第 1 部分：《格子 Boltzmann 方法的理论及应用》的有关源程序

程序 3　基于伪势多相模型的相分离格子 Boltzmann 模拟

3.1 问题描述

与亚稳态区域基于成核生长而发生的相变不同，**相分离**（phase separation）通常是指发生在体系的不稳定区域的**旋节分解**（spinodal decomposition）。在不稳定区域，任何微小的涨落或扰动都可能导致自发的相分离。以淬火过程为例，当体系淬火到不稳定区域时，会立即发生旋节分解，即相分离。类似地，对气液两相系统，假设初始时刻的密度或比体积处在曲线 *MON* 所对应的热力学不稳定区域，只要施加微小的扰动就可能促使相分离的发生。

在本附录，我们提供了一个基于伪势模型的相分离格子 Boltzmann 模拟程序。具体地，采用 D2Q9 离散速度模型以及相应的格子 Boltzmann-BGK 方程。作用力项和伪势相互作用力参见《格子 Boltzmann 方法的理论及应用》中式(10.20)和式(10.21)。式(10.21)中的权系数 w_α 与平衡态分布函数中的权系数 ω_α 存在 $w_\alpha = \omega_\alpha / c_s^2 = 3\omega_\alpha$ 的关系。在本算例中，不考虑重力，并且采用指数形式的伪势，即 $\psi(\rho) = \psi_0 \exp(-\rho_0/\rho)$ [11, 12]。相关的模拟参数为：$\psi_0 = 1$，$\rho_0 = 1$，$G = -10/3$。根据 Maxwell 等面积法则，对平直气液界面，这些模拟参数对应的气液共存密度为 $\rho_l \approx 2.783$ 和 $\rho_g \approx 0.3675$；对曲线界面，气液共存密度与平直表面的结果稍有不同。

计算区域的大小为 $L_x \times L_y = 150 \times 150$（格子单位），采用湿节点式布置方式，因此相应的网格节点数为 $N_x \times N_y = 151 \times 151$。$x$ 方向和 y 方向均采用周期性边界条件，为此须在计算区域的四周各布置一层虚拟层，即 $x = 0$ 和 $x = N_x + 1$ 为 x 方向布置的虚拟层，而 $y = 0$ 和 $y = N_y + 1$ 则为 y 方向布置的虚拟层。对于这样一个二维问题的周期性边界条件，可以采取一种较为简便的处理：假定虚拟层的节点也参与碰撞迁移，因此在迁移步之后，$t + \delta_t$ 时刻整个真实物理区域的分布函数都是已知的，于是可直接将真实物理层 $x = N_x$ 上的分布函数赋给虚拟层 $x = 0$ 的节点，同时将真实物理层 $x = 1$ 上的分布函数赋给虚拟层 $x = N_y + 1$ 的节点，y 方向也可进行类似的处理。这样处理符合周期性边界条件的物理本质。程序由 C++语言编写，支持的环境为 Microsoft Visual Studio 2010。初始时刻，给定一个随机的密度扰动，由 C++语言中的函数 rand()产生随机数。虽然该程序是基于指数形式的伪势，

读者可在此基础上采用平方根形式的伪势[13]及相应的作用力格式[14, 15]，并将程序拓展至多松弛碰撞算子[15]。此外，还可在该程序的基础上加入壁面边界及相应的接触角格式[16-19]。

3.2 程序变量表及源程序

变量表

变量名	变量含义
Q	离散速度的总个数
Nx, Ny	x, y 方向的节点数
Lx, Ly	x, y 方向的长度
i, j	节点坐标
dx, dy	x, y 方向的网格步长
dt	时间步长
m	演化次数
k	离散速度方向 α
niu	运动黏度系数 ν
tau_f	无量纲松弛时间 τ
e[Q][2]	离散速度 $\boldsymbol{e}_\alpha$
w[Q]	权系数 ω_α
totalmass0	系统初始总质量
totalmass	系统瞬态总质量
rho[Nx+2][Ny+2]	密度
u[Nx+2][Ny+2][2]	速度
fai[Nx+2][Ny+2]	伪势
Fm[Nx+2][Ny+2][2]	伪势相互作用力
F[Nx+2][Ny+2][Q]	碰撞后的分布函数

源程序

```
#include "stdafx.h"
#include<iostream>
#include<cmath>
#include<cstdlib>
#include<iomanip>
#include<fstream>
#include<sstream>
#include<string>

using namespace std;

const int Q = 9;      //D2Q9 模型
const int Nx = 151;
const int Ny = 151;

int         e[Q][2] = {{0,0}, {1,0}, {0,1}, {-1,0}, {0,-1}, {1,1}, {-1,1}, {-1,-1}, {1,-1}};
double    w[Q] = {4.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/9, 1.0/36, 1.0/36, 1.0/36, 1.0/36};
double    rho[Nx+2][Ny+2], u[Nx+2][Ny+2][2], f[Nx+2][Ny+2][Q], F[Nx+2][Ny+2][Q];
double    tau_f, dx, dy, dt, Lx, Ly, niu, G, fai[Nx+2][Ny+2], Fm[Nx+2][Ny+2][2], totalmass0, totalmass;
int         i, j, k, ip, jp, m;

void SetPrameter()
{
      dx = 1.0;
      dy = 1.0;
      Lx = dx*double(Nx-1);
      Ly = dy*double(Ny-1);
      dt = dx;
      tau_f = 0.8;
```

```
    niu = (tau_f - 0.5)/3.0;
    std::cout <<"niu = "<<niu<<endl;
}

double feq(int k,double rho,double u[2]) //计算平衡态分布函数
{
    double eu,uv,feq;
    eu = (e[k][0]*u[0]+e[k][1]*u[1]);
    uv = (u[0]*u[0]+u[1]*u[1]);
    feq = w[k]*rho*(1.0+3.0*eu+4.5*eu*eu-1.5*uv);
    return feq;
}

void init()
{
    totalmass0 = 0;
    for(i=0; i<=Nx+1; i++)   //流场及密度场初始化
      for(j=0; j<=Ny+1; j++)
      {
          u[i][j][1] = 0.0;   u[i][j][0] = 0.0;
          int range = 10;
          int min =0;
          int r = rand()&range + min;   //生成随机数
          rho[i][j] = 1.0 - 0.01*double(r)/50.0;   //初始时刻密度施加随机扰动
          totalmass0 += rho[i][j];
      }
}

void compute_interaction_force() //计算伪势相互作用力
{
```

```
for(i=0; i<=Nx+1; i++)
   for(j=0; j<=Ny+1; j++)
   {
       fai[i][j] = 1.0*exp(-1.0/rho[i][j]);
   }
for(i=1; i<=Nx; i++)
   for(j=1; j<=Ny; j++)
   {
       G = -10.0/3.0;
         Fm[i][j][0] = 0.0;
         Fm[i][j][1] = 0.0;
         for(k=1; k<Q; k++)
       {
           Fm[i][j][0] += (-G)*fai[i][j]*3.0*w[k]*fai[i+e[k][0]][j+e[k][1]]*e[k][0];
           Fm[i][j][1] += (-G)*fai[i][j]*3.0*w[k]*fai[i+e[k][0]][j+e[k][1]]*e[k][1];
       }
   }

   for(j=1; j<=Ny; j++)
   {
         Fm[0][j][0] = Fm[Nx][j][0];
         Fm[0][j][1] = Fm[Nx][j][1];
        Fm[Nx+1][j][0] = Fm[1][j][0];
         Fm[Nx+1][j][1] = Fm[1][j][1];
   }
   for(i=0; i<=Nx+1; i++)
   {
         Fm[i][0][0] = Fm[i][Ny][0];
         Fm[i][0][1] = Fm[i][Ny][1];
        Fm[i][Ny+1][0] = Fm[i][1][0];
```

```
                Fm[i][Ny+1][1] = Fm[i][1][1];
            }
}

void init_micro() //分布函数初始化
{
    for(i=0; i<=Nx+1; i++)
        for(j=0; j<=Ny+1; j++)
            for(k=0; k<Q; k++)
            {
                f[i][j][k] = feq(k, rho[i][j], u[i][j]);
            }
}

void evolution()
{
      for(i=0; i<=Nx+1; i++) //碰撞步
            for(j=0; j<=Ny+1; j++)
                for(k=0; k<Q; k++)
                {
                    double Force[Q], ux, uy, Fx, Fy;
                    ux = u[i][j][0];
                    uy = u[i][j][1];
                    Fx = Fm[i][j][0];
                    Fy = Fm[i][j][1];
                    Force[k] = (1.0 - 0.5/tau_f)*w[k]*3.0*( (e[k][0]-ux)*Fx + (e[k][1]-uy)*Fy
                        + 3.0*(e[k][0]*ux + e[k][1]*uy)*( e[k][0]*Fx + e[k][1]*Fy) );
                    F[i][j][k] = f[i][j][k] - (f[i][j][k] - feq(k,rho[i][j],u[i][j]))/tau_f + Force[k];
                }
```

```
for(i=0; i<=Nx+1; i++) //迁移步
    for(j=0; j<=Ny+1; j++)
       for(k=0; k<Q; k++)
       {
            ip = i - e[k][0];
            jp = j - e[k][1];
           if( ip>= 0 && ip<=Nx+1 && jp>=0 && jp<=Ny+1)
           {
                f[i][j][k] = F[ip][jp][k];
           }
       }

//边界处理
  for(j=1; j<=Ny; j++)
    for(k=0; k<Q; k++)
    {
           f[0][j][k] = f[Nx][j][k];
           f[Nx+1][j][k] = f[1][j][k];
    }
  for(i=0; i<=Nx+1; i++)
    for(k=0; k<Q; k++)
    {
           f[i][0][k] = f[i][Ny][k];
           f[i][Ny+1][k] = f[i][1][k];
    }

totalmass = 0;
for(i=0; i<=Nx+1; i++) //计算宏观密度
   for(j=0; j<=Ny+1; j++)
   {
```

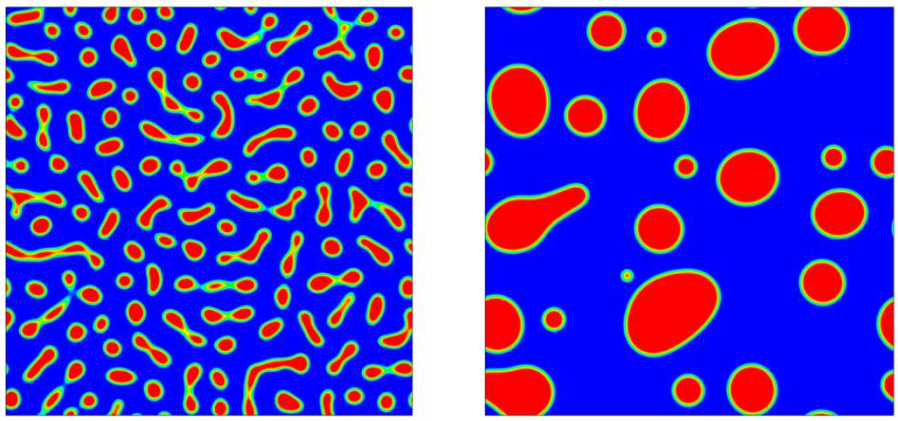
```
            rho[i][j] = 0;
            for(k=0; k<Q; k++)
            {
                rho[i][j] += f[i][j][k];
            }
            totalmass += rho[i][j];
        }

    compute_interaction_force(); //更新伪势相互作用力

    for(i=0; i<=Nx+1; i++) //计算宏观速度
        for(j=0; j<=Ny+1; j++)
        {
            double Utemp1, Utemp2;
                Utemp1 = 0;    Utemp2 = 0;
             for(k=0; k<Q; k++)
             {
                Utemp1 += e[k][0]*f[i][j][k];
                Utemp2 += e[k][1]*f[i][j][k];
             }
             u[i][j][0] = (Utemp1 + 0.5*(Fm[i][j][0] ))/rho[i][j];
             u[i][j][1] = (Utemp2 + 0.5*(Fm[i][j][1] ))/rho[i][j];
        }
}

void output(int m) //输出
{
        ostringstream name;
        if(m==0)
        {
```

```
            name<<"Phase_sepa0000"<<m<<".dat";
        }
        if(m>1 && m<100)
        {
            name<<"Phase_sepa000"<<m<<".dat";
        }
        if(m>=100 && m<1000)
        {
            name<<"Phase_sepa00"<<m<<".dat";
        }
        if(m>=1000 && m<10000)
        {
            name<<"Phase_sepa0"<<m<<".dat";
        }
        if(m>=10000 && m<100000)
        {
            name<<"Phase_sepa"<<m<<".dat";
        }
      ofstream out(name.str().c_str());
       out<<"Title =\"LBM Phase_sepa\"\n"<<"VARIABLES = \"X\", \"Y\",\"rho\"\n"
          <<"ZONE T=\"BOX\", I="<<Nx+1<<", J="<<Ny+1<<", F=POINT"<<endl;
       for(j=1; j<=Ny+1; j++)
          for(i=1; i<=Nx+1; i++)
          {
             out<<double(i-1)/Lx<<" "<<double(j-1)/Ly<<" "<<rho[i][j]<<endl;
          }
}
int main()
{
      using namespace std;
```

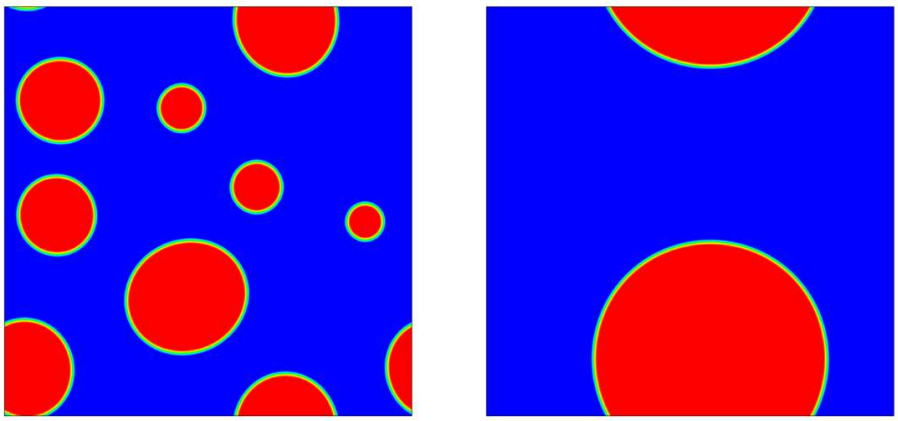
```
    SetPrameter();
    init();
    compute_interaction_force();
    init_micro();
    for(m=0;m<=80000; m++)
    {
        evolution();
        if(m%200 == 0)
       {
            cout<<"The "<<m<<"th computation result:"<<endl<<"The total mass ratio is:"
<<setprecision(7)<<totalmass/totalmass0<<endl;
            output(m);
        }
    }
    return 0;
}
```

3.3 数值模拟结果

图 1 给出了相分离过程的动态演化过程，对应的时刻分别为 $t=100\delta_t$ 、$400\delta_t$ 、$2000\delta_t$ 和 $50000\delta_t$ 。由于初始时刻的密度处在热力学不稳定区域且施加了一个微小扰动，因此很快就会发生相分离，在 $t=100\delta_t$ 时刻可以看到许多大小不一且形状不规则的液滴。随着时间的推移，一些液滴会与其周围的液滴融合而形成更大的液滴。此外，在系统达到稳定之前，会一直发生气液两相的转变（与蒸发及冷凝现象类似），因此可以看到一些小液滴的消失以及一些大液滴的长大，直至系统达到稳定平衡，形成一个大液滴（参见 $t=50000\delta_t$ 时刻）。程序中输出了系统瞬态总质量与初始时刻系统总质量的比值，由于模拟中的气液界面为扩散界面，而初始时刻并不存在界面，因此这一比值会在 1.0 附近有非常微小的波动。当系统稳定时，比值同样趋于稳定，约为 0.99928。



(a) $t=100\delta_t$　　(b) $t=400\delta_t$



(c) $t=2000\delta_t$　　(d) $t=50000\delta_t$

图 1　基于伪势多相模型的相分离模拟