構同第三組）
            F16_in = (1/(2*Q16_d[(k-3)*NYD6+j]))*f17_old[index] +
                     ((2*Q16_d[(k-3)*NYD6+j]-1)/(2*Q16_d[(k-3)*NYD6+j]))*f16_old[index];
        }
        if(Q16_d[(k-3)*NYD6+j] < 0.5) {  // Lagrange 7點插值
            Y_XI_Intrpl7(f17_old, F16_in, i,j,k, (i-3),(j-3),3, i,j, idx_xi,
                         YBFLf16_0,...,YBFLf16_6, XiBFLf16_0,...,XiBFLf16_6);
        }
        //F0_in = F0_in + ModifydRho_F16(...);  // 密度修正（已註解）
    }
}
```

Listing 16: evolution.h168-223: Bouzidi 插值反彈法實現（簡化版）

七、10.3 程式碼結構解析

上述程式碼處理了四組分佈函數的壁面邊界條件，分別對應 D3Q19 模型中不同的速度方向：

Table 5: D3Q19 壁面處理方向對照表

處理組別	入射方向	反射方向	物理意義
第一組	f_4, f_{10}, f_9	F_3, F_7, F_8	正 y 方向（平面）
第二組	f_3, f_8, f_7	F_4, F_9, F_{10}	負 y 方向（平面）
第三組	f_{18}	F_{15}	正 y 方向（立體）
第四組	f_{17}	F_{16}	負 y 方向（立體）

關鍵變數說明：

- `BFLReqF3_d[]` 等：標記該節點是否需要壁面處理（1 = 需要，0 = 不需要）
- `Q3_d[], Q4_d[]` 等：儲存無量綱壁面距離 q 值
- `idx_xi`：壁面座標索引，用於提取 ξ 方向的網格資訊

兩種插值方案比較：

特性	線性插值 ($q > 0.5$)	**Lagrange 7 點插值** ($q < 0.5$)
計算複雜度	低	高
精度	一階	六階
所需節點數	2 個	$7^3 = 343$ 個（三維）
適用情況	壁面靠近流體節點	壁面靠近固體內部
程式碼函數	直接計算	`Y_XI_Intrpl7, X_Y_XI_Intrpl7`

為何需要高階插值？

當 $q < 0.5$ 時，壁面交點非常靠近固體內部，此時若使用線性插值會導致：

1. 數值不穩定（分母 $2q$ 過小）
2. 精度損失（外推而非內插）
3. 無法準確捕捉曲面幾何

因此採用 Lagrange 七點內插，利用鄰近 $7 \times 7 \times 7$ 個節點的資訊，提供高精度的分佈函數值。

七、11　補充

`f_old` 為上一個時間步所更新的碰撞後插值後一般態分佈函數
`f_new` 為本時間步所更新的碰撞後插值後一般態分佈函數

八　函數式: periodicSW 程式碼說明

摘要: 主流方向 (Stream-wise direction) 的週期性邊界條件。從主程式碼可以發現，只有在 x 方向 (stream-wise direction) 設置使用週期性邊界條件，在一般教科書中，對於 distribution function 的週期性邊界條件的實現方式如下:

```
__global__ void periodicSW(
    double *f0_new, ..., double *f18_new,  // 19個distribution functions
    double *u, double *v, double *w, double *rho_d)
{
    int buffer = 3;  // 緩衝層厚度

    // ============ 第一部分: 左邊界處理 ============
    // 將右側內部節點複製到左側邊界緩衝區
    idx_buffer = j*NZ6*NX6 + k*NX6 + i;           // 左邊界位置 (i=0,1,2)
    idx = idx_buffer + (NX6-2*buffer-1);          // 對應的右側內部節點
    //上式證明(NX6-2*buffer-1)為物理周長計算點數量
    f0_new[idx_buffer] = f0_new[idx];   // 複製所有19個distribution
```

```
13      f1_new[idx_buffer] = f1_new[idx];
14      ...
15      f18_new[idx_buffer] = f18_new[idx];
16
17      u[idx_buffer] = u[idx];              // 複製巨觀變量
18      v[idx_buffer] = v[idx];
19      w[idx_buffer] = w[idx];
20      rho_d[idx_buffer] = rho_d[idx];
21
22      // ============ 第二部分: 右邊界處理 ============
23      // 將左側內部節點複製到右側邊界緩衝區
24      idx_buffer = j*NX6*NZ6 + k*NX6 + (NX6-1-i);    // 右邊界位置
25      idx = idx_buffer - (NX6-2*buffer-1);           // 對應的左側內部節點
26
27      f0_new[idx_buffer] = f0_new[idx];    // 複製所有19個distribution
28      ...
29      f18_new[idx_buffer] = f18_new[idx];
30  }
```

Listing 17: evolution.h671-715

所以一句話描述週期性邊條件: 複製另一側內點的值作為 buffer 層的值，且兩者相距一個物理週期，且兩者在同一個 times step 上，如此，則為實現週期性邊界條件。需要注意的是:

```
1  f0_new[idx_buffer] = f0_new[idx]; //f0_new為碰撞遷移步(periodicSW(...))後得到，
       且這個複製行為發生在periodicSW 之後。
```

Listing 18: evolution.h688

問題: 為甚麼複製後的 buffer 層的值，與另一端內點的數個值在同一個 time step 上。 這個問題可以分成兩句話來解答:

- `f0_new[idx]`(等式右側): 另一端內側點數個值，是在 Collision step(碰撞步) 與 Streaming step(遷移步) 後得到的值，time step = n+1。
- `f0_new[idx_buffer] = f0_new[idx];`: 是在函數式 `stream_collide` 之後執行，所以這句程式碼只是空間上的週期性連接，兩者的 time step 在同一個時間上: n+1。

問題: 他認定最右側三層作為右側 buffer，那問題是: 為甚麼多一層，跳過，從頭到尾 s 根本用不到那一層? 回答: 這裡沒有「多一層沒用到」，而是週期端點重複 + 7 點內插所需的 ghost 層。具體可分成以下幾點說明:

- 變數定義中 NX=32, NX6=NX+7，而內部計算區間由 `if( i <= 2 || i >= NX6-3 ) return;` 決定，故內點為 $i = 3 \sim NX6 - 4$（此例即 $3 \sim 35$）。
- 右側 buffer 實際為 $i = NX6 - 3 \sim NX6 - 1$（即 36, 37, 38），由 periodicSW 複製內點填入；因此最外層 38 不是跳過，而是被填值。
- 38 會被內點使用: 7 點內插會讀到 $i + 3$；當 $i = 35$ 時就需要讀到 38。
- 內點數出現 33 是因為週期端點重複：$x = 0$ 與 $x = LX$ 為同一物理位置，因此獨立物理點仍是 $NX = 32$。

```
#define NX  32
#define NX6 (NX+7)
...
if( i <= 2 || i >= NX6-3 ) return; // 內點 i=3..NX6-4 (=3..35)
```

Listing 19: variables.h + evolution.h (內點範圍)

```
idx = j*nface + k*nline + i_c;
F1_in = Intrpl7( f[idx],   x_0[idx_x],
                 f[idx+1], x_1[idx_x],
                 f[idx+2], x_2[idx_x],
                 f[idx+3], x_3[idx_x],
                 f[idx+4], x_4[idx_x],
                 f[idx+5], x_5[idx_x],
                 f[idx+6], x_6[idx_x] );
// 當 i=35 時, i_c=32 -> 讀到 32..38
```

Listing 20: interpolationHillISLBM.h13-15

```
if( Uniform_In_Xdir ){
  dx = LX / (double)(NX6-2*bfr-1);//x方向實際格子大小
  for( int i = 0; i < NX6; i++ ){
    x_h[i]  = dx*((double)(i-bfr));//實際每一個格點的位置
  }
  } else {
    printf("Mesh needs to be uniform in periodic hill problem, exit...\n");
    exit(0);
}
```

Listing 21: initialization.h55-63 (GenerateMesh_X)

```
idx_buffer = j*NX6*NZ6 + k*NX6 + (NX6-1-i);
//右側 ghost: i=0,1,2 -> idx_buffer=38,37,36
//對應左側內點: idx=5,4,3
//i=35 為內點，會在 stream_collide 中被計算
```

Listing 22: evolution.h700

index:35 存在在計算物理點的視角中，為”端點重複”之意義，換言之，在 x 方向上，參與碰撞與遷移的計算點為一段週期 + 下一段週期的第一個點。所以設置 index:35 只是為了在端點 Lagrange 內插的時候必須用到下一段周期的第一格點週期性邊界條件 periodic boundary condition: 週期性邊條件由以下兩點實現: 其一為 buffer 層的賦予，其二為壁面來向第一段 Stream-Collision 交由另一端邊界計算點 (等效下一段值周期第一個算點) 來呈現。

- buffer 層的值由另一端物理空間計算點的值給定:

```
f_new[0] = f_new[32] ; //左側buffer layer第一個計算點的值由一端計算點32 的
    函數值來決定
f_new[1] = f_new[33] ;
```

```
3 f_new[2] = f_new[34] ;
4 f_new[36] = f_new[4] ; //注意這邊:右側buffer layer第一個算點的值由左側物理
      空間第二個計算點4的值給定
5 f_new[37] = f_new[5] ;
6 f_new[38] = f_new[6] ;
```

Listing 23: left buffer layer and right buffer layer

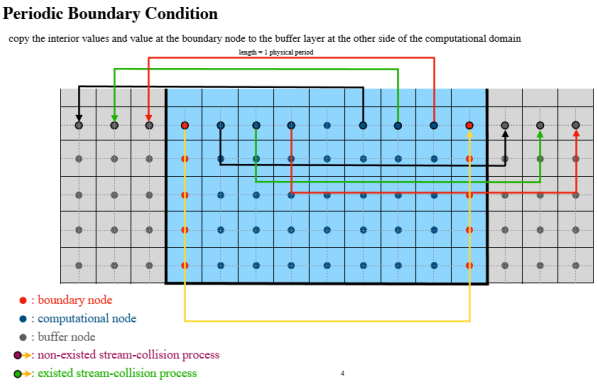

Figure 八.5: periodic condition setting

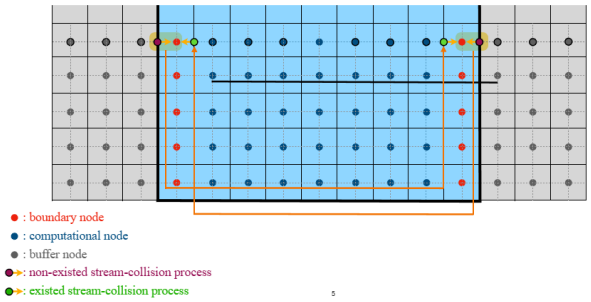

Figure 八.6: collision-streaming step is copied to the other point