

# WCSPH踩坑指南

2021. 12

李春蕾

---

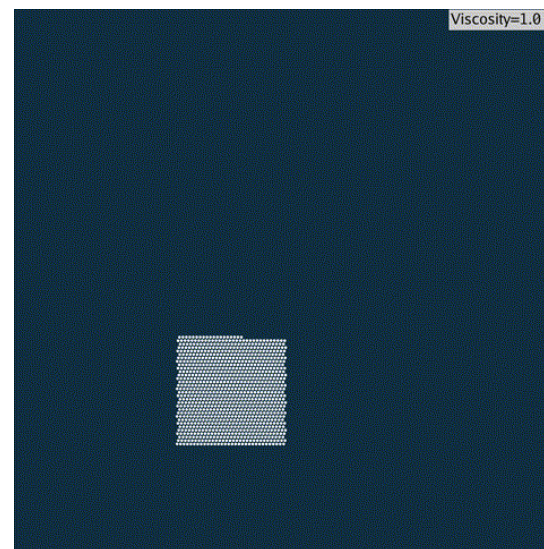
## □ 本文讲：

- WCSPH通俗概念解释 （个人理解）
- WCSPH踩坑指南

## □ 本文不讲：

- SPH的具体原理
- SPH的数学推导 （请参考刘天添老师和张铭睿助教的）
- 如何写代码

## □ 面向对象：小白



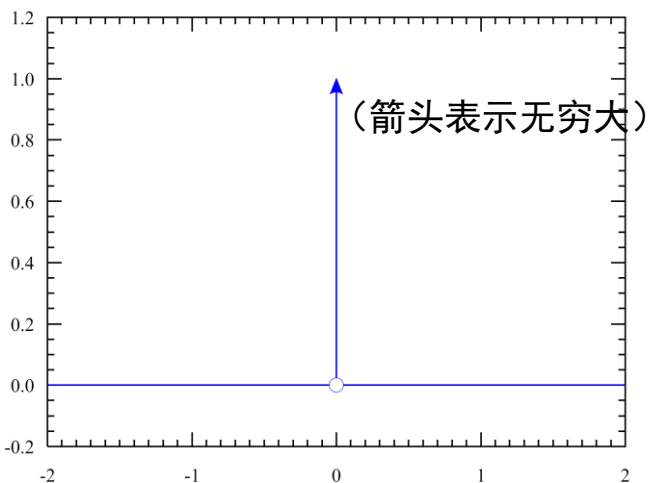
# Part I 基于Q&A的通俗概念解释

# Q1： 为什么叫SPH? (smoothed如何理解?)

A: smoothed particle hydrodynamics, 后两个词好理解, 关键是smoothed怎么理解。

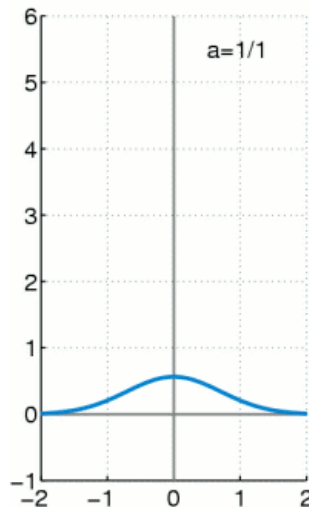
我认为有两种理解:

1. “光滑了的, 缓和了的。” 即把脉冲的狄拉克函数给“缓和”掉。



原本的Dirac函数

Dirac函数: “质点”概念在数学上的表示



缓和后的Dirac函数的辅助函数

$$\delta_a(x) = \frac{1}{a\sqrt{\pi}} e^{-x^2/a^2} \quad a \rightarrow 0$$

高斯函数的方差趋于0的时候的极限就是Dirac函数

Dirac函数的性质

积分得1  $\int_{-\infty}^{\infty} \delta(x) dx = 1$

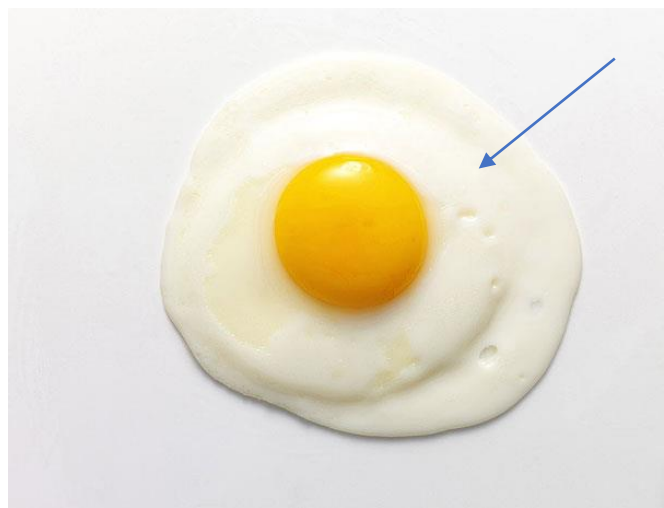
“脉冲”  $\delta(x) = \begin{cases} +\infty, & x = 0 \\ 0, & x \neq 0 \end{cases}$

注: 极限趋于Dirac函数的函数称为Dirac函数的辅助函数。有很多种, 比如右图的高斯函数就是一种。我们在SPH采用的核函数也是一种Dirac函数的辅助函数!

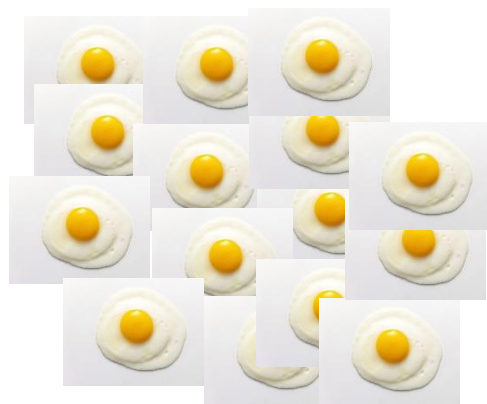
# Q1: 为什么叫SPH? (smoothed如何理解?)

2. “抹平了的，摊平了的。”

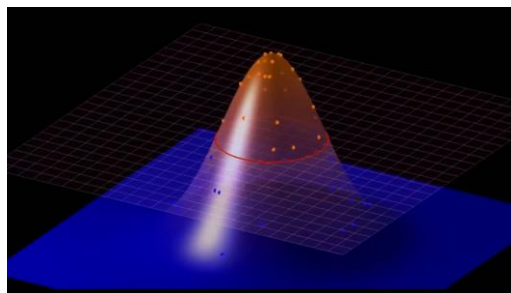
从另一种角度来看，SPH其实是一种空间离散的技术。它只采样一些质点，**质点**上携带了**所有的物理信息**（密度压力速度等）。而“质点”的概念是一个点，不占据空间的，需要把它“摊平”成一个圆（3维是球）。就像摊鸡蛋一样。



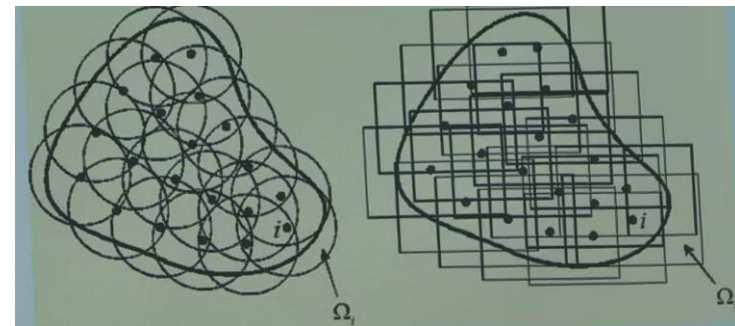
质点摊平后变成支撑域



支撑域之间相互影响



二维的支撑域  
(高度代表值的大小)



支撑域甚至可以是方的!

支撑域=影响域

# Q2: WCSPH与其他SPH相比有什么优缺点？

A：一叶障目，不知秋。如果只知道WCSPH，就会以为其他SPH也都是这样的，实则不然。

WCSPH与其他SPH的差异仅仅在一个地方：计算压力的方式。

| 简称     | 全称                                     | 中文名        | 计算压力的方式                 |
|--------|--|------------|-------------------------|
| WCSPH  | weakly compressible SPH                | 弱可压SPH     | 状态方程（Equation of state） |
| PCISPH | predictor-corrector incompressible SPH | 预测校正不可压SPH | 关于压力的迭代循环的以保证密度不变       |
| IISPH  | implicit incompressible SPH            | 隐式不可压SPH   | 求解压力泊松方程以最小化散度          |
| DFSPH  | divergence-free SPH                    | 无散度SPH     | 结合以上两者                  |

从上到下复杂度依次增大，性能依次增大。

性能最差，最简陋的：WCSPH

性能最好，最复杂的：DFSPH

 最简单，这就是采用WCSPH的核心理由。

注：场景越复杂，复杂方法的性能优越性越好。

# Q3：为什么压力是特殊的？

□ 从NS方程组来看，压力就是一个十分“不合群”的物理量。

注：关于xx = 未知量是xx

$$\begin{array}{c} \text{关于速度的} \quad \text{关于速度的} \quad \text{关于压力的} \quad \text{关于速度的} \\ \downarrow \quad \downarrow \quad \uparrow \quad \downarrow \\ \boxed{\frac{\partial u}{\partial t}} + \boxed{u \cdot \nabla u} = \boxed{-\frac{1}{\rho} \nabla p} + \boxed{\nu \nabla^2 u} \quad \longrightarrow \text{动量方程} \\ \quad \quad \quad \boxed{\nabla \cdot u} = 0, \quad \longrightarrow \text{连续性方程} \\ \quad \quad \quad \uparrow \\ \quad \quad \quad \text{关于速度的} \end{array}$$

从物理上考虑，压力也十分特殊。

不那么严谨地思考一个问题：哪个力最有可能成为驱动流体运动的力？

所有对流体微团产生影响的力：粘性力、重力、压力。

→ 只有压力，能担此大任！

粘性力：可以认为是一种阻尼。

重力：常数，永远不变。

PS：其他力（表面张力、电磁力...）都暂时不考虑

## Q4: 为什么WCSPH不是物理的？

---

A: （仅供讨论）WCSPH最大的缺陷在于：它本来是处理不可压流体的，但是却借用了可压流体的公式，即理想气体状态方程（EOS）。

回想起高中化学课上的理想气体状态方程： $p = \rho RT$

WCSPH中借用了这个公式，改写为： $p = k_{Eos} \left( \left( \frac{\rho}{\rho_0} \right)^\gamma - 1 \right)$

WCSPH：虽然不对但好用

其中 $k_{Eos}$ 和 $\gamma$ 是两个参数，SPlishSPlash中取50和7。

最大的问题在于：理想气体状态方程的**适用条件**：理想气体。

什么是理想气体？越接近高温高压，越离液体状态远，可压缩性越大，越接近于“理想气体”。这正好与不可压缩流体的性质相反。因此，模拟液体绝对不可能使用理想气体状态方程。（不过可压问题EOS可以适用，比如航空航天绝大部分问题）。这属于“张冠李戴”——“张”是可压问题，“李”是不可压问题。



## Q5: $\rho$ 与 $\rho_0$ 有什么区别?

---

### □Q5: $\rho$ 与 $\rho_0$ 有什么区别?

A: 一句话, 前者是计算过程量, 后者是真实物理参数 (比如水是1000kg/m<sup>3</sup>)。

之所以过程中首先要计算 $\rho$ , 是因为SPH插值公式要用:

$$f(r) \approx \sum_j V_j f(r_j) W(r - r_j, h) = \sum_j \frac{m_j}{\rho_j} f(r_j) W(r - r_j, h)$$

那为什么用这个 $\rho$ 而不是用 $\rho_0$ ?

因为不可压流体求解器属于压力基求解器, 先对密度放宽要求, 然后再保证密度的不变性。可以认为计算过程中计算密度不断追逐着真实密度。

真实的物理世界中, 对不可压流体来说, 流体密度永远都是 $\rho_0$ 不变。

$\rho_0$  又被称为是静止密度 (rest density), 因为是在初始静止的时候测得的密度。

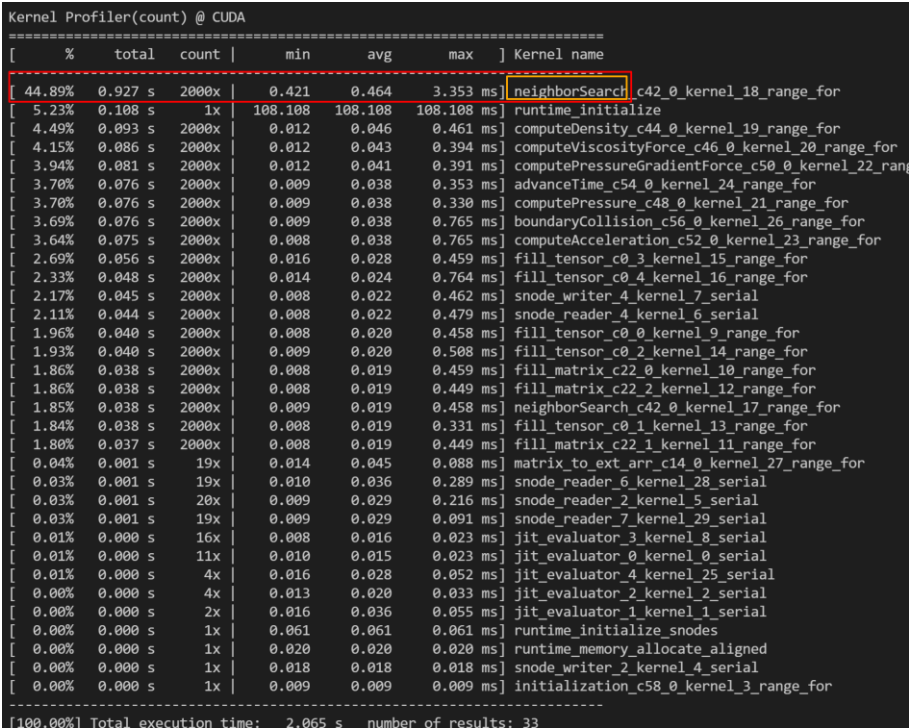
# Q6: 邻域搜索怎么做？

A: 当你去写SPH程序，第一个遇到的问题就是邻域搜索。即：知道当前粒子位置，怎么知道它核半径范围内的邻居都是谁？

方法1：基于网格的链表搜索

方法2：四叉树（3D:八叉树）

方法3：KD树（与四叉树类似）  
（K-Dimension Tree）



|   | %      | total   | count | min     | avg     | max        | Kernel name  |
|---|--------|---------|-------|---------|---------|------------|--|
| [ | 44.89% | 0.927 s | 2000x | 0.421   | 0.464   | 3.353 ms   | neighborSearch_c42_0_kernel_18_range_for               |
| [ | 5.23%  | 0.108 s | 1x    | 108.108 | 108.108 | 108.108 ms | runtime_initialize                                     |
| [ | 4.49%  | 0.093 s | 2000x | 0.012   | 0.046   | 0.461 ms   | computeDensity_c44_0_kernel_19_range_for               |
| [ | 4.15%  | 0.086 s | 2000x | 0.012   | 0.043   | 0.394 ms   | computeViscosityForce_c46_0_kernel_20_range_for        |
| [ | 3.94%  | 0.081 s | 2000x | 0.012   | 0.041   | 0.391 ms   | computePressureGradientForce_c50_0_kernel_22_range_for |
| [ | 3.70%  | 0.076 s | 2000x | 0.009   | 0.038   | 0.353 ms   | advanceTime_c54_0_kernel_24_range_for                  |
| [ | 3.70%  | 0.076 s | 2000x | 0.009   | 0.038   | 0.330 ms   | computePressure_c48_0_kernel_21_range_for              |
| [ | 3.69%  | 0.076 s | 2000x | 0.009   | 0.038   | 0.765 ms   | boundaryCollision_c56_0_kernel_26_range_for            |
| [ | 3.64%  | 0.075 s | 2000x | 0.008   | 0.038   | 0.765 ms   | computeAcceleration_c52_0_kernel_23_range_for          |
| [ | 2.69%  | 0.056 s | 2000x | 0.016   | 0.028   | 0.459 ms   | fill_tensor_c0_3_kernel_15_range_for                   |
| [ | 2.33%  | 0.048 s | 2000x | 0.014   | 0.024   | 0.764 ms   | fill_tensor_c0_4_kernel_16_range_for                   |
| [ | 2.17%  | 0.045 s | 2000x | 0.008   | 0.022   | 0.462 ms   | snode_writer_4_kernel_7_serial                         |
| [ | 2.11%  | 0.044 s | 2000x | 0.008   | 0.022   | 0.479 ms   | snode_reader_4_kernel_6_serial                         |
| [ | 1.96%  | 0.040 s | 2000x | 0.008   | 0.020   | 0.458 ms   | fill_tensor_c0_0_kernel_9_range_for                    |
| [ | 1.93%  | 0.040 s | 2000x | 0.009   | 0.020   | 0.508 ms   | fill_tensor_c0_2_kernel_14_range_for                   |
| [ | 1.86%  | 0.038 s | 2000x | 0.008   | 0.019   | 0.459 ms   | fill_matrix_c22_0_kernel_10_range_for                  |
| [ | 1.86%  | 0.038 s | 2000x | 0.008   | 0.019   | 0.449 ms   | fill_matrix_c22_2_kernel_12_range_for                  |
| [ | 1.85%  | 0.038 s | 2000x | 0.009   | 0.019   | 0.458 ms   | neighborSearch_c42_0_kernel_17_range_for               |
| [ | 1.84%  | 0.038 s | 2000x | 0.008   | 0.019   | 0.331 ms   | fill_tensor_c0_1_kernel_13_range_for                   |
| [ | 1.80%  | 0.037 s | 2000x | 0.008   | 0.019   | 0.449 ms   | fill_matrix_c22_1_kernel_11_range_for                  |
| [ | 0.04%  | 0.001 s | 19x   | 0.014   | 0.045   | 0.088 ms   | matrix_to_ext_arr_c14_0_kernel_27_range_for            |
| [ | 0.03%  | 0.001 s | 19x   | 0.010   | 0.036   | 0.289 ms   | snode_reader_6_kernel_28_serial                        |
| [ | 0.03%  | 0.001 s | 20x   | 0.009   | 0.029   | 0.216 ms   | snode_reader_2_kernel_5_serial                         |
| [ | 0.03%  | 0.001 s | 19x   | 0.009   | 0.029   | 0.091 ms   | snode_reader_7_kernel_29_serial                        |
| [ | 0.01%  | 0.000 s | 16x   | 0.008   | 0.016   | 0.023 ms   | jit_evaluator_3_kernel_8_serial                        |
| [ | 0.01%  | 0.000 s | 11x   | 0.010   | 0.015   | 0.023 ms   | jit_evaluator_0_kernel_0_serial                        |
| [ | 0.01%  | 0.000 s | 4x    | 0.016   | 0.028   | 0.052 ms   | jit_evaluator_4_kernel_25_serial                       |
| [ | 0.00%  | 0.000 s | 4x    | 0.013   | 0.020   | 0.033 ms   | jit_evaluator_2_kernel_2_serial                        |
| [ | 0.00%  | 0.000 s | 2x    | 0.016   | 0.036   | 0.055 ms   | jit_evaluator_1_kernel_1_serial                        |
| [ | 0.00%  | 0.000 s | 1x    | 0.061   | 0.061   | 0.061 ms   | runtime_initialize_snodes                              |
| [ | 0.00%  | 0.000 s | 1x    | 0.020   | 0.020   | 0.020 ms   | runtime_memory_allocate_aligned                        |
| [ | 0.00%  | 0.000 s | 1x    | 0.018   | 0.018   | 0.018 ms   | snode_writer_2_kernel_4_serial                         |
| [ | 0.00%  | 0.000 s | 1x    | 0.009   | 0.009   | 0.009 ms   | initialization_c58_0_kernel_3_range_for                |

[100.00%] Total execution time: 2.065 s number of results: 33

图：邻域搜索约占45%计算量

# 方法1：基于网格的链表搜索法

核心思想：借助网格天然的拓扑性。

建立链表：（每个时间步开始）

particle2Cell数组：粒子编号做下标（键/索引），  
将粒子编号转换为网格编号（存储的值）。

cell2Particle数组：网格编号做下标（键/索引），  
将网格编号转换为粒子编号（存储的值）。

PS:本质上是两个哈希表，因此为了节约内存，  
还可以使用紧凑哈希算法。  
具体可见：[Eurographics Tutorial 2019](#)

## 搜索链表：

根据粒子编号找到网格编号，  
从而找到网格内的其他粒子。  
进而找到所有邻居网格的其他粒子，再比较距离。

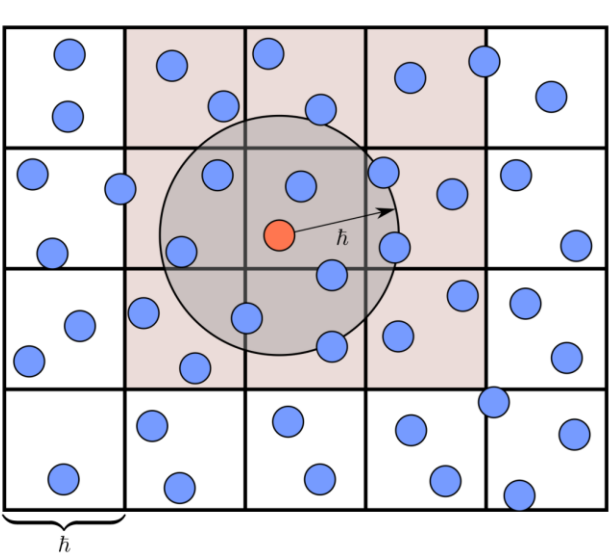


图1：链表搜索的背景网格  
引自Eurographics tut

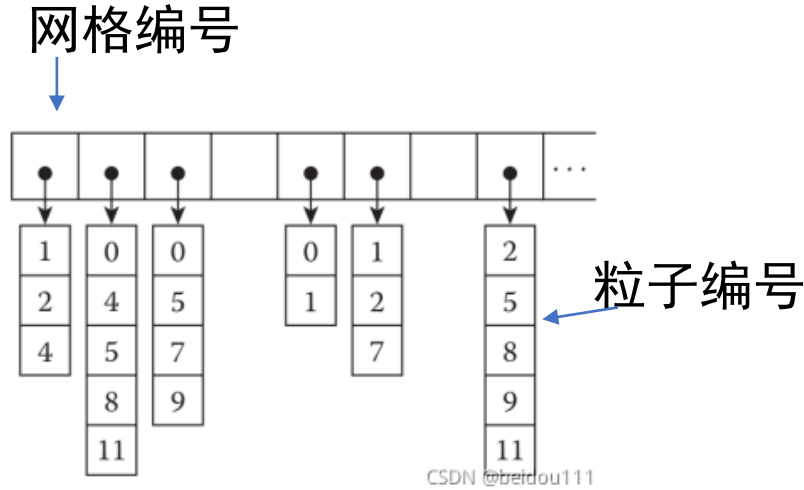


图2：链表搜索的数据结构  
(cell2Particle数组)

注：链表搜索这个背景网格只是用作搜索的，  
与MPM FLIP等用于存储信息的网格不一样。

# 方法2：四叉树 & 方法3：KD树

- 1. **建立树**：先把空间平均分为四个卦象，假如一个正方形中存在多个粒子，那么就继续把该子卦象分为四个卦象，如此反复，直到每个卦象中只有一个粒子。于是就存储成了一棵四叉树。
- 2. **搜索树**：假如想搜索粒子*i*的邻域。以2倍核半径为边长，画出粒子*i*的搜索正方形（如图中阴影）。当阴影正方形与其他粒子的卦象相重叠的时候，就搜索这一卦象。其他的卦象则不必搜索。

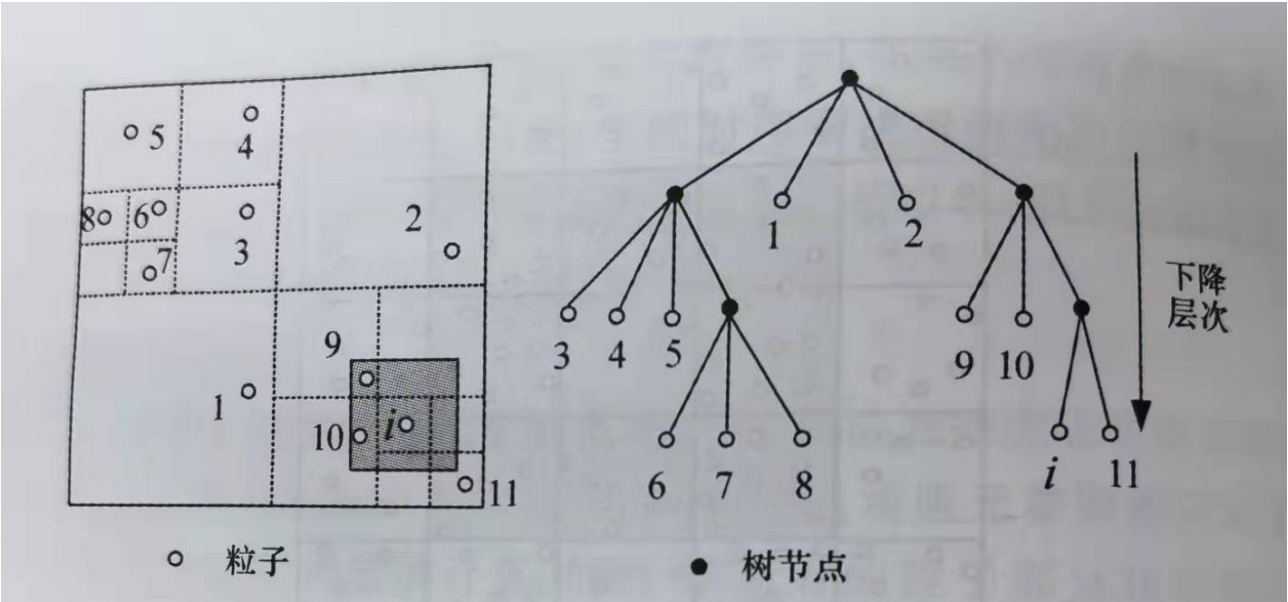


图1：四叉树。左图是空间分割，右图是对应的数据结构。引自刘桂荣，刘谋斌

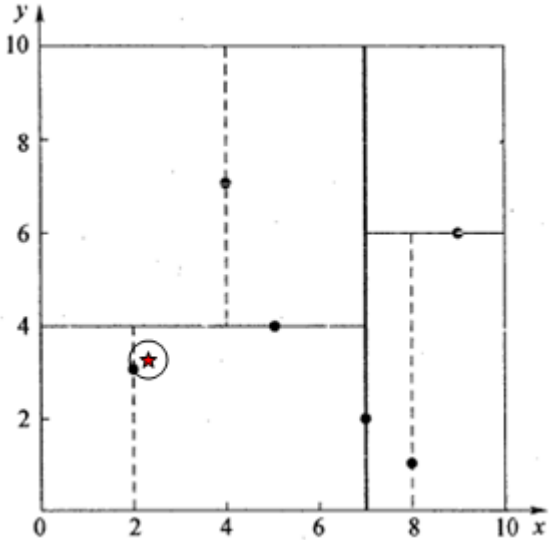


图2：KD树的空间分割

## 方法3： KD树

与四叉树类似。只是分割点选在粒子上，每次只切一刀（平行于x轴或者y轴）。

# Part II WCSPH中可能遇到的BUG

# 显现BUG的技巧

---

## ❑1 开启各种debug开关

```
ti.init(arch=ti.cpu, debug=True, excepthook=True, cpu_max_num_threads=1, advanced_optimization=False)
```

## ❑2 可用于任何情况的print

```
print(“a is {}, b is {}:”.format(a, b))
```

可用于任何情况

## ❑3 妙用assert做断点

在想要停下的位置前`assert 1==2`，相当于打断点了。

假如报出`1==2`的错误，证明前面的代码起码是没有编译错误的。

还可以在前面加上if语句，相当于条件断点了。

# 显现BUG的技巧

## □4 GUI 追踪单个粒子

利用gui.circle可以重复画出单个粒子（如粒子0），给个红色可以突出显示粒子

```
# highlight particle 0 with red
gui.circle(pos[0],
           0xDC143C,
           radius=3.0,)
```

（注意画单个circle是circle而不是circles）

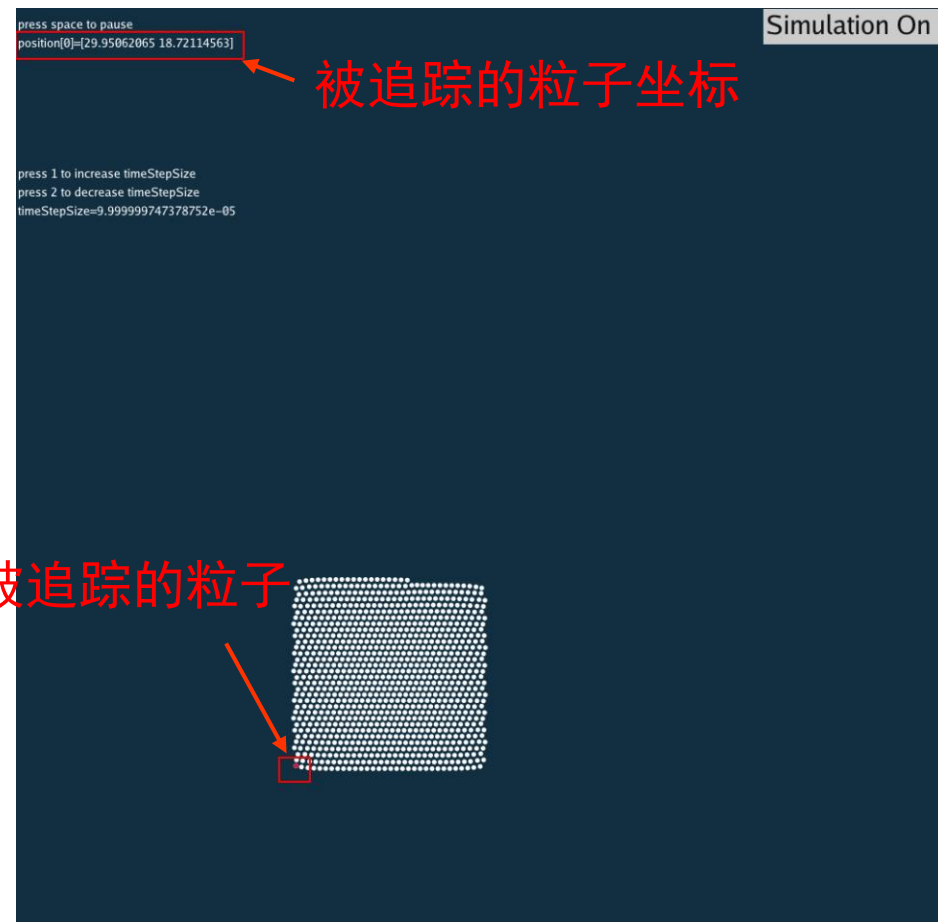
通过GUI.text可以把它的坐标实时显示在GUI上

```
gui.text(content="position[0]={}".format(position[0]),
         pos=(0, 0.97),
         color=0xFFFFF)
```

## □5 GUI 空格键随时暂停

```
if not paused[None]:
    for s in range(100):
        step()
```

```
# press space to pause
elif e.key == gui.SPACE:
    paused[None] = not paused[None]
```



# 显现BUG的前期准备和技巧

## □5 自制TEST函数

```
def TESTPrintParticleInfo(i):
    print('#####print particle info {}#####'.format(i))
    print('density[{}]={}'.format(i, density[i]))
    print('pressure[{}]={}'.format(i, pressure[i]))
    print('position[{}]={}'.format(i, position[i]))
    print('velocity[{}]={}'.format(i, velocity[i]))
    print('#####END print particle info {}#####'.format(i))
```

TEST1 打印出粒子信息

```
def TESTKernel():
    h = kernelRadius

    r = np.zeros(1000)
    y = np.zeros(1000)
    y1 = np.zeros(1000)
    y2 = np.zeros(1000)
    for i in range(1000):
        r[i] = i * h/1000
        y[i] = kernelFunc(r[i])
        y1[i] = firstDW(r[i])
        y2[i] = secondDW(r[i])
    plt.plot(r, y, 'r')
    plt.plot(r, y1, 'g')
    plt.plot(r, y2, 'b')
    plt.show()
    # np.savetxt("y.csv", y, delimiter=',')
    # np.savetxt("y1.csv", y1, delimiter=',')
    # np.savetxt("y2.csv", y2, delimiter=',')
    np.savetxt("kernelFunc.csv", [y, y1, y2], delimiter=',')
```

TEST2 测试核  
函数数值正确

```
def TESTTwoPar():
    # init with only two particles
    # set numPar=2
    # cancel gravity
    gui = ti.GUI("SPHDamBreak",
        background_color=0x112F41,
        res=(1000, 1000)
    )

    distX = 5*kernelRadius/10.0
    distY = 0.0 # 5*kernelRadius/10.0

    position[0] = [
        boundX / 2.0,
        boundY / 2.0,
    ]
    position[1] = [
        boundX/2.0 + distX,
        boundY/2.0 + distY,
    ]

    while not gui.get_event(ti.GUI.ESCAPE, ti.GUI.EXIT):
        for s in range(1):
            step()
        draw(gui)
```

TEST3 仅仅加入两个粒子测试受力

注：采用np.savetxt("kernelFunc.csv", [y, y1, y2], delimiter=',')  
来输出整个流场到TXT



# BUG1 边界网格问题

## 描述

在使用基于网格的邻域搜索的时候，需要通过粒子位置计算出粒子所在网格编号。

假如网格尺寸为4.0，计算域大小为100x100， 那么总共应该有25行，25列网格。

从下到上依次排列，0-624号网格：

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

采用如下公式计算网格编号：

$$\text{网格编号} = \left[ \frac{\text{粒子位置x坐标}}{4.0} \right] + \left[ \frac{\text{粒子位置y坐标}}{4.0} \right] \times 25$$

注：[]表示向下取整

假如粒子跑到了最右上角，坐标（100，100）的位置，那么计算出来的网格编号为：

$[100/4]+[100/4]*25=650$ （？？？超出了网格总数？？）

# BUG1 边界网格问题

在使用基于网格的邻域搜索的时候，需要通过粒子位置计算出粒子所在网格编号。

假如网格尺寸为4.0，计算域大小为100x100， 那么总共应该有25行， 25列网格。

从下到上依次排列， 0-624号网格：

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

采用如下公式计算网格编号：

注： [] 表示向下取整

$$\text{网格编号} = \left[ \frac{\text{粒子位置x坐标}}{4.0} \right] + \left[ \frac{\text{粒子位置y坐标}}{4.0} \right] \times 25$$

假如粒子跑到了最右上角，坐标（100， 100）的位置，那么计算出来的网格编号为：

[100/4]+[100/4]\*25=650（ ??? 超出了网格总数 ??? ）


# BUG1 边界网格问题

---

## 原因

祸根在向下取整上。因为100向下取整恰好是100，而python是从0-99。所以超出范围。100.1同理。因此这个BUG只会出现在粒子运行到右边界和上边界的时候，左下边界却没有问题，因为0和-0.1取整还是0。

## 解决

1. 整体向下平移0.5个单位  网格编号 =  $\left\lfloor \frac{\text{粒子位置x坐标}}{4.0} - 0.5 \right\rfloor + \left\lfloor \frac{\text{粒子位置y坐标}}{4.0} - 0.5 \right\rfloor \times 25$
2. 边界留出空隙eps=0.1（或padding）  
(比如到99.9的时候就认为到了边界并反弹)

# BUG2 物理参数问题

---

## □1 时间步

时间步显然是由于WCSPH的限制，经过测试  $dt < 1e-4$  的时候较稳定。每帧设置100个substeps较为合理。

## □2 静止密度，核半径，粒子半径和粒子质量

静止密度，核半径，粒子半径和粒子质量是相互关联的。具体可看老师答疑。

这里给出一组大致合理值： $\rho_0 = 1.0$ ,  $h = 1.0$ ,  $h_{particle} = 0.25$ ,  $mass = 1.0$

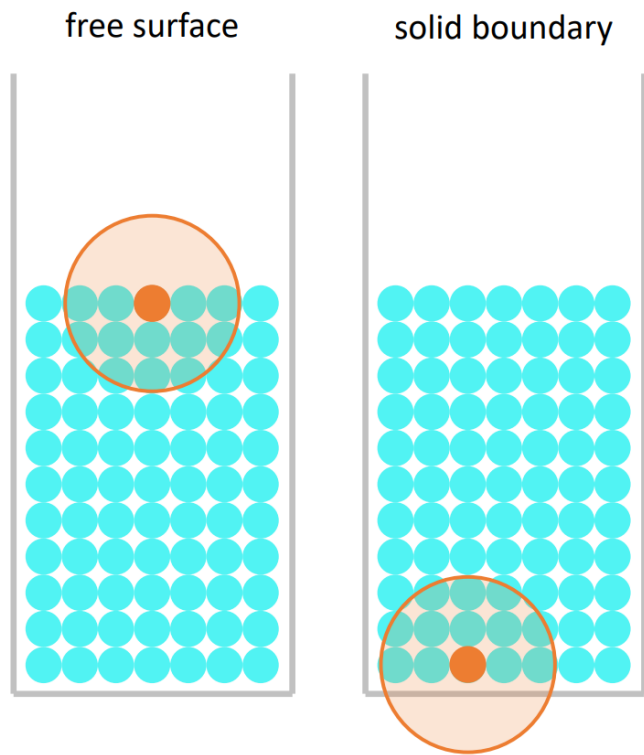
静止密度过高：粒子过密

粒子质量过大：粒子过散

粒子半径：是用于初始排列的。过小则过密排列，会炸开。

# BUG3 边界粒子问题

粒子的边界处会出现支撑域内粒子不足的问题 (particle deficiency)



后果：边界处粒子可能凑在一起。

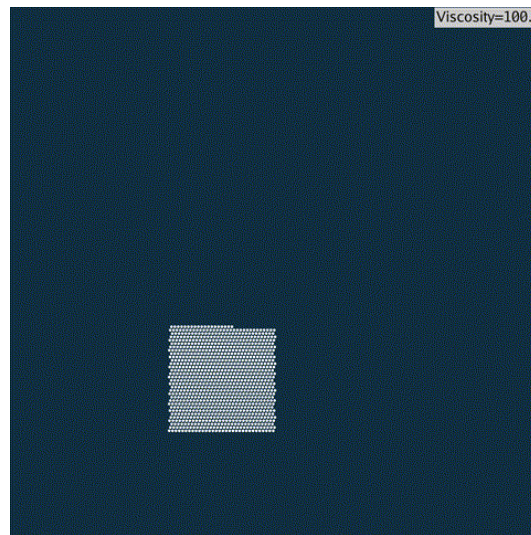
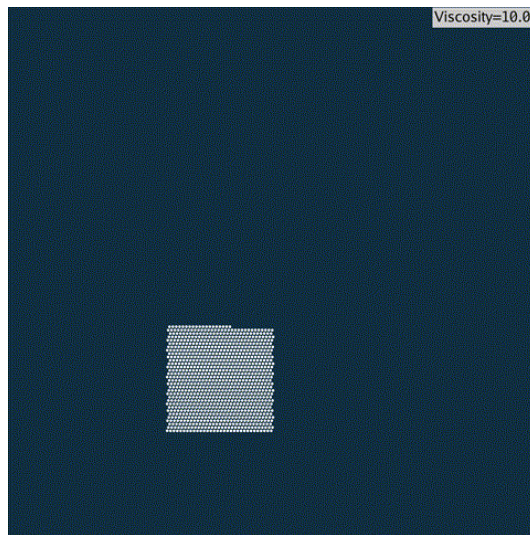
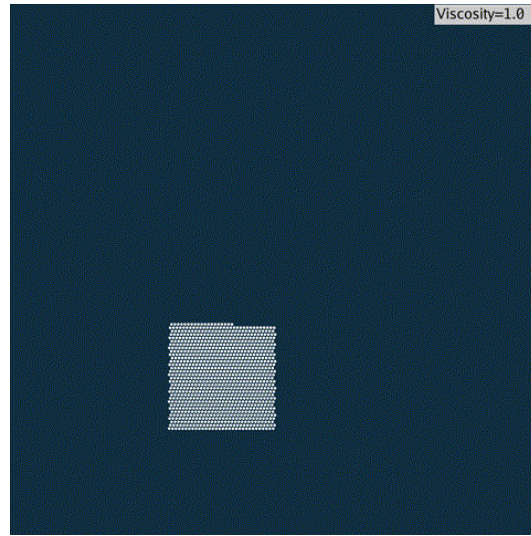
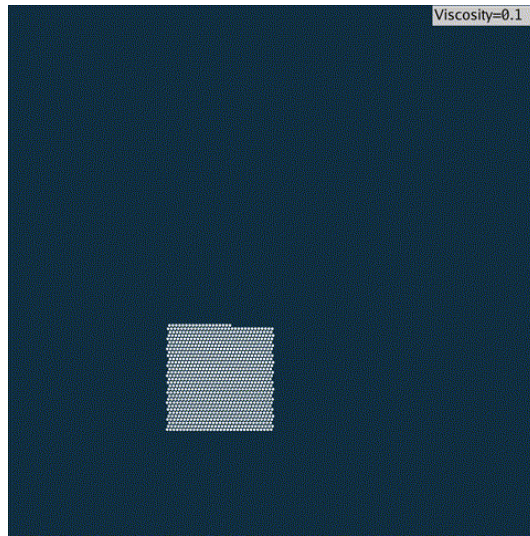
原因：边界处由于粒子少，造成密度算小了。由于核函数插值公式的分母是密度，所以所有力都偏大了。

解决：强行补齐边界处粒子的密度或强行令压力为0。

# 结果展示

---

粘度0.1, 1.0, 10.0, 100.0



# 参考资料

---

## □ 参考资料

- SPLishSplash (开源SPH库以及相应的Eurographics 2019 tutorial)

<https://interactivecomputergraphics.github.io/SPH-Tutorial/>

- 《Fluid Engine Development》, 2017, Doyub Kim. <https://fluidenginedevelopment.org/>

- 《光滑粒子流体动力学：一种无网格粒子方法》，2003，刘桂荣（美国辛辛那提大学） 刘谋斌（北大工学院）

<https://doi.org/10.1142/5340>

- 无网格法（物质点法）：清华大学张雄课题组：[张 雄 教授 \(tsinghua.edu.cn\)](http://zhangxiong.tsinghua.edu.cn)

## □ 本人

博客：[https://blog.csdn.net/weixin\\_43940314](https://blog.csdn.net/weixin_43940314)

代码：<https://github.com/chunleili/WCSPHTaichiHW>

# 对太极的建议

---

❑一定要坚持开设太极图形课！

❑一定要坚持开设太极图形课！

❑一定要坚持开设太极图形课！

完