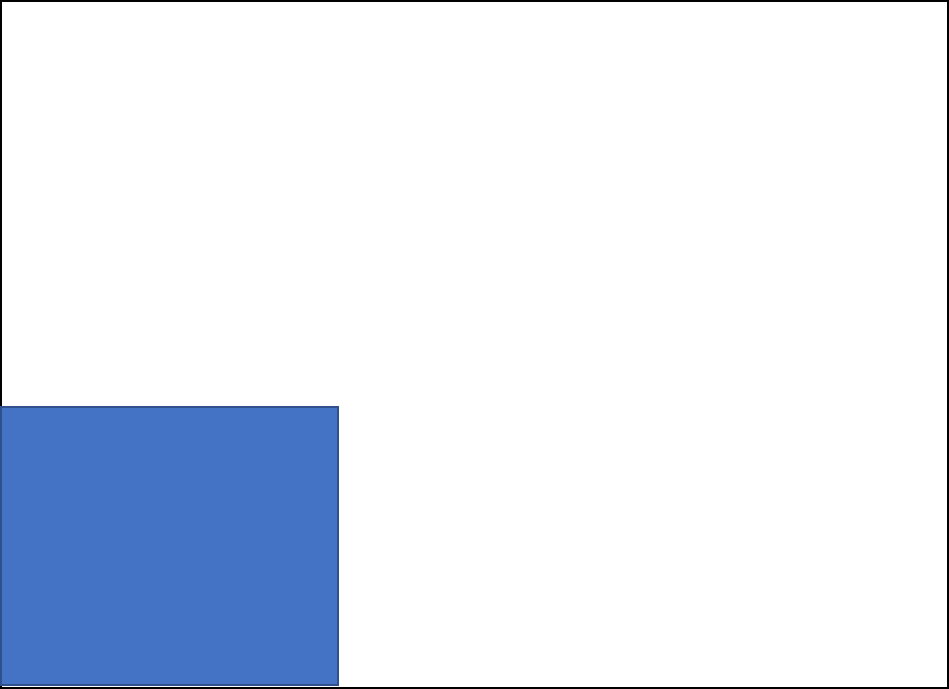
# 1程序概述

二维SPH小程序，模拟溃坝问题。

如图，蓝色方块为水。初始时设置如图，程序开始后水方块溃散。（注：实际上水方块初始化在中间了）



编程语言：taichi

采用弱压缩的SPH

# 2程序主要模块说明

主要步骤包含：

1. 定义核函数
2. 邻域搜索
3. 计算密度
4. 计算压力
5. 计算力和加速度
6. 时间推进
7. 处理边界碰撞
8. 初始化
9. 人工粘性

附录：

1. 邻域搜索具体实现
2. 时间步的选定
3. 给定密度的选定

## 2.1 核函数和其一阶导数二阶导数

**函数/模块名称**：kernelFunc()

**输入**： r，为从当前粒子与相邻粒子j的距离

**作用**： 计算核函数值

可以采用标准核函数

Poly6函数

**输出**：核函数值

**函数/模块名称**：firstDW (r)

**输入**： r，为从当前粒子与相邻粒子j的距离

**作用**： 计算核函数一阶导的值

这里采用的是spiky核函数的一阶导数。由于标准核函数的一阶导数并不是单调的。核函数的导数和核函数本身可以采用不同函数。

**输出**：核函数一阶导数值。

**注意**：这里不定义梯度，而只定义导数。是因为梯度是个二维向量，返回值定义较为麻烦。实际上只要把一阶导数乘以单位方向向量，就得到了梯度。

**函数/模块名称**：secondDW (r)

**输入**： r，为从当前粒子与相邻粒子j的距离

**作用**： 计算核函数二阶导的值

采用的是spiky核函数的二阶导数。

**输出**：核函数二阶导数值。

**注意:** 拉普拉斯算子返回的就是个标量，因此拉普拉斯算子和二阶导数完全一致。

## 2.2 邻域搜索

**函数/模块名称**：neighborSearch()

**输入**：double currentParticleID 当前粒子的ID

**作用**：根据当前粒子找到其周围粒子。

**输出**：周围粒子的ID（多个） 。

**注**：因粒子位置存储在全局变量position中，故只要给出ID即可找到对应位置，即一个哈希表。具体邻域搜索算法需要采用gridBased搜索建立背景网格。

## 2.3 计算密度

**函数/模块名称**：computeDensity()

**输入**： mass, 粒子质量(全局常量)

**作用**：计算密度场，计算公式如下：

**输出**：密度场Density（全局场）。

## 2.4 计算压力

**函数/模块名称**：computePressure()

**输入**： 密度场density；给定密度density0；常量参数EosCoeff()；常量参数EosExponent()

**作用**：根据理想气体状态方程（EOS）计算压力场。计算公式如下：

其中取50.0，EosExponent（gamma）取7.0。参数借鉴自splishSplash

同时，为了处理表面粒子（表面粒子会造成负压力且压力梯度力会很大），要先保证

**输出**：压力场（全局场）。

## 2.5 计算力和加速度

分为三个子模块：粘性力, 压力梯度力和加速度

### 2.5.1 计算粘性力

**函数/模块名称**：computeViscosityForce ()

**输入**：velocity，速度场（全局场）；viscosity, 粘度（全局常量）

**作用**： 计算粘性力，公式为：

**输出**：粘性力场viscosityForce（全局场）

### 2.5.2 计算压力梯度力

**函数/模块名称**：computePressureGradientForce ()

**输入**：pressure，压力场（需辅助函数额外计算，全局场）

**作用**： 计算压力梯度力，公式为：

**输出**：压力梯度力场pressureGradientForce（全局场）

### 2.5.3 叠加力并计算加速度

**函数/模块名称**：computeAcceleration()

**输入**：已计算出的两个力场：pressureGradientForce; viscosityForce

重力加速度gravity

旧加速度场acceleration

**作用**： 计算加速度，公式为：

**输出**：加速度（全局场）

## 2.6 时间推进

**函数/模块名称**：advanceTime ()

**输入**：旧位置场position；旧速度场velocity；计算得到的加速度acceleration

**作用**： 向前推进时间步。利用欧拉显式时间积分更新粒子位置和速度，公式为：

**输出**：新速度场velocity和加速度场position（全局场）

## 2.7 处理边界碰撞

**函数/模块名称**：boundaryCollision()

**输入**：粒子当前位置position，边界位置，粒子速度velcotiy，弹性恢复系数restitutionCoeff; 摩擦系数frictionCoeff

**作用**： 判断粒子是否穿过壁面边界，如果穿过，则将粒子位置挪到最近的边界点处，速度修正为：

**输出**：粒子新速度场velocity和新位置场position

## 2.8 初始化

**函数/模块名称**：initialization()

**输入**：无

**作用**：按照初始条件初始化如下场：

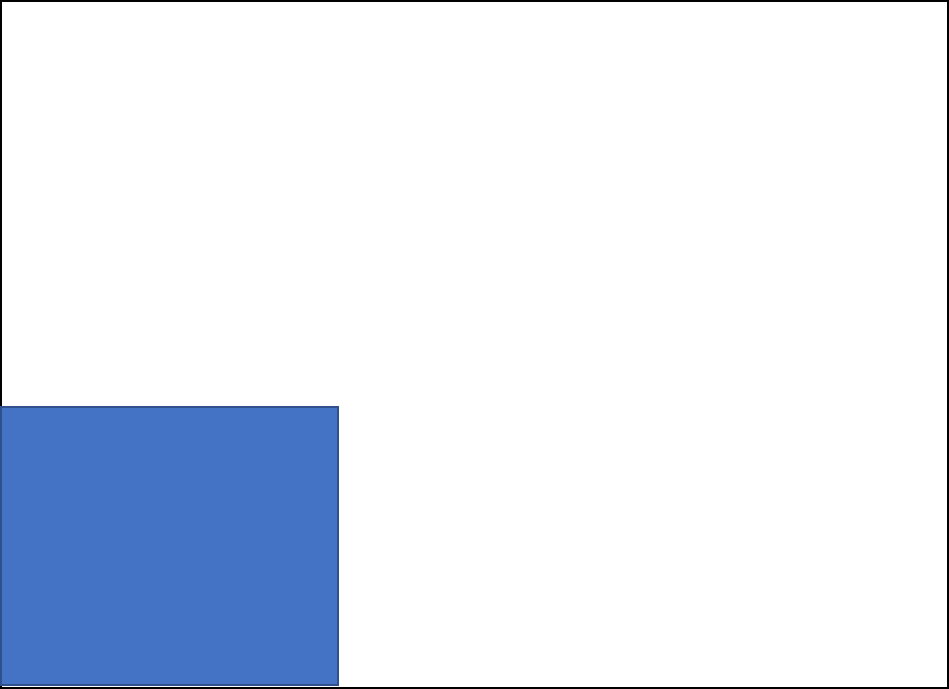
位置场position：按照粒子初始排列位置给值，为图示方块。

速度场velocity：全0

密度场density：无需给定，由computeDensity根据m计算

压力场pressure：无需给定，按照EOS自动计算。

其余场如压力梯度力和加速度等均自动初始化为0。



**输出**：粒子新速度场velocity和新位置场position

**注意**：我们定义了两种初始化，一种是**整齐排列**的，一种是随机排列的。随机排列的是有BUG的，因为它可能会导致粒子离得非常近，一开始会被飞快地弹开。所以需要用整齐排列的初始化。

## 2.9 人工粘性

**函数/模块名称**：pseudoViscosity()

**输入**：无

**作用**：迟缓速度过快的粒子。如果不加人工粘性，可能会导致粒子非常“飘”。

zhang

Doyub kim

然后将速度在中线性插值。

# 3 变量说明

## 3.1 全局常量

粒子数目 numPar

粒子静止密度 density0

核半径 kernelRadius

粒子质量 mass

整个计算区域大小（方形） boundX 和boundY

水区域初始化大小 waterBoundX 和 waterBoundY

水区域初始化位置(左下角) waterPosX和waterPosY

弹性恢复系数 restitutionCoeff

摩擦系数 frictionCoeff (暂未考虑)

EOS参数(指数) EosExponent

EOS参数(乘数） EosCoeff

粘度 viscosity\_mu

时间步 dt

## 3.2全局场

物理量场

场大小均为 numPar

位置场 position 2D

速度场 velocity 2D

密度场 density 1D

压力场 pressure 1D

加速度场 acceleration 2D

压力梯度力场pressureGradientForce 2D

粘性力场 viscosityForce 2D

邻域搜索相关变量

maxNumNeighbors 邻域粒子最大数目 常量

maxNumParInCell 一个网格内最大粒子数目 常量

numParInCell 某网格内当前包含粒子数目 1D场

neighbor 邻域粒子链表 1D场

numNeighbor 邻域粒子数目 1D场

cell2Par 网格ID转换为粒子ID的哈希表 1D场

par2Cell 粒子ID转换为网格ID的哈希表 1D场

# 附录A 邻域搜索

算法为基于网格的邻域搜索

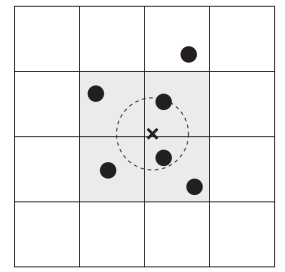
注：这里的网格只是一种邻域搜索时使用辅助的背景网格，不是真正的网格。不要和网格法或者混合法里的网格搞混。

原理大概就是：

我们想要求解的问题：知道粒子的编号（parID），想求和它在相邻网格的粒子的编号（neighbor）。

因为只要知道相邻网格内的粒子，就只需要搜索相邻网格而不用搜索所有粒子了。

如图：



图中x代表当前粒子，虚线圆圈代表其邻域（也就是核半径内区域），黑点代表其他粒子。

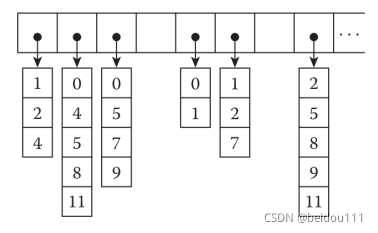
可见，只有相邻四个网格内的粒子有可能是在邻域内的，这样就不必搜索其他粒子，大大节约了计算量。

## 具体实现1

设置两个数组（两个哈希表）：

数组1（particleID2CellID[n]）：particleID作为下标（哈希表的键），cellID作为存储的值（哈希表的值）。

数组2（cellID2particleID[m]）：cellID作为下标（哈希表的键），particleID作为存储的值（哈希表的值，有多个）。这是个二维数组，其第二维长度是不定的，其形状示意图如下：



其第一维代表的是网格，按网格编号排列，第二位是该网格内的粒子。

n代表粒子总数，m代表网格总数。

其中cellID不用真正建立网格，只要根据Cartisian 坐标得到即可

假设计算域大小为boundX, boundY

于是每行排列的网格数目为：particleNumPerRow=boundX/cellSize

其中cellSize可以设置为2倍核半径：cellSize=2\*kernelRadius

根据当前粒子位置可以计算cellID：

cellID=floor(position[particleID].x/cellSize)+floor(position[particleID].y/cellSize)\*particleNumPerRow

floor代表舍弃小数点后，只保留整数

即为一行一行排列，每行满了之后再往上排列。

每次时间步开始的时候，先更新这两个数组。

更新方法：particleID2CellID即根据上面计算的公式得到cellID，与此同时，把相应的粒子编号放入cellD2ParticleID。

于是回到原问题：已知粒子ID求相邻粒子。

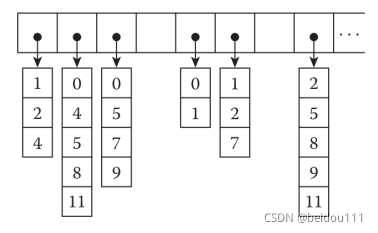
1. 根据particleID找到所在网格cellID

2. 根据网格cellID找到其相邻的四个网格，cellID, cellID-1, cellID-particleNumPerRow, cellID-particleNumPerRow-1（这里是为了和图示相符，具体要根据位置小数点后面的数字再做判断）

3. 在这四个网格内计算和当前粒子距离，假如在核半径内，就认为它在邻域内，可以存入一个邻域链表（neiborList）。

注：维持邻域链表是为了以后访问邻域信息方便。

邻域链表也是这样的形状，只不过现在第一维度代表的是粒子，之前第一维代表的是网格。



这个算法的计算量都在于维持这两个哈希表了，所以每个时间步的算法复杂度为2n

## 具体实现2

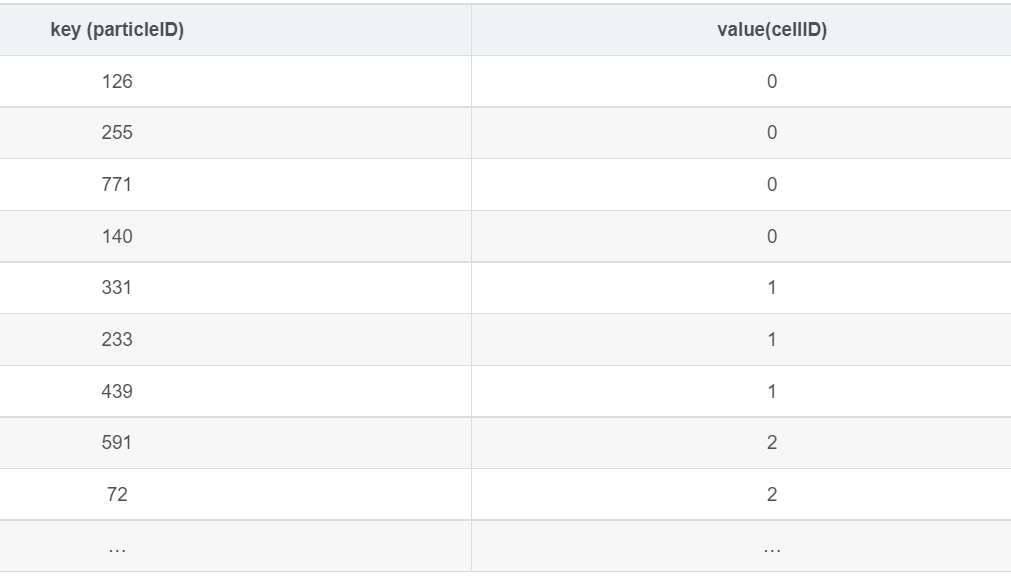
可以不用维持两个哈希表，只需要一个即可。但是每个时间步需要进行排序。

由于需要排序，又要能随机访问，可以用类似于c++的map来实现。

该哈希表的键为粒子ID。值为cellID

每个时间步开始的时候，按照cellID排序

排序之后的表:



当已知粒子编号需要检索邻域粒子的时候：

1. 根据粒子编号，直接索引到网格编号

2. 根据网格编号，在该哈希表内向上向下搜索，直到cellID!=当前粒子的cellID。同样地，对相邻的四个网格都这么处理。

3. 在这四个网格内计算和当前粒子距离，假如在核半径内，就认为它在邻域内，存入邻域链表（neiborList）。

和上面的算法相比，每个时间步需要进行一次搜索，同时还需要进行上下双向搜索。我们就姑且假设双向搜索只需要搜索10次吧（因为一个网格超过10个粒子就太密了）。算法复杂度为nlogn+10\*m。而恰好我们假设的是一个网格内有10个粒子，所以其实这里可以大致认为10\*m=n。所以可以认为算法复杂度为nlogn+n

# DEBUG注意点汇总

## DEBUG开关

首先，Debug的时候，最好开启一些开关

ti.init(arch=ti.cpu, debug=True, excepthook=True,

cpu\_max\_num\_threads=1, advanced\_optimization=False)。

开关的含义在博客中有说明。

## DEBUG技巧1：测试函数与DEBUG FLAG

为了测试和debug，编写了几个测试函数，测试函数以TEST开头，采用 python函数

### 1 TESTKernel()

用于测试核函数和核函数的导数。利用matplotlib.plot画出图像，信息存入csv文件中。

### 2 TESTTwoPar ()

用于测试仅仅两个颗粒的力、加速度、密度、压力等信息是否计算正确。

两个颗粒的力初始位置手动给出，相隔仅半个核半径。

可以在advanceTime中注释掉更新position那一行以实现保持固定。

还可在GUI中画出这两个颗粒。

### 3 TESTPrintNeighbor(i)

打印邻域颗粒信息。

接受一个参数，即当前颗粒i的编号。

打印出它的邻域有几个粒子，分别是谁，所在位置和速度。

### 4 TESTPrintParticleInfo (i)

打印颗粒信息。

接受一个参数，即颗粒i的编号。

打印出该颗粒的密度、压力、速度、位置。

### DEBUG FLAG

设置一个全局布尔常量DEBUG，用于开启和关闭打印DEBUG信息。

使用：if DEBUG == True:

打印语句

这样做的好处是DEBUG结束不用手动删除print语句了。

## DEBUG技巧2：利用GUI

### 随时暂停

设置一个paused[None]变量，可以实现随时暂定。具体用法可以参考taichi官方例子mass\_spring\_game.py

大致用法如下：

# press space to pause

if e.key == gui.SPACE:

paused[None] = not paused[None]

# press s to step once

elif e.key == 's':

for s in range(100):

step()

if not paused[None]:

for s in range(100):

step()

还可以暂停之后再只运行一步

# press s to step once

elif e.key == 's':

for s in range(1):

step()

### 检测单个粒子

DEBUG过程中，可以着重关注某一个粒子的位置和参数。比如粒子0。

首先，将其标红（注意画单个粒子是circle，画多个粒子是circles）

gui.circle(pos[0], 0xDC143C, radius=3.0)

注意taichi的field不能作为GUI函数参数，所以要用position.to\_numpy()转为numpy数据

其次，在屏幕上打印其信息

gui.text(content="position[0]={}".format(position[0]),

pos=(0, 0.95),

color=0x0)

## BUG1清空数组

较为简单的BUG，原因是在使用+=的时候没有先清空。

需要清空的有各种力、加速度、密度、邻域链表。

可以在使用前清空，也可以集中写一个clear()函数。这里采用后者，利用了taichi 的fill函数。但要注意fill函数不能在taichi-scope中使用，所以clear()是个python函数。

## BUG2边界网格编号的计算

边界处网格编号会出现问题，原因叙述如下：

假设网格尺寸为4.0，计算域大小为100.0X100.0。

计算得到每行有25个网格，共25行。

从0开始排列，排列的示意图如下。最后一个编号是624

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 600 | 601 |  |  | 624 |
| 575 |  |  |  | 599 |
| … | … | … | … | … |
| 25 | 26 | … | 48 | 49 |
| 0 | 1 | … | 23 | 24 |

计算网格编号

计算公式：cellID=[posX/4]+[posY/4] \*25

其中[]表示向下取整

举例：

坐标（1，2），（5，2），（2，5），（100，1），（100，100），（1，100）

对于(1，2），位于最左下角，计算为[1/4]+[2/4]\*25=0 正确

对于(5，2），位于左下角第一排第二个，计算为[5/4]+[2/4]\*25=1 正确

对于(2，5），位于左下角第二排第一个，计算为[2/4]+[5/4]\*25=25 正确

对于(100,1), 位于右下角，计算为[100/4]+[1/4]\*25=25错误 实际为24

对于(100,100), 位于右上，计算为[100/4]+[100/4]\*25=650错误 实际为624

对于(1,100), 位于左上角，计算为[1/4]+[100/4]\*25=625错误 实际为600

**错误的原因如下：**

当粒子在上边界和右边界的时候，cell ID的计算会超出网格总数

**具体原因如下：**

为了方便起见，我们考虑无量纲化之后的坐标，即（posX / cellSize, posY / cellSize）

假设粒子在右上角点的网格内，

当坐标为24.9，24.9的时候，落入到网格（24，24）中，其ID恰好为24+24\*25=624，正确

但是当坐标为25, 25的时候，仍应落入到网格（24，24）=624中，但是其ID计算为 25+25\*25=650超出范围，导致错误

同样地，当坐标为25，0的时候，仍应落入到（24，0）=24，但是其ID计算为25+0\*25=25导致错误

故当粒子恰好落到右边和上边的时候，应当想办法将其仍然归纳入内部网格。

---

**解决方案**

我们考虑在处理上边界的时候，给一个较小量eps=0.01，当位置超过boundY-eps的时候，认为该粒子超出的上边界，将其挪到boundY-eps（而非boundY， 因为挪到boundY会恰好触发BUG）。

此外，我们还应该修正计算公式为

cellID=[posX/4 – 0.5]+[posY/4 – 0.5] \*25

这里的0.5就是为了防止超出边界而特意设置的。

根据新公式，我们重新计算下

对于(1，2），应为0 计算为[1/4 -0.5] + [2/4 -0.5 ]\*25 =0 正确

对于(5，2），应为1 计算为[5/4 - 0.5] + [2/4 - 0.5]\*25 = 0 错误

对于(2，5），应为25 计算为[2/4 - 0.5] + [5/4 - 0.5]\*25 = 0 错误

对于(100,1), 应为24 计算为[100/4 - 0.5]+ [1/4 - 0.5]\*25 =24 正确

对于(100,100), 应为624 计算为[100/4 - 0.5]+ [100/4 - 0.5]\*25=624 正确

对于(1,100), 应为600 计算为[1/4 - 0.5] + [100/4 - 0.5]\*25=600 正确

可见，这样不会导致out of bound的错误，但是所计算出来的网格ID并不是我们期望的。但是这样并不会导致错误，因为网格编号只要满足拓扑结构，能计算出邻居编号就够了。

实质上，

## BUG3 核半径的选择

核半径不能随意选择，尤其是**不能小于1.0**

观察核函数

当h取很小的值，例如0.01的时候，会导致W非常大。

尤其是对于W的一阶导数和二阶导数，在系数的分母部分会出现h，这会导致其值格外的大，进而导致压力梯度力非常大，进而粒子会飞快地移动，很容易发散。

所以必须令h>1。在douyub Kim的代码中，h=10.0或者3.0

在taichi SPH 代码中 h=1.1

此外，对W进行一阶导以后，显然系数前面会出现负号，所以W的一阶导数都是负数

## BUG4 物理参数时间步长、密度和质量

物理参数会影响流场的表现和是否崩溃。

首先是**时间步长**，囿于弱可压SPH的天然缺陷，一般要把时间步长设置到1e-4以下。时间步越小，更新的就越慢，长时间盯着发现只走了一点点，因此需要计算100步画一次，这样动画时间就是0.01s一帧。但是计算量就会增大，通常需要开启GPU。

其次是密度和质量的值会严重影响计算的正确性，不正确的设置会导致程序崩溃。**质量**参数还对动画的表现影响很大。**质量越大，粒子就越“散”**，质量调小，粒子就会重叠到一起。

**静止密度**更是决定了程序是否崩溃。尤其是对于外围粒子。这时由于他们核半径内粒子数目少，所以密度会偏小（密度就是邻域粒子核函数的加权和，权就是粒子质量）。而压力梯度力会除以密度的平方，所以会外围粒子导致受力很大。就会有爆炸开的效果，很容易会发散。

一种解决方案是：对于密度小于静止密度的粒子，强行令其压力为0，这样他们就不会受到压力梯度力。所以**静止密度效果上相当于密度的一个下限**。密度小于这个下限时，就不计算压力梯度力中相应的粒子。

具体实现时，SPliSHSPlash的做法是：在计算压力那一步，设置密度为静止密度和计算出的密度的较大的那一个。Doyub Kim的代码是：计算完压力后，假如算出负值，就乘以一个negativePressureScaling，其实这个negativePressureScaling就是0。两者的实现其实是一样的。

## BUG5 初始化

初始化的时候，不应该使用random初始化，因为可能会导致粒子离得非常近而令其迅速弹开。就像炸开一样。