FSI fish swim by MLS-MPM in Taichi

27 Mar. 2026

Taichi の MLS-MPM を使って双方向 FSI で魚の遊泳をシミュレートしてみる．MLS-MPM は格子と粒子を両方利用する手法．魚も粒子で弾性体として表現する．水は横弾性係数 0 の（粘性の概念が無い．速度の平滑化で粘性があるように見える）粒子．気相は真空で粒子が無い状態だが，格子はある．粒子は移動し（Lagrangian description），格子は移動しない（Euler description）．表面張力の概念はない．SI 単位系の物性値の概念もない（あるんだが正しい挙動を返さない）．スケール（格子＆粒子粗さ）によって変更される（格子サイズによってスケーリングされる）．carangiform による遊泳制御は，内部の粒子間に力をかけ，左右側で内部応力差をつけることによって実現する（材力の梁の曲げと同じ）．衝突計算など近傍の格子単位でしか計算しないので組み合わせ爆発などが起こらず便利.

計算空間：0.4m*0.4m*0.4m

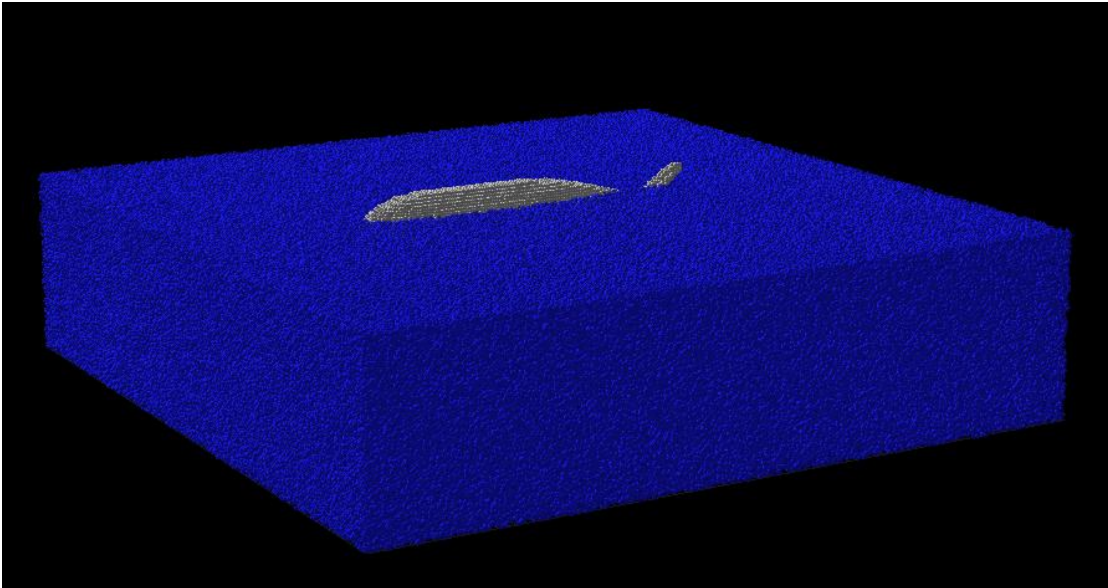
格子解像度：128*128*128→格子サイズ 0.4/64=6.25 [mm]

粒子サイズ：1 格子に 2^3=8 つ

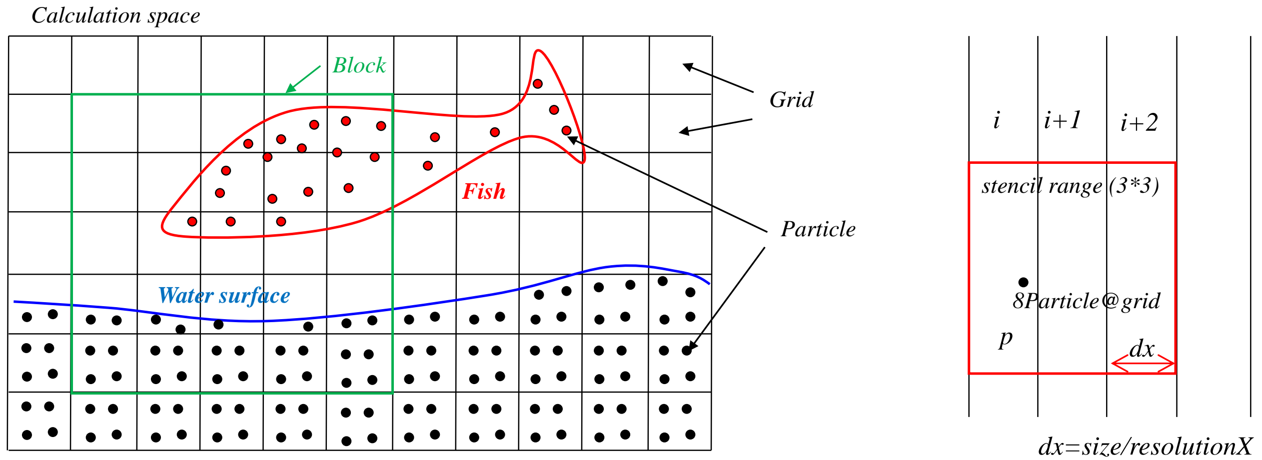
ブロックサイズ：GPU で高速化するための領域範囲．128/4=32．4*4*4 の格子で構成される

水領域：0.4m*0.4m*0.1m

魚：弾性体粒子，0.2m 体長，周波数 10Hz，Carangiform（16 分割制御）



2D で描くと，計算空間はこんなイメージ．粒子法なので，描画においては輪郭が粒子の径程度ボケる．物性値においては，格子 3 つ分で平滑化しているのでその分ボケる．3 格子＝6 粒子．これより，境界層が問題になるような流体力学問題は解析できない.



(1) Carangiform による遊泳

ボディを体軸方向に N 分割し，各ボディ間の境界面に応力を発生させ，その左右差で曲げモーメントを作り出す．背骨によって直接回転モーメントを作り出す方法もあるが，ここでは筋肉の収縮をイメージしてみた．

尾びれ振りの運動は以下の通り．詳細は openfoam トビウオ資料参照．x が体軸方向，y は体側方向．振幅が尾鰭方向に線形に大きくなり，進行波を作り出す．L は体長，A_0 は尾鰭振幅，λ は波長，f は周波数．Carangi では L=λ.

$$y(x,t) = A_0\frac{x}{L}\sin(2\pi(\frac{x}{\lambda} - ft))$$

位置で制御せず，力で制御するので長さの補正は無し．体側方向の y 変位により体長 L が勝手に長くなったりはしない．

今，細長い魚を構造力学の片持ち梁と考えて，体のうねりを曲率 r(x)で表現できると考えると，

$$\frac{1}{r} = \frac{M}{EI} = \frac{d^2y}{dx^2}$$

と書けるので，曲げモーメント M で Carangiform の運動を実現できる．2 回微分すると，

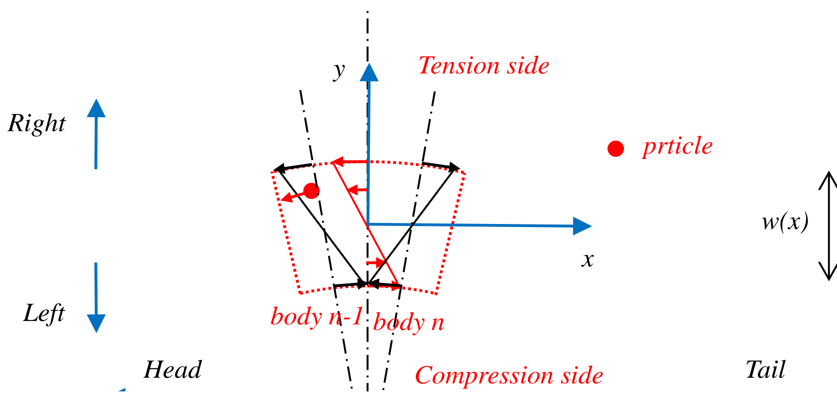
$$\frac{dy}{dx} = \frac{A_0}{L}\sin\left(2\pi\left(\frac{x}{\lambda} - ft\right)\right) + 2\pi A_0\frac{x}{\lambda L}\cos(2\pi(\frac{x}{\lambda} - ft))$$

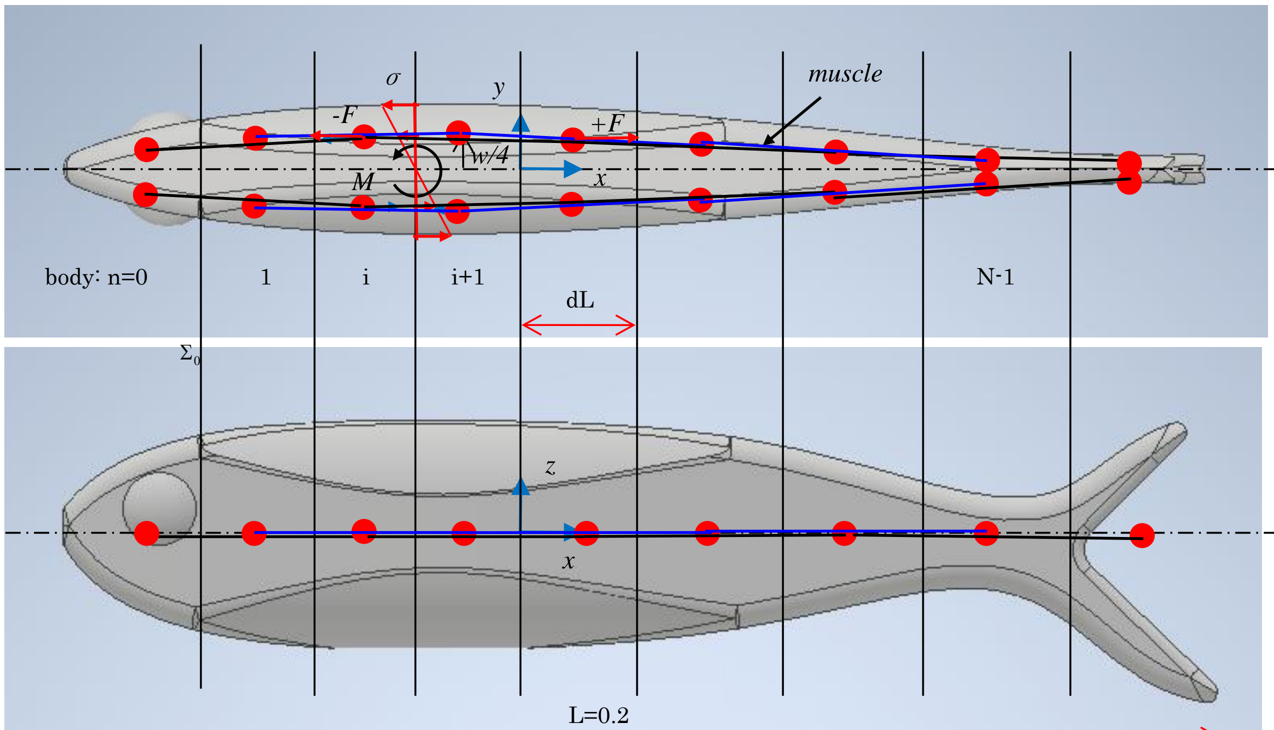
$$\frac{d^2y}{dx^2} = \frac{4\pi A_0}{\lambda L}\cos\left(2\pi\left(\frac{x}{\lambda} - ft\right)\right) - 4\pi^2 A_0\frac{x}{\lambda^2 L}\sin(2\pi(\frac{x}{\lambda} - ft))$$

となるので，

$$M = 2f\frac{w}{4} = EI\frac{d^2y}{dx^2}$$

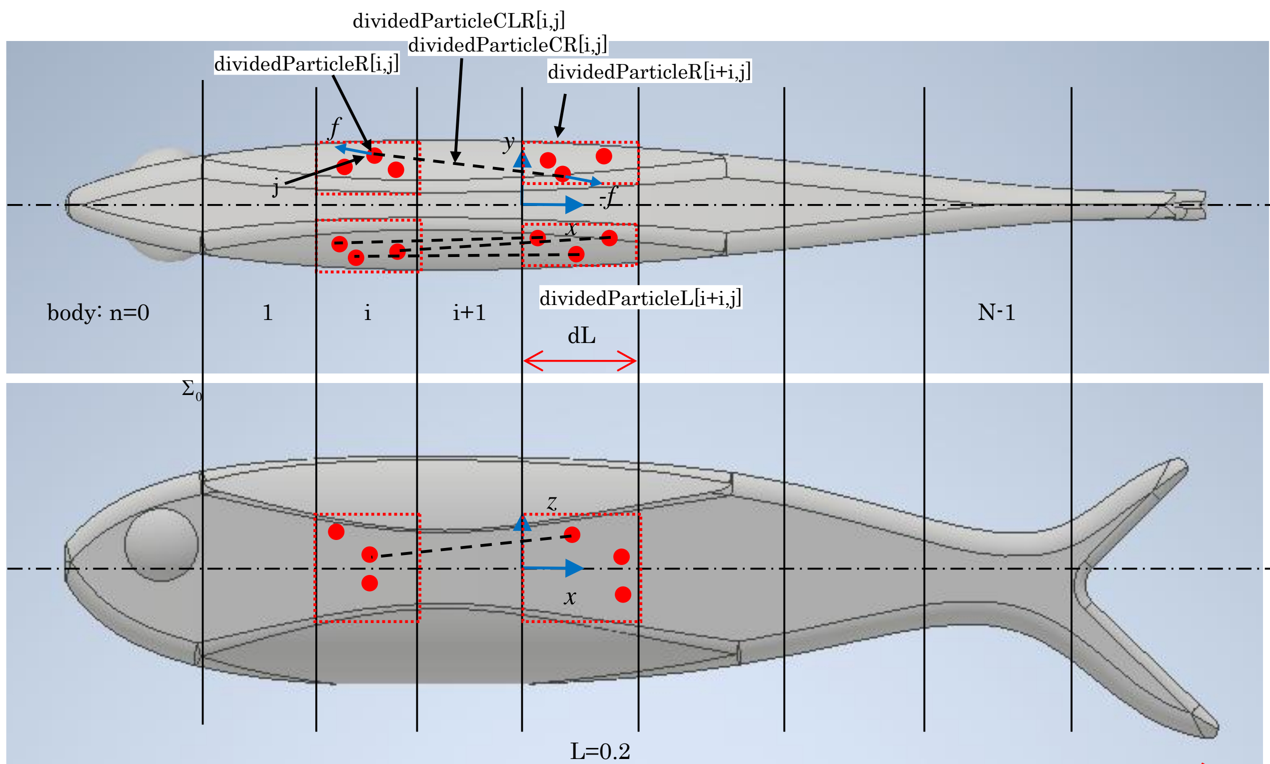
として，筋肉の引張 f で制御する．w は体の幅．体軸から 0.3w あたりの位置にある筋肉が左右二つで引っ張り合うという意味．筋肉は押さないけどここでは押すモデル．魚を構成しているのが離散化された粒子なので，体軸方向に N 分割している．筋肉は 2(N-2)本．切断された面（赤い線のところ）に，曲げモーメントを発生させる応力分布を粒子間の力で作り出している．実際の計算では，図のようにひとつ飛ばしでボディを結合し，筋肉と体軸との平行性を上げている．最後の N-1 番目のボディは適切な位置に粒子が無いので外している．よって実際は N-2 番目のボディまで計算.

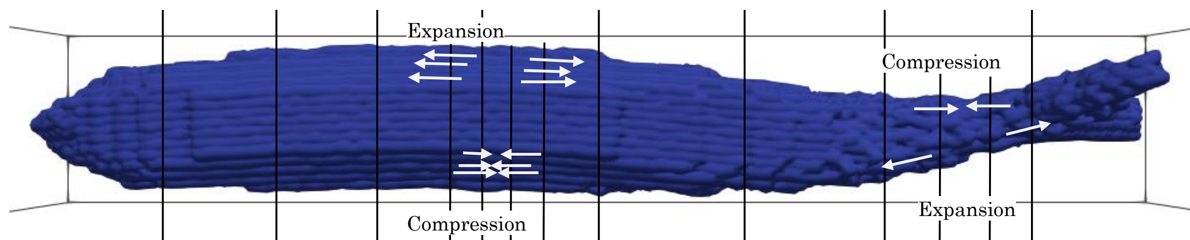




上記の力学モデルを粒子で再現しようとすると以下のようになる．体軸方向に N 分割された領域をさらに右側（図の上）と左側（図の下）に分ける．この領域を体表側に寄せ，モーメントが発生しやすいようにする．その i 番目の領域の粒子と i+2 番目の領域の粒子を結合させ，この間に力を加える．なお，領域間で粒子の数が異なるため，ひとつ前の領域とつながっていない粒子も存在する場合がある．プログラムでは i+2 番目の粒子の個数が i 番目の粒子の個数より少なくなった場合には確率分布でつなげている．

dividedParticleR_len[i]：右側の i 番目の body の中の粒子の数
dividedParticleR[i,j]：右側の i 番目の body の中の j 番目の粒子の全体から見た id
dividedParticleCR[i,j]：右側の i 番目の body の中の j 番目の粒子と力を交換する（作用反作用する）尾鰭側のとなりの body の粒子の body 内番号
dividedParticleCLR [i,j]：右側の i 番目の body の中の j 番目の粒子と i+2 番目の粒子の初期長さ





ちなみに，Anguilliform は，

$$y(x,t) = \frac{A_0}{L}\sin(2\pi(\frac{x}{\lambda} - ft))$$

$$\frac{dy}{dx} = \frac{2\pi A_0}{L\lambda}\cos(2\pi(\frac{x}{\lambda} - ft))$$

$$\frac{d^2y}{dx^2} = -\frac{4\pi^2 A_0}{L\lambda^2}\sin(2\pi(\frac{x}{\lambda} - ft))$$

であり，実際はこっちでプログラムした．尾びれ周りの断面二次モーメントがボディ中心に対して小さいので小さい力で曲がる．

(2) **Taichi & Python アーキテクチャ**

以下は，昭和の C プログラマーから見た世界．読み飛ばしていい．

．大雑把な概念は，CPU のアーキテクチャ（メモリ領域やコード）をつかう python と，GPU のアーキテクチャ（Vulkan/CUDA）をつかう Taichi がある．でもって，GPU コードの CPU マシンで対応できないところを書き換えている．

．SNode（メモリ構造）と field（データ）を分けている設計

C/C++的に言うと：

field ＝ 配列変数

SNode ＝ その配列をどうメモリに並べるかの指定（SNode = **Structural Node（構造ノード）**）

particle = ti.root.dynamic(ti.i, max_num_particles, chunk_size)

重い計算を taichi field 内ではなく，python メモリ内で行うと，計算時間が圧倒的にかかる．

（なお，この chunk_size にバグ（？）があり，segmentation fault を吐き続けていたが修正した．）

三角関数一つとっても，GPU のコードなのか，CPU のコードなのかで呼ぶ出す関数（ちなみに，昭和プログラマーは関数とメソッドをあまり区別していないので注意）

Taichi内部の概念整理

名前	正体	役割
ti.root	ルートSNode	ツリーの根
dense / pointer / dynamic	SNode生成関数	子SNodeを作る
SNode	クラス	メモリ構造ノード
field	クラス	実データ

．メモリの管理の件

Python scope のメモリと Taichi scope のメモリ空間の概念があり，後者は基本確保しっぱなし．これにより，部分的にあらかじめ大きめのメモリをぶつ切りで確保しておく chunk サイズという概念が出てくる．Taichi scope 内でメモリを開放しても python scope（or OS）から見ると解放されていない．

Garbage collect（GC）はされている．呼び出されなくなった list は自動的に GC の対象になる．この辺の概念も C プログラマーには理解しがたい．

．格子（Grid node），ボクセル（Voxel），block，粒子（particle）の関係

- grid（Eulerian approach）
  - 定義：ノードまたはセルで構成され、速度・質量・力など物理量を格納して運動方程式を解く場所。dx（=格子サイズ）が物理解像度を決める。計算空間内で動かない．
  - 役割：時間発展（symplectic Euler）・力学解・接触・CFL 制約の決定。
- voxel（ジオメトリ離散）
  - 定義：三角形や形状を占有する小さな立方体セル。このコードでは physical dx の半分（super_sample=2 → voxel_size = dx/2）。→結局，このプログラムでは使っていない
  - ブロック：4*4*4 の格子で構成されている．GPU による高速化のための領域範囲．solver 内では，粒子番号 p=self.pid[I]のようにアクセスするようになっており，I=[i, j, k, n]の i,j,k はブロック座標，n はブロック内の粒子番号．
- particle（Lagrangian approach）
  - 定義：位置 x、速度 v、質量 m、変形勾配マトリクス F 等を持つ点要素。物理量の「主記憶媒体」。計算空間内で動く．
  - 役割：voxel／mesh から初期化され（あるいは直接生成され）、P2G/G2P を通じて grid と相互作用して運動を表現する。

．GPU 計算における float と double の件

float（f32）は 7 桁程度，double（f64）は 16 桁程度の精度．

計算速度は CPU ではどちらもほぼ同じ．64bit アーキテクチャだから．GPU は double だと 10 倍程度，モデルによっては 30 倍遅くなる．GPU は float アーキテクチャ．
また，GPU 割り算は時間がかかるので同じ値で何度も割る場合には掛け算にして計算している．GPU コードはこの辺の可読性が極めて悪い．

．**Taichi にとっての「コンパイル時」とは**カーネル初回呼び出し時．クラス定義時（python 実行時）ではない．python は，インタプリタ（Basic とか）とコンパイラ（C とか）の両方の概念をもっている言語．JIT ができる．

．並列化に関して．
勝手に並列化している．for 文の中で逐次処理には注意が必要．順次インクリメントしているわけではない．for 文の外で定義された k=0 などを for 文の中で k+=1 しても順番にインクリメントされない．

．python の型の扱い
Python は「変数に型がある」のではなく「オブジェクトに型がある」言語
a = [10]　　　　# これはリスト型
a = a[0]　　　　# リストの 0 番目を a に入れたのでこの時点で a は int 型．リスト型は消滅し，GC の対象になる．
print(a) # 10

C / C++ 的な感覚：

```
int a;
a = 10;    // aは常にint
```

Python：

```
a = 10      # int
a = [1,2]   # list
a = "abc"   # str
```

a はただの「名前（参照）」であり、
型を持っているのは右側のオブジェクト

Taichiの

```
ti.field(...)
ti.Vector.field(...)
```

は完全に静的型です。

なお，()はタプルで，[]はリスト．リストは自由に型変更できる．

(3) **Program 概説**

ソースコード FSIswim00.py が，
ソルバクラス engine/mpm_solver.py を呼び出し，
それが，魚の運動クラス motion.py を呼び出し，
それが，CAD モデルを読み込んで実行している．
engine ディレクトリ内のコードもちょいちょい読んでいるが基本はこの三つ．

(3-1)　MLS-MPM ソルバクラス（engine/mpm_solver.py）

格子（動かない grid node）と粒子（動く particle）を生成し，動力学を計算する．以下，概略．self は省略してある．

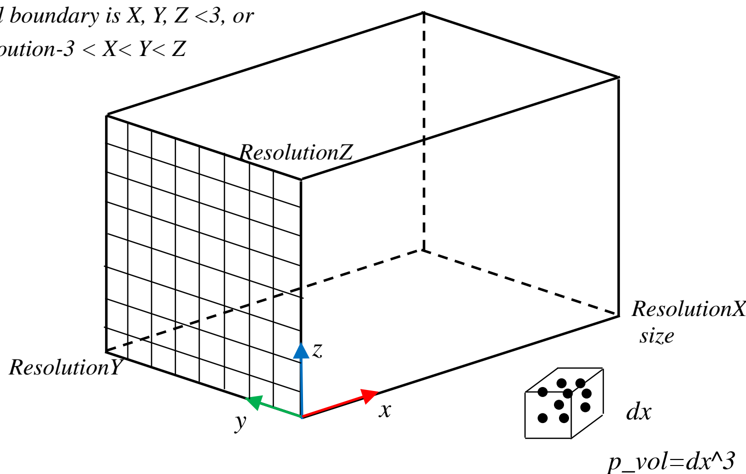
engine/mpm_solver.py/MPMSolver(

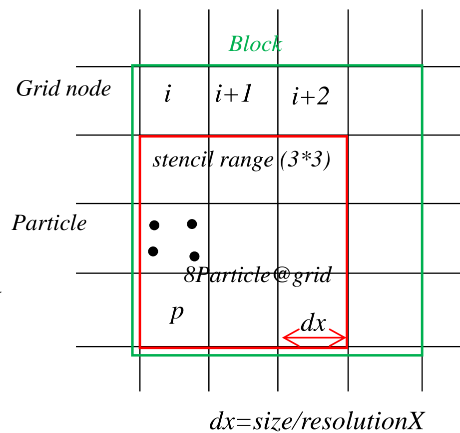
res=(ResolutionX, ResolutionY, ResolutionZ): x, y, z 方向の格子の解像度
size=：x, y, z の空間のサイズ．立方体
max_num_particles=2 ** 23: 2^23≒8.3M 粒子
quant：メモリの管理方法かな．．．． default は False． 以下は False として記載
use_ggui=True
．．．． 略
)

(a) Member variables:

．材料の名称

```
material_water = 0
material_elastic = 1
material_snow = 2
material_sand = 3
material_stationary = 4
….. ここに，複数の弾性体を登録した．
materials = {
    'WATER': material_water,
    'ELASTIC': material_elastic,
    'SNOW': material_snow,
    'SAND': material_sand,
    'STATIONARY': material_stationary,
    ……
}
```





粒子の物理量は，近傍 27 格子で計算している．このため，壁の方向に 3 格子を使ってその方向の衝突を制御するため，壁近傍の境界層の話などはまったくできない．なお壁の衝突もクーロン摩擦を使えない．反力に比例する摩擦を計算すると，dt を非常に細かくしなければならなくなる．1 格子内に粒子は 8 個．ただし，粒子は動くため，格子内に一つも粒子が無い場合もある．なお，実際の計算は block 単位で 4*4*4 格子．

．壁面の境界条件（Boundary projection operator）

```
surface_sticky = 0
surface_slip = 1
surface_separate = 2
surfaces = {
    'STICKY': surface_sticky,
    'SLIP': surface_slip,
    'SEPARATE': surface_separate
```

}

$$\mathbf{Proj}(\mathbf{v}, \mathbf{n}, B, \mu) = \begin{cases} \vec{0}, & B \text{ is sticky}, \\ \mathbf{v}_t, & B \text{ is slip}, \\ \zeta\hat{\mathbf{v}}_t, & B \text{ is separate and } \mathbf{v}\cdot\mathbf{n} \le 0, \\ \mathbf{v}, & B \text{ is separate and } \mathbf{v}\cdot\mathbf{n} > 0, \end{cases} \quad (25)$$

$\mathbf{v}_t = \mathbf{v} - (\mathbf{v}\cdot\mathbf{n})\mathbf{n}$, ：壁面の接線方向速度（tangent），壁の法線方向速度のみ 0．壁面を滑る．

$\zeta = \max(0, |\mathbf{v}_t| + \mu\mathbf{v}\cdot\mathbf{n})$, ：接線方向に速度依存の動摩擦を入れ，逆向きにはしない．押しつけ力依存ではない．つまり，μF ではなく，μv_n．これが一気に計算を加速している．Δt も小さくていい．F=cx'+kΔx は，Δx を小さく取らなくてはならない．

$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{|\mathbf{v}_t|}$.

μ は動摩擦係数，という意味．

．粒子パラメータ

dim = 3 ：次元
grid_size = 4096 : =2^12 default
t ：時間
res=(resolutionX, resolutionY, resolutionZ)：x, y, z 方向解像度
n_particles ：粒子の総数
dx = size / resolution ：格子長さ
inv_dx = 1 / dx ：格子長さの逆数
p_vol=dx^3 ：体積
p_rho = 1000 ：密度 ρ
p_mass = p_vol * p_*rho ：格子（粒子？）の質量
water_SG = water_SG ：比重 specific gravity
elastic_SG = elastic_SG
max_num_particles ：粒子最大数
gravity=(gx, gy, gz) ：重力
source_bound=((x,y,z),(x,y,z))：cube のとき，開始点頂点とサイズベクトル
C[p]= ti.Matrix.field(self.dim, self.dim, dtype=ti.f32): rv/rx マトリクス，移流項
v[p] = ti.Vector.field(dim, dtype=ti.f32) ：float の速度．seed で初期値
x[p] = ti.Vector.field(dim, dtype=ti.f32) ：float の位置．seed で初期値
F[p] = ti.Matrix.field(dim, dim, dtype=ti.f32) ：変形勾配行列＝平行移動と回転．seed で初期値，応力テンソルにより計算される．
material[p] ：i32，材料 seed で代入
color[p] :i32．Splat grid-wise **colored distance field (CDF)**

粒子における衝突計算で使われる SDF（singed distance function）に対して，剛体が n 個になったときに色情報を付加して区別できるように拡張している．Traditional signed distance functions (SDFs：符号付距離関数)は，点が物体の内部か外部化を問い合わせるのに便利．

SDF による衝突判定は「φ(x)<0 で衝突」「∇φ で法線」「−φ で侵入量」という極めて単純で安定なアルゴリズム．

物体境界を signed distance function φ(x) で表す

- φ(x) > 0 ：外部
- φ(x) = 0 ：境界
- φ(x) < 0 ：内部

■ 基本アルゴリズム（粒子 vs 固定境界）

1. 粒子位置 x を評価

   d = φ(x)

2. 衝突判定

   if d < 0 → 貫通（衝突）

3. 法線計算
   $n = \nabla\varphi(x)$
   $n = n / |n|$
4. 位置補正（projection）
   x_new = x - d n
   （d<0 なので外側へ押し出す）
5. 速度補正（反発・摩擦）

法線方向速度：

v_n = (v · n) n
接線方向速度：

v_t = v - v_n
完全反射：

v_new = v_t - restitution * v_n
摩擦あり：

v_t_new = max(0, 1 - μ |v_n| / |v_t|) v_t
v_new = v_t_new - restitution * v_n
連続体（MPM/SPH）での使い方

MPM では通常：

grid node 位置 x_g で

if φ(x_g) < 0:

法線方向速度を除去

つまり「グリッド速度を拘束する」方法がよく使われる。

計算フロー（時間積分付き）

```
for each particle:
x += v dt
d = φ(x)
if d < 0:
n = normalize(∇φ(x))
x -= d n
v = collision_response(v, n)
```

これに対し，SDF を拡張した CDF では，i 番目のグリッド距離情報 d(**x**i)の他に色（color は変数の名前）情報を加えている．色情報は，近傍境界面の集合と，**x**i がどの面に位置するか，を記号化している．これで複数の境界面を表現している．

5.2.1 Grid unsigned distance. 格子符号なし距離

Rigid particles
各方向の剛体境界面 r

補助剛体粒子：$r_1, r_2, \ldots, r_R$: R は全剛体粒子数

基本指標 ε（r_η）：おそらく，剛体境界面への写像

u_{i,r_η} ：i 番目の格子の剛体粒子 rη の境界面までの符号付距離,

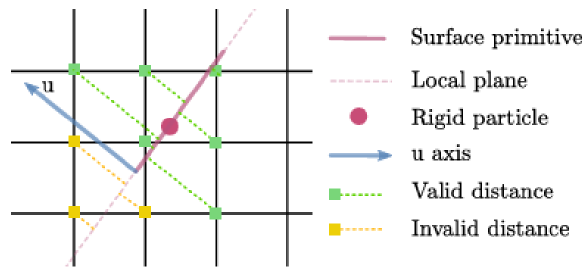


Fig. 11. Splatting the unsigned distance field from a rigid particle on a segment to 9 grid nodes. The u axis is the normal to the plane defined by the primitive. The value of u_{i,r_η} for each grid node thus represents the signed point-plane distance between grid node i and the plane that rigid particle r_η lies on. Note that such a distance is only considered to be valid (or existing) if the projection onto the plane actually lies *inside* the primitive geometry.

緑が有効距離，黄色は無効距離

distance rasterization
i 番目の格子の粒子で最も境界面に近いものを保存しておく？

$$d_i = \min_{r,\eta} |u_{i,r_\eta}|.$$

$r^*(\mathbf{x}_i)$ に保存しておいて，5.4 で使う

Rigid particle seeding:
最小粒子距離 ＜ 格子間隔（grid spacing）Δx

Grid color field:
それぞれの格子の色変数（color）は，(1) 格子の面への affinity(closeness) Air と，(2)その辺（side）がある側をラベリングしたタグ Tir を含んでいる．

$$A_{ir} = \begin{cases} 1, & \exists\eta \text{ with valid } u_{i,r_\eta}, \\ 0, & \text{otherwise.} \end{cases} \qquad (20)$$

有効な符号付距離 $u_{i,r\eta}$ となる η が一つでも存在するなら 1， なければ 0． i は格子ノード，r_η は剛体粒子．

Tir= $\text{sign}(u_{i,r_{\eta^*}(\mathbf{x}_i)})$.

5.3 particle-wise colored distance field の構築
grid CDF (di, Air andTir)
MPM particle locations **x**p
図は，赤い丸い剛体が，薄い青と緑のナイフみたいな剛体で三つに切られた状況？？？
(a) 赤，緑，青の三つの交差する剛体の薄い境界
(b) 格子の符号なし距離場：おそらく赤が距離 0， 黄色が近い距離，白が遠い（力学的影響を考慮しない）距離
(c) 格子 colors：三種類の格子に色を付けた．剛体の内側を薄く，外側を濃く
(d) 粒子にも色：おそらく，剛体内側を薄く，外側を濃く
(e) 粒子距離：粒子の境界までの符号なし距離
(f) MLS によって最高リクされた法線：

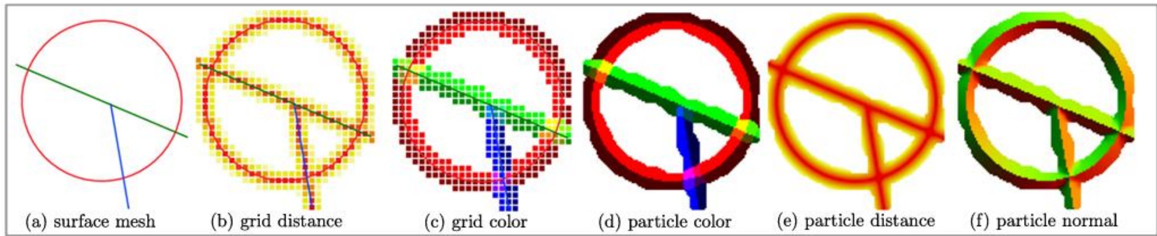


Fig. 12. (a) Three intersecting thin rigid boundaries; (b) Grid unsigned distance field; (c) Grid colors (relationship to boundaries); (d) Maintained particle color; (e)(f) Particle distances to the boundary and the normals reconstructed with MLS.

円を緑と青のラインが切っているところ．

Particle color field
粒子タグ Tpr：距離重み平均

$$T_{pr} = \mathrm{sign}\left(\sum_i N_i(\mathbf{x}_p) d_i T_{ir}\right), \tag{21}$$

Particle distance and normal
粒子法線 np：

$$\mathbf{n}_p = \nabla d_p / |\nabla d_p|.$$

dp は粒子距離

color_with_alpha　:GUI 用．f32 の 4 次元．（R=1.0, G=1.0,B=1.0,transparence=1.0）

Jp　　　　　　　　：f32, plasticity:　float の可塑性指標．雪と砂用．横滑りしていく率かな....
indice = ti.ijk　　　：3 次元（i, j, k）をまとめて表示，Taichi 公式 SNode 軸指定
offset = (-2048, -2048, -2048)：=2^11 負のインデックスを使うための offset．i-2048, j-2048, k-2048.
kernel のアクセス速度を上げるため．index の範囲を-2048 から 2048 にしている．

num_grids=1
grid_block_size = 128
leaf_block_size = 4

grid = []　　　　　：格子 ti.root => pointer [grid_size//grid_block_size=32, 32, 32] 定数？？？
grid_v = []　　　　：(f32, f32, f32)格子速度，途中運動量 mv になっている瞬間がある．
grid_m = []　　　　：f32，格子質量．
pid = []　　　　　 ：int32，粒子 id

block　　　　　　：ti.root => pointer [grid_block_size // leaf_block_size =32, 32, 32] => pointer [1024, 1024, 1024]
block.dense
　　taichi Point SNode の reference はこの辺．
　　https://docs.taichi-lang.org/docs/sparse#pointer-snode

padding = 3　　　　：壁近傍処理に使っている．壁の近傍の 2 番目の格子速度の初期値を 0
E = 1e7 * size * E_scale　　：スケーリングされたヤング率　相当柔らかい．大きくすると dt を小さくする必要がある．E_scale=1．default=1e6．この後物性ごとに変更できるようにした．
nu = 0.2　　　　　　　　：ν，ポアソン比．0.3 じゃない....
mu_0 = E/(2*(1+nu))　　　：μ=G 横弾性係数と同じ．ラメ定数を使っている．水は 0 にしている
lambda_0 = E * nu / ((1 + nu) * (1 - 2 * nu))：$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}$：さらにこの後これらは，材料によってそれぞれ変更されている．

alpha = math.sqrt(2 / 3) * 2 * sin_phi / (3 - sin_phi)　　# 砂の安息角のパラメータ？

particle：Taichi のメモリレイアウト構造

```
particle (dynamic SNode)
 ├─ x
 ├─ v
 ├─ F
 ├─ material
 └─ color
```

|-C : matrix

|-Jp

|-color_with_alpha

Taichi のクラス外（Python 側）からは，x（mpm.x[i]：mpm は MPM_solver 型変数）や v としてアクセスできる．

voxelizer = Voxelizer(　　　：ボクセルの初期化

res=self.res = (resolution, resolution, resolution),

dx=self.dx=size/resolution,

padding=self.padding=3,

super_sample=voxelizer_super_sample=2)

Voxelizer クラス．　　　　　　　　　# これ，使ってないのでは？　全力メモリの無駄

res: (resoX, resoY, resoZ) 倍になっている．

dx：半分になっている

grid_postprocess

. collide

. grid_v

(b)　　　　　Member functions:

. **add_bounding_box**(False)：

grid_postprocess.append(lambda t, dt, grid_v: self.grid_bounding_box(t, dt, unbounded, grid_v))

lambda は無名関数

lambda t, dt, v: B(t, dt, False, v)という関数オブジェクトが追加される．B()は呼び出されていない．以下の時に呼び出される．

for f in A:

f(1.0, 0.01, v)

なぜ lambda を使うか

い（ここでは False）

コールバック関数として登録したい

後でまとめて実行したい

A.append(B(...))：　今すぐ B を実行し結果を追加

A.append(lambda ...: B(...))：　後で呼ぶための関数を追加

で，lambda は予約語なので，変数には使えない．このトラップにも引っ掛かった．

. **grid_bounding_box**(self, t: ti.f32, dt: ti.f32, unbounded: ti.template(), grid_v: ti.template()):

padding=3 より壁に近い格子の速度を 0 にする

格子が壁近傍であるかの判定は，0 から始まる自然数の格子座標を[Rx, Ry, Rz]として

Ri < padding=3, Resolution – padding < Ri

で判定している．

. **add_cube**(

```
lower_corner,          # 左下の開始点座標
cube_size,             # Sx, Sy, Sz 方向サイズ
material,              # 材料
color=0xFFFFFF,        # 色じゃない．CDF？？？　実際に色として使っているのは，color_with_alpha
sample_density=None,
velocity=None):        # 初速度
```

直方体の材料を生成する

sample_density=2^3=8

vol = Sx*Sy*Sz

num_new_particles = int(8 * vol / dx^3 + 1) # 格子数の 8 倍になっている．

seed（num_new_particles, material, color）

. **seed**(new_particles: ti.i32, new_material: ti.i32, color: ti.i32):

各粒子に位置，速度，材料定数を設定

```
for i in range(n_particles[None], n_particles[None] + new_particles):
    material[i] = new_material
    x[k] = source_bound[0][k] + ti.random() * self.source_bound[1][k]　ランダムに位置を設定
```

seed_particle で速度 source_velocity も代入．

砂なら Jp[i] = 0　それ以外は Jp[i]=1

step が実際に python 側から呼ばれる計算スキーム

step(frame_dt, print_stat=False, smry_writer=None):

```
begin_substep = total_substeps
substeps = int(frame_dt / default_dt) + 1          ：2e-2 * dx / size
dt = frame_dt / substeps
frame_time_left = frame_dt

while frame_time_left > 0:
    total_substeps += 1
    frame_time_left -= dt

    grid.deactivate_all()      ：grid のメモリを解放
```

```
root
└─ pointer / bitmasked  ← ブロック単位で有効化
    └─ dense
        └─ field (grid_v など)
```

のような階層になっています。

`deactivate_all()` を呼ぶと：

- pointer / bitmasked ノード配下の **全ブロックを無効化**
- それにより
  - そのブロックは「未使用」扱いになる
  - 次のステップで必要になれば再び自動的に activate される

sparse マトリクスを使っているときだけ意味をもつ．taichi 内でメモリを開放することに近い．OS には返さない．

build_pid(pid, grid_m, 0.5)　　：粒子位置を格子座標に変換
p2g(dt)　　：粒子情報を使って格子情報（grid_v, grid_m）を更新
grid_normalization_and_gravity(dt, self.grid_v, self.grid_m)
for p in self.grid_postprocess:　：壁の衝突処理．壁近傍 3 グリッドの速度を 0 に
　p(self.t, dt, self.grid_v)：実際は add_bounding_box を呼び出し
self.t += dt　　：時刻更新
self.**g2pV**(dt)　　：格子情報で粒子情報（v[p], C[p]）を更新
motion.carangiform(t, self.x, self.v, self.Eelastic) #　これが carangiform
self.**g2pX**(dt)　　：格子情報で粒子情報（x[p]）を更新
cur_frame_velocity = self.compute_max_velocity()　：最大粒子速度成分計算
self.all_time_max_velocity = max(self.all_time_max_velocity, cur_frame_velocity)

build_pid(
　pid: ti.template()
　grid_m: ti.template()
　offset: ti.template()):
　for p in self.x:
　　base = int(ti.floor(x[p] * inv_dx - 0.5)) - ti.Vector(offset)　　：粒子位置を格子座標に変換
　　# Pid grandparent is `block`
　　base_pid = ti.rescale_index(grid_m, pid.parent(2), base)：base というインデックスを、pid.parent(2) の SNode 階層から grid_m の SNode 階層にスケール変換（座標変換）
　　ti.append(pid.parent(), base_pid, p)：pid.parent() が指す dynamic SNode に対して、base_pid 位置のリストへ値 p を追加

carangiform を入れるために，分けた．symplectic Euler を時間積分に使っているために，
v(n+1) = v(n) + a(n) * dt
x(n+1) = x(n) + v(n+1) * dt
になっている．float の 7 桁精度なので，dt がちいさいスキームは使えない．
なお，格子の速度は，運動量 mv から計算している．

p2g(dt: ti.f32):　　# ******** transfer particles to grids
　ti.no_activate(particle)　　：sparse SNode を自動 activate しないようにする指定
　for I in ti.grouped(self.pid):　：粒子 id ループ
　　p = self.pid[I]
　　base = ti.floor(x[p] * inv_dx - 0.5).cast(int)　：p 番目の粒子がある格子座標（＝格子整数番号）

　　Im = ti.rescale_index(pid, grid_m, I)　：I というインデックスを、grid_m の SNode 階層から pid の SNode 階層にスケール変換（座標変換）

　　fx = x[p] * inv_dx - base.cast(float)　　：x[p]/dx – base で fx は自身が存在する grid 内での相対座標．グリット内位置．
　　# Quadratic kernels　[http://mpm.graphics　Eqn. 123, with x=fx, fx-1,fx-2]

　　w = [0.5 * (1.5 - fx)**2, 0.75 - (fx - 1)**2, 0.5 * (fx - 0.5)**2]
　　　：

$$\begin{bmatrix} w_{xx} & w_{xy} & w_{xz} \\ w_{yx} & w_{yy} & w_{yz} \\ w_{zx} & w_{zy} & w_{zz} \end{bmatrix} = [\frac{1}{2}(1.5-f_x)^2,\ \frac{3}{4}-(f_x-1)^2, \left(f_x-\frac{1}{2}\right)^2]$$

一様2次B-スプライン曲線 [編集]

一様なノットにおける2次B-スプライン曲線において、B-スプライン基底関数は次のようになる。

$$b_{j,2}(t) = \begin{cases} \frac{1}{2}t^2 \\ -t^2 + t + \frac{1}{2} \\ \frac{1}{2}(1-t)^2 \end{cases}$$

これを行列形式にすると、

$$\mathbf{S}_i(t) = \begin{bmatrix} t^2 & t & 1 \end{bmatrix} \frac{1}{2} \begin{bmatrix} 1 & -2 & 1 \\ -2 & 2 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{p}_{i-1} \\ \mathbf{p}_i \\ \mathbf{p}_{i+1} \end{bmatrix} \text{for } t \in [0,1], i = 1, 2 \ldots m-2$$

`w` は粒子と格子ノード間の**補間（形状）関数の1次元成分（2次Bスプライン）**を各次元ごとに計算したもので、それらを掛け合わせて粒子→格子／格子→粒子の重み（スカラー）を得ています。
信頼度：High — コードは標準的な二次カーネル（quadratic B-spline）による3×3（各軸3点）補間を実装しています。

短い理由： `fx` が粒子のセル内の小数位置で， `w` の3成分はその位置に対するノード毎のカーネル値（各次元）を与え，ループで次元積を取ることで多次元の重みを得ているからです。

簡潔な解説（ステップ順）

1. `base` と `fx` の意味
   - `base = floor(x * inv_dx - 0.5)` ：粒子 p が影響を与える左下（あるいは最小インデックス）ノードのインデックス。
   - `fx = x * inv_dx - base` ：そのセル内での**相対位置**（各成分が 0..1 の値） ＝粒子の格子ローカル座標（fractional offset）。
2. カーネル（1D）の3点評価（コード）

コード

```
w0 = 0.5 * (1.5 - fx)**2
w1 = 0.75 - (fx - 1.0)**2
w2 = 0.5 * (fx - 0.5)**2
```

   - これは二次Bスプライン（quadratic B-spline）を3点で評価したもので、各 `wk` は **各次元ごとの**ノード重み（ベクトル）を返します。
   - 具体的にはノードインデックス `i = base + k` （k=0,1,2）に対する 1D カーネル値 $N(x_f - (k-1))$ に相当します。

3. 多次元化（積分性）

```
weight = Π_{d=0..dim-1} w[offset[d]][d]
```

- 各軸で独立にカーネルを評価し、その積をとる（tensor-product カーネル）。
- これにより、例えば2Dでは 3×3 = 9 個の格子ノードが粒子の補間に使われます（3Dなら 27 個）。

4. `new_v` の計算（補間）
   - `new_v += weight * g_v`：格子ノード速度 `g_v` を重みで総和し、粒子の速度（あるいは補間値）を得ます。これは標準的な P2G/G2P 補間の逆も同様です。

5. `C` の計算（APIC 的な affine 成分）
   - `dpos = offset.cast(float) - fx` は「粒子位置からノード位置へのベクトル（grid 単位）」です。
   - `C += 4 * self.inv_dx * weight * g_v.outer_product(dpos)` は APIC で使う**粒子に保存するアフィン速度勾配**の寄与を累積しています。
     - なぜ `4 * inv_dx`？：二次カーネルの勾配係数とセル長スケールを合わせた定数で、APIC の理論式に現れる（カーネル勾配のスケーリング）。実装上は「格子速度と位置差の外積」を適切なスケールに乗じて速度勾配の近似を得るための定数です。
     - 結果 `C` は（粒子に保存する）affine velocity matrix の近似で、後に P2G 時に `affine @ dpos` の形で運動量に寄与します。

6. 境界や範囲チェック
   - `assume_in_range` / `rescale_index` 周りは GPU ブロック最適化のヒントや pointer タイルの扱いで、補間の論理には直接影響しないが、範囲外アクセスを防ぎつつ局所ブロックで効率よく計算しています。

簡単な数値例（2D）

- もし `fx = [0.3, 0.7]` のとき（粒子はセルの左側にやや寄り，垂直方向は上寄り）：
  - `w0 = 0.5*(1.5 - fx)^2` → per-dim vectors
  - 積を取れば、あるノード（offset = (0,0)）への重みは `w0[0]*w0[1]`、（offset=(1,2)）なら `w1[0]*w2[1]` のようになります。
  - 各ノード重みに基づいて `new_v` と `C` を集めます。

実装的ポイント（短く）

- この 3点二次カーネルは滑らかで保存性と安定性のバランスが良く、MPM 実装で広く使われます。
- `weight` の合計は（理想的には）1 に近いはずで、補間の一貫性を保ちます（浮動小数点・範囲処理で微小なずれあり）。
- `C` を強めると APIC の平滑化と数値粘性が増すため、先に話した `apic_alpha` / `affine_scale` で調整するのが有効です。

結局，粒子速度が，近傍 27 格子から平滑化されて求められているので，格子が粗いと圧倒的になまる．ただし，このなまりが，流体粘性としてとらえられている．粘性係数の概念はない．NS 方程式も解いていない．おそらく渦は発生しない．発生しても力学とは関係ない．

```
# Deformation gradient update   変形勾配行列 F
F = self.F[p]                    ：p は粒子番号
if self.material[p] == self.material_water:    # 水のルーチン
    F = ti.Matrix.identity(ti.f32, self.dim)
```

if ti.static(self.support_plasticity):　　　　　　# default true
　　F[0, 0] = self.Jp[p]

$$F = \begin{bmatrix} Jp[p] & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

F = (ti.Matrix.identity(ti.f32, self.dim) + dt * C[p]) @ F

$$F = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + dt * [C(p)] \begin{bmatrix} Jp[p] & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$ C matrix が不明

# Hardening coefficient: snow gets harder when compressed
h = 1.0
if ti.static(self.support_plasticity): # default true
　　if self.material[p] != self.material_water:　　　　　　：水以外なら
　　　　h = ti.exp(10 * (1.0 - self.Jp[p]))　　　　　：h = e^10*(1-Jp[p])
　　　　Jp[p]はほぼ 0 で 0.02 あたりになったりする．
　　　　e^10(1-0.02)=e^9.8=2*10^4 ぐらい硬くなる．
if self.material[p] == self.material_elastic:　# jelly, make it softer 弾性体なら
　　h = 0.3
mu = self.mu_0 * h　　　　　　：水以外は横弾性係数を強く．
la = self.lambda_0 * h　　　　　：ラメ定数も強く．弾性体はこちらは弱く．
if self.material[p] == self.material_water:　# liquid
　　mu = 0.0　　　　　：水は横弾性係数を 0
U, sig, V = ti.svd(F)　　　　　　：特異値分解 $F=U\Sigma V^T$

```
U, sig, V = ti.svd(F)
```

と書くと,

- `U` : Matrix
- `sig` : Vector（特異値が並ぶ）
- `V` : Matrix

変形勾配 F を，回転成分と伸び（スケール）成分に分解する．

J = 1.0
if self.material[p] != self.material_sand:　　　　　：砂でないなら
　　for d in ti.static(range(self.dim)):
　　　　new_sig = sig[d, d]　　　　　　　　：Σ の対角成分＝平行移動成分
　　　　if self.material[p] == self.material_snow:　# Snow なら
　　　　　　new_sig = min(max(sig[d, d], 1 - 2.5e-2), 1 + 4.5e-3)　# $1-2.5^{-2} < sig < 1+4.5^{-3}$
　　　　if ti.static(self.support_plasticity): # default is true
　　　　　　self.Jp[p] *= sig[d, d] / new_sig　　　：　Jp[p] = sig[d,d] / new_sig
　　　　sig[d, d] = new_sig
　　　　J *= new_sig　　　　　　　　　　：
if self.material[p] == self.material_water:　：水なら
　　# Reset deformation gradient to avoid numerical instability
　　F = ti.Matrix.identity(ti.f32, self.dim)
　　F[0, 0] = J

$$F = \begin{bmatrix} Jp[p] & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

　　if ti.static(self.support_plasticity):
　　　　self.Jp[p] = J　　　　　　　　　：よこしべりしていく率？？？
elif self.material[p] == self.material_snow:　：雪なら
　　# Reconstruct elastic deformation gradient after plasticity
　　F = U @ sig @ V.transpose()　　　　：特異値分解を戻す

stress = ti.Matrix.zero(ti.f32, self.dim, self.dim)　　　：応力テンソル

if self.material[p] != self.material_sand:　　　　　：砂じゃないなら

stress = 2 * mu * (F - U @ V.transpose()) @ F.transpose() + ti.Matrix.identity(ti.f32, self.dim) * la * J * (J - 1)

$$\begin{bmatrix}\sigma_{xx} & \tau_{xy} & \tau_{xz}\\ \tau_{yx} & \sigma_{yy} & \tau_{yz}\\ \tau_{zx} & \tau_{zy} & \sigma_{zz}\end{bmatrix} = 2\mu(F - UV^T)F^T + \begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}\frac{E\nu}{(1+\nu)(1-2\nu)}J(J-1)$$

else: ：砂なら

if ti.static(self.support_plasticity):

sig = self.sand_projection(sig, p) ：平行移動量を修正

F = U @ sig @ V.transpose() ：戻す，F=UΣV^T

log_sig_sum = 0.0

center = ti.Matrix.zero(ti.f32, self.dim, self.dim)

for i in ti.static(range(self.dim)):

log_sig_sum += ti.log(sig[i, i])

center[i, i] = 2.0 * self.mu_0 * ti.log(sig[i, i]) * (1 / sig[i, i])

for i in ti.static(range(self.dim)):

center[i,i] += self.lambda_0 * log_sig_sum * (1 / sig[i, i])

stress = U @ center @ V.transpose() @ F.transpose()

$$\begin{bmatrix}\sigma_{xx} & \tau_{xy} & \tau_{xz}\\ \tau_{yx} & \sigma_{yy} & \tau_{yz}\\ \tau_{zx} & \tau_{zy} & \sigma_{zz}\end{bmatrix} = U\begin{bmatrix}a & 0 & 0\\ 0 & b & 0\\ 0 & 0 & c\end{bmatrix}V^T F^T$$

self.F[p] = F ：回転と平行移動の変形勾配行列

stress = (-dt * self.p_vol * 4 * self.inv_dx**2) * stress

$$= -\frac{4dt*V}{dx^2}\begin{bmatrix}\sigma_{xx} & \tau_{xy} & \tau_{xz}\\ \tau_{yx} & \sigma_{yy} & \tau_{yz}\\ \tau_{zx} & \tau_{zy} & \sigma_{zz}\end{bmatrix}$$ [m^{3}s/m^2] * [Pa]=[Ns/m]????

TODO: implement g2p2g pmass

mass = self.p_mass : particle mass?

ここで，それぞれの材料の質量は定義できる．

if self.material[p] == self.material_water: ：水なら

mass *= self.water_density ：質量×密度？ だとしたら density は比重？

affine = stress + mass * self.C[p]

$$affine = -\frac{4dt*V}{dx^2}\begin{bmatrix}\sigma_{xx} & \tau_{xy} & \tau_{xz}\\ \tau_{yx} & \sigma_{yy} & \tau_{yz}\\ \tau_{zx} & \tau_{zy} & \sigma_{zz}\end{bmatrix} + m[C(p)]$$

$$affine = -\frac{4dt*V}{dx^2}\begin{bmatrix}\sigma_{xx} & \tau_{xy} & \tau_{xz}\\ \tau_{yx} & \sigma_{yy} & \tau_{yz}\\ \tau_{zx} & \tau_{zy} & \sigma_{zz}\end{bmatrix} + m\begin{bmatrix}\frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} & \frac{\partial u}{\partial z}\\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} & \frac{\partial v}{\partial z}\\ \frac{\partial w}{\partial x} & \frac{\partial w}{\partial y} & \frac{\partial w}{\partial z}\end{bmatrix}$$

```
# Loop over 3x3 grid node neighborhood
for offset in ti.static(ti.grouped(self.stencil_range())):  # (3, 3, 3)
    dpos = (offset.cast(float) - fx) * self.dx
    weight = 1.0
    for d in ti.static(range(self.dim)):
        weight *= w[offset[d]][d]
    self.grid_v[base + offset] += weight * (mass * v[p] + affine @ dpos)
    self.grid_m[base + offset] += weight * mass
```

self.grid_v[base + offset] += weight * (mass * v[p] + affine @ dpos)：ここでは運動量

grid_normalization_and_gravity(格子運動量から速度に変換

```
        dt: ti.f32,
        grid_v: ti.template(),
        grid_m: ti.template()):
    v_allowed = self.dx * self.g2p2g_allowed_cfl / dt
```

v_allowed = self.dx * self.g2p2g_allowed_cfl / dt ： dx * クーラン数=0.9 /dt
クーラン数 0.9 で計算できる速度限界

```
    for I in ti.grouped(grid_m):
```

if grid_m[I] > 0:　# No need for epsilon here　格子に質量（粒子）があったら
　　grid_v[I] = (1 / grid_m[I]) * grid_v[I]　# Momentum to velocity
　　grid_v[I] += dt * self.gravity[None]

 # Grid velocity clamping：クーラン数から求めた速度制限より早くならないようにする速度制限．これだと，dt を粗くすればするほど速度が制限され，ゆっくりになる．式を見ると，格子を半分にしたら，dt も半分にしなければならない．

if ti.static(self.g2p2g_allowed_cfl > 0 and self.use_g2p2g and self.v_clamp_g2p2g):
　　grid_v[I] = min(max(grid_v[I], -v_allowed), v_allowed)：　$-v_{max} < v < v_{max}$

g2p(self, dt: ti.f32):
　　ti.no_activate(particle)　　　　　　：particle の非アクティベイト
　　for I in ti.grouped(pid):　　　　：粒子 id
　　　　p = pid[I]　　　　　　　　　　：
　　　　base = ti.floor(x[p] * inv_dx - 0.5).cast(int)　：格子座標計算　x[p]/dx -0.5
　　　　Im = ti.rescale_index(pid, grid_m, I)　　　：I というインデックスを grid の SNode 階層から pid の SNode 階層にスケール変換（座標変換）

　　　　for D in ti.static(range(self.dim)):　　　　：　speed up for GPU
　　　　　　base[D] = ti.assume_in_range(base[D], Im[D], 0, 1)
　　　　fx = self.x[p] * self.inv_dx - base.cast(float)　：　fx = x[p]/dx – base，格子座標
　　　　w = [0.5 * (1.5 - fx)**2, 0.75 - (fx - 1.0)**2, 0.5 * (fx - 0.5)**2]
　　　　　：重みベクトル　$[0.5 * (1.5 - \mathrm{fx})^2, 0.75 - (\mathrm{fx} - 1)^2, 0.5 * (\mathrm{fx} - 0.5)^2]$，fx はベクトル
　　　　new_v = ti.Vector.zero(ti.f32, self.dim)
　　　　new_C = ti.Matrix.zero(ti.f32, self.dim, self.dim)
　　　　# Loop over 3x3 grid node neighborhood
　　　　for offset in ti.static(ti.grouped(self.stencil_range())):　：# (3, 3, 3)
　　　　　　dpos = offset.cast(float) – fx　　　　：中心から粒子までの位置ベクトル
　　　　　　g_v = self.grid_v[base + offset]　　　　　　：格子速度
　　　　　　weight = 1.0
　　　　　　for d in ti.static(range(self.dim)):
　　　　　　　　weight *= w[offset[d]][d]　　　　　　：重み
　　　　　　new_v += weight * g_v　　　　　　　　　　：粒子速度
　　　　　　new_C += 4 * inv_dx * weight * g_v.outer_product(dpos)　：
　　　　　　$C += \frac{4w}{dx}(v_{grid} \times dpos)$

$$V = (u, v, w), \quad X = (x, y, z)$$

$$V \otimes X = \begin{pmatrix} u \\ v \\ w \end{pmatrix} \begin{pmatrix} x & y & z \end{pmatrix} = \begin{pmatrix} ux & uy & uz \\ vx & vy & vz \\ wx & wy & wz \end{pmatrix}$$

　　　　　　Taichi では，outer product はテンソル積（行列）のこと．外積ベクトルは（cross product）

　　　　if self.material[p] != self.material_stationary:　　　：静止物体じゃなかったら
　　　　　：速度を先に更新しているので symplectic Euler
　　　　　　v[p] = new_v　　　　　　　　　　　　：粒子速度更新
　　　　　　C[p]　　= new_C　　　　　　　　　　：C を更新
　　　　　　x[p] += dt * v[p]　# advection　　　：粒子位置更新（symplectic Euler）

. **compute_max_velocity**(self) -> ti.f32:　　　：x, y, z 成分で一番速い速度
　　max_velocity = 0.0
　　for p in self.v:
　　　　v = self.v[p]
　　　　v_max = 0.0
　　　　for i in ti.static(range(self.dim)):
　　　　　　v_max = max(v_max, abs(v[i]))
　　　　ti.atomic_max(max_velocity, v_max)

```
return max_velocity
```

(3-2)　魚のクラス（motion.py）

メインコードから受け取った魚データに基づいて粒子群から魚の弾性体粒子を探し出し，ボディを分割して粒子番号や初期位置などを保存している．運動制御もここ．諸事情で Carangi ではなく，Anguilli で運動させている．

```
import taichi as ti
import math
import random

PI = ti.math.pi        #  これが GPU 内でつかえる定数

@ti.data_oriented
class Motionclass:
    def __init__(self, fish, mpm: ti.template(), dt: ti.f32, offset: ti.f32):
        self.translation = fish["translation"]        #  渡された fish リストから translation を代入．これもリスト
        self.start_pos = fish["start_pos"]
        self.scale = fish["scale"]
        self.relax =    fish["relax"]                 # relaxation time
        self.freq =    fish["frequency"]                        # [Hz]
        self.length =    fish["length"]               # [m]
        self.lambdaF =    fish["lambda"]                        # [m]
        self.amp =    fish["amplitude"]                         # [m]       lambda is reserved name by python
        self.N =    fish["N"]                         # body division number
        self.dt = dt                                            # [s]
        self.offset = offset                          # [m] particle radius
        self.flg = True                                         # calculation flg

        self.fish_particle_num = 0;                   # count elastic material number
        for i in range(mpm.x.shape[0]):
            if mpm.material[i] == mpm.material_elastic:
                self.fish_particle_num += 1;

        # mapping global particle number to fish particle number
        self.fishX0 = ti.Vector.field(3, ti.f32, shape=self.fish_particle_num)
        self.particleID = ti.field(dtype=ti.i32, shape=self.fish_particle_num)

        j = 0
        for i in range(mpm.x.shape[0]):
            if mpm.material[i] == mpm.material_elastic:
                self.fishX0[j] = mpm.x[i]
                self.particleID[j] = i
                j += 1

        # **** mapping fish particel number to Divided particle nubmer ****
        dL = self.length / self.N                                       # N is divided body number

        # calculate the particle number and the center position
        dividedPR_lenTemp = ti.field(ti.i32, shape=self.N)            # total particle number for divided right body
        dividedPL_lenTemp = ti.field(ti.i32, shape=self.N)            # for left
        dividedPYRTemp = ti.field(ti.f32, shape=self.N)               # y coodinate of average position of particlers for divided right body
        dividedPYLTemp = ti.field(ti.f32, shape=self.N)               # for left
        dividedPZUR_lenTemp = 0
        dividedPZLR_lenTemp = 0
```

```
        dividedPZURTemp = 0.0                                   # z coodinate of average position
of upper particlers
        dividedPZLRTemp = 0.0                                   # z coodinate of average position
of lower particlers
        for i in range(self.N):
            dividedPR_lenTemp[i] = 0
            dividedPL_lenTemp[i] = 0
            dividedPYRTemp[i] = 0.0
            dividedPYLTemp[i] = 0.0
        for i in range(self.fish_particle_num):                 # p: 0..fish_particle_num-1
            fishx = mpm.x[self.particleID[i]] - self.translation
            j = (int)((fishx[0] + 0.5 * self.length) / dL)         # ( fishx + length / 2.0 ) / dL
            if 0 <= j and j < self.N:
                if 0.0 < fishx[1]:      # if right
                    dividedPR_lenTemp[j] += 1
                    dividedPYRTemp[j] += fishx[1]
                if fishx[1] < 0.0:      # if left
                    dividedPL_lenTemp[j] += 1
                    dividedPYLTemp[j] += fishx[1]
                if 0.0 < fishx[2]:      # if upper
                    dividedPZURTemp += fishx[2]
                    dividedPZUR_lenTemp += 1
                if fishx[2] < 0.0:      # if lower
                    dividedPZLRTemp += fishx[2]
                    dividedPZLR_lenTemp += 1

        for i in range(self.N):
            dividedPYRTemp[i] = 0.5 * dividedPYRTemp[i] / dividedPR_lenTemp[i]
            dividedPYLTemp[i] = 0.5 * dividedPYLTemp[i] / dividedPL_lenTemp[i]
        dividedPZURTemp = 1.5 * dividedPZURTemp / dividedPZUR_lenTemp
        dividedPZLRTemp = 1.5 * dividedPZLRTemp / dividedPZLR_lenTemp

        #
        self.dividedParticleR_len = ti.field(ti.i32, shape=self.N)
        self.dividedParticleL_len = ti.field(ti.i32, shape=self.N)
        for i in range(self.N):
            self.dividedParticleR_len[i] = 0
            self.dividedParticleL_len[i] = 0

        for i in range(self.fish_particle_num):                 # p: 0..fish_particle_num-1
            fishx = mpm.x[self.particleID[i]] - self.translation
            j = (int)((fishx[0] + 0.5 * self.length) / dL)         # ( fishx + length / 2.0 ) / dL
            if 0 <= j and j < self.N:
                if dividedPYRTemp[j] < fishx[1] and fishx[2] < dividedPZURTemp and dividedPZLRTemp <
fishx[2]:  # if far right
                    self.dividedParticleR_len[j] += 1
                if fishx[1] < dividedPYLTemp[j] and fishx[2] < dividedPZURTemp and dividedPZLRTemp <
fishx[2]:                   # if far left
                    self.dividedParticleL_len[j] += 1

        max_nR = self.dividedParticleR_len[0]
        max_nL = self.dividedParticleL_len[0]
        for j in range(1, self.N):
            if max_nR < self.dividedParticleR_len[j]:    max_nR = self.dividedParticleR_len[j]
            if max_nL < self.dividedParticleL_len[j]:    max_nL = self.dividedParticleL_len[j]

        if max_nR < 1 or max_nL < 1: print("ERROR: motion class")

        #
        rowR = ti.field(ti.i32, shape=self.N)
```

```
        rowL = ti.field(ti.i32, shape=self.N)
        for i in range(self.N):
            rowR[i] = 0
            rowL[i] = 0
        self.dividedParticleR = ti.field(dtype=ti.i32, shape=(self.N, max_nR))# [devided body id, body particle
id ] = global particle id
        self.dividedParticleL = ti.field(dtype=ti.i32, shape=(self.N, max_nL))
        self.dividedParticleCR= ti.field(dtype=ti.i32, shape=(self.N, max_nR))          # [devided body id,
body particle id ] = connected adjecent body particle id
        self.dividedParticleCL= ti.field(dtype=ti.i32, shape=(self.N, max_nL))          #
        for i in range(self.fish_particle_num):                                     # p: 0..fish_particle_num-1
            fishx = mpm.x[self.particleID[i]] - self.translation
            j = (int)((fishx[0] + 0.5 * self.length) / dL)              # ( fishx + length / 2.0 ) / dL
            if 0 <= j and j < self.N:
                if dividedPYRTemp[j] < fishx[1] and fishx[2] < dividedPZURTemp and dividedPZLRTemp <
fishx[2]:                                  # if far right
                    self.dividedParticleR[j,rowR[j]] = self.particleID[i]
                    if j < self.N -2:
                        self.dividedParticleCR[j,rowR[j]] =
int(random.randrange(self.dividedParticleR_len[j+2])) # ti.random(dtype=ti.i32) % self.dividedParticleR_len[j+2]
                    rowR[j] += 1
                if fishx[1] < dividedPYLTemp[j] and fishx[2] < dividedPZURTemp and dividedPZLRTemp <
fishx[2]:                                  # if far left
                    self.dividedParticleL[j,rowL[j]] = self.particleID[i]
                    if j < self.N -2:
                        self.dividedParticleCL[j,rowL[j]] =
int(random.randrange(self.dividedParticleL_len[j+2])) # ti.random(dtype=ti.i32) % self.dividedParticleL_len[j+2]
                    rowL[j] += 1

        # initial distance between particles
        self.dividedParticleCRL= ti.field(dtype=ti.f32, shape=(self.N, max_nR))        # [devided body id,
body particle id ] = length to the connected adjecent body particle
        self.dividedParticleCLL= ti.field(dtype=ti.f32, shape=(self.N, max_nL)) #
        for i in range(self.N-2):
            for j in range(self.dividedParticleR_len[i]):
                x0 = mpm.x[ self.dividedParticleR[i,j] ]
                x1 = mpm.x[ self.dividedParticleR[i+2, self.dividedParticleCR[i,j] ] ]
                dis = x0 - x1
                self.dividedParticleCRL[i,j] = dis.norm()

            for j in range(self.dividedParticleL_len[i]):
                x0 = mpm.x[ self.dividedParticleL[i,j] ]
                x1 = mpm.x[ self.dividedParticleL[i+2, self.dividedParticleCL[i,j] ] ]
                dis = x0 - x1
                self.dividedParticleCLL[i,j] = dis.norm()

    @ti.kernel      # kernel は python から呼び出せるが, func は kernel からしか呼び出せない.
    def carangiform( self, t: ti.f32, px: ti.template(), pv: ti.template(), YoungModulus: ti.f32, dx: ti.f32   ):
        """
        # carangi
        E = YoungModulus                    # Young's modulus
        I = 2.0e-8                          # 2.0*10^4 mm^4 = 2.0 * 10^-8 m^4
        S = 0.064                                   # 670 mm^2 = 0.064 m^2
        w = 0.04                            # body width
        k = 20.0 * 0.5 / (E*I) * S / (w/4.0)    # F = d2y_dx2 / (EI) * S

        for i in range(self.N-2):
            x = ( i + 0.5 ) * self.length / self.N
            theta = 2.0 * PI * ( x/self.length - self.freq * t )
```

```
            d2y_dx2 = 4.0 * PI * self.amp / self.length * ti.math.cos(theta) - 4.0 * PI**2.0 * self.amp * x /
( self.lambdaF**2.0 * self.length) * ti.math.sin(theta)
            stress = k * d2y_dx2 / self.dividedParticleR_len[i] * min( t/self.relax, 1.0 )
            #print(stress)

            for j in range(self.dividedParticleR_len[i]):
                # right side
                id0 = self.dividedParticleR[i,j]
                id1 = self.dividedParticleR[i+1, self.dividedParticleCR[i,j]]
                p0 = px[ id0 ]
                p1 = px[ id1 ]
                n = p0 - p1
                f_m = stress * n.normalized() * self.dt
                pv[ id0 ] += f_m
                pv[ id1 ] -= f_m

            for j in range(self.dividedParticleL_len[i]):
               # left side
                id0 = self.dividedParticleL[i,j]
                id1 = self.dividedParticleL[i+1, self.dividedParticleCL[i,j]]
                p0 = px[ id0 ]
                p1 = px[ id1 ]
                n = p0 - p1
                f_m = stress * n.normalized() * self.dt
                pv[ id0 ] -= f_m
                pv[ id1 ] += f_m
        """

        # anguilli
        E = YoungModulus                        # Young's modulus
        I = 2.0e-8                              # 2.0*10^4 mm^4 = 2.0 * 10^-8 m^4
        S = 0.064                                       # 670 mm^2 = 0.064 m^2
        w = 0.04                                # body width
        k = 5.0 * 0.5 / (E*I) * S / (w/4.0)     # F = d2y_dx2 / (EI) * S
        limitU = dx / self.dt                   # limited velocity by Courant condition

        for i in range(self.N-3):
            x = ( i + 1.5 ) * self.length / self.N
            theta = 2.0 * PI * ( x/self.length - self.freq * t )
            d2y_dx2 = - 4.0 * PI**2.0 * self.amp / ( self.lambdaF**2.0 * self.length) * ti.math.sin(theta)
            accel = k * d2y_dx2 / self.dividedParticleR_len[i] * min( t/self.relax, 1.0 )                #
F/m
            #print(accel)

            #if limitU < ti.abs(accel)*self.dt:
            # print("Courant condition error ", limitU, " ", ti.abs(accel)*self.dt )

            accel = min( max( -limitU/self.dt, accel ), limitU/self.dt )

            for j in range(self.dividedParticleR_len[i]):
                # right side
                id0 = self.dividedParticleR[i,j]
                id1 = self.dividedParticleR[i+2, self.dividedParticleCR[i,j]]
                p0 = px[ id0 ]
                p1 = px[ id1 ]
                n = p0 - p1
                vel = ti.Vector.zero(ti.f32, 3)
                dis = n.norm()
                init_length = self.dividedParticleCRL[i,j]
                if 0.95 * init_length < dis and dis < 1.05 * init_length:
```

```
                vel = accel * n.normalized() * self.dt
            pv[ id0 ] += vel
            pv[ id1 ] -= vel

        for j in range(self.dividedParticleL_len[i]):
            # left side
            id0 = self.dividedParticleL[i,j]
            id1 = self.dividedParticleL[i+2, self.dividedParticleCL[i,j]]
            p0 = px[ id0 ]
            p1 = px[ id1 ]
            n = p0 - p1
            vel = ti.Vector.zero(ti.f32, 3)
            dis = n.norm()
            init_length = self.dividedParticleCLL[i,j]
            if 0.95 * init_length < dis and dis < 1.05 * init_length:
                vel = accel * n.normalized() * self.dt

            pv[ id0 ] -= vel
            pv[ id1 ] += vel

    @ti.kernel
    def move( self, t: ti.f32, px: ti.template(), pv: ti.template(), time:ti.f32 ):
        d = ti.Vector( self.start_pos )
        d -= self.translation
        v = d / time
        d *= t/time
        for i in range(self.fish_particle_num):
            px[self.particleID[i]] = self.fishX0[i] + d
            pv[self.particleID[i]] = v
```

(3-3)　メインコード（FSIswim00.py）

説明に力尽きた.... 後日説明

```
import taichi as ti
import numpy as np
import os
import glob

from moviepy import ImageSequenceClip

from engine.mpm_solver import MPMSolver
from VTU import VTUclass
from motion import Motionclass

# Parameters
SimulationTime = 1.0        # [s]
CalculationSpace = 0.4      # [m]
Resolution = 64
dt = 5e-5
VTUinterval = 20  # VTU saving interval
fileName = "flyingfish.obj"
fish = {
        "translation": [CalculationSpace/2.0, CalculationSpace/2.0, 0.2],
        "start_pos": [CalculationSpace/2.0, CalculationSpace/2.0, 0.1],
        "scale": [1.0, 1.0, 1.0],
        "relax": 0.1,
        "frequency": 10,            # [Hz]
        "length": 0.2,                        # body length [m]
```

```
        "lambda": 0.2,                          # lambda of carangiform [m]
        "amplitude": 0.02,                # tail amplitude of carangiform [m]
        "N": 16,                                # body division number
        }

ti.init(arch=ti.cpu)

mpm = MPMSolver(res=(Resolution, Resolution, Resolution), size=CalculationSpace, max_num_particles=2 **
23, use_ggui=True)

mpm.add_cube(lower_corner=[0, 0, 0], cube_size=[CalculationSpace, CalculationSpace, 0.1],
material=MPMSolver.material_water)                # water
mpm.add_CADmodel(fileName, offset=fish["translation"], scale=fish["scale"],
material=MPMSolver.material_elastic, size_ratio=0.5)        # fish

mpm.set_gravity((0, 0, -9.8))

@ti.kernel
def set_color(ti_color: ti.template(), material_color: ti.types.ndarray(), ti_material: ti.template()):
    for I in ti.grouped(ti_material):
        material_id = ti_material[I]
        color_4d = ti.Vector([0.0, 0.0, 0.0, 0.8])
        for d in ti.static(range(3)):
            color_4d[d] = material_color[material_id, d]
        ti_color[I] = color_4d

res = (1024, 760)
window = ti.ui.Window("Real MPM 3D", res, vsync=True)
canvas = window.get_canvas()
scene = ti.ui.Scene()
camera = ti.ui.make_camera()
camera.position(-0.5, -1, 1)
camera.lookat(0.2, 0.2, 0.2)
camera.up(0, 0, 1)
camera.fov(20)
particles_radius = CalculationSpace / Resolution / 4.0

# ground lines
N_ground = (int)(Resolution/3)
ground_vbo = ti.Vector.field(3, dtype=ti.f32, shape=4*(N_ground+1))
ground_verts = np.zeros((4*(N_ground+1), 3), dtype=np.float32)
for i in range(N_ground+1):
    x = i * CalculationSpace / N_ground
    ground_verts[4*i+0] = [x, 0, 0]
    ground_verts[4*i+1] = [x, CalculationSpace, 0]
    ground_verts[4*i+2] = [0, x, 0]
    ground_verts[4*i+3] = [CalculationSpace, x, 0]

def draw_ground(scene):
    ground_vbo.from_numpy(ground_verts)
    scene.lines(ground_vbo, width=1, color=(0.1,0.1,0.1))

def render():
    camera.track_user_inputs(window, movement_speed=0.03, hold_key=ti.ui.RMB)
    scene.set_camera(camera)
    scene.ambient_light((0.4, 0.4, 0.4))
    set_color(mpm.color_with_alpha, material_type_colors, mpm.material)
    scene.particles(mpm.x, per_vertex_color=mpm.color_with_alpha, radius=particles_radius)
    scene.point_light(pos=(-0.5, 1.5, 0.5), color=(0.7, 0.7, 0.7))
    scene.point_light(pos=(-0.5, 1.5, 1.5), color=(0.7, 0.7, 0.7))
```

```
    draw_ground(scene)      # ground z = 0 plane
    canvas.scene(scene)

def show_options():
    global particles_radius

    window.GUI.begin("Solver Property", 0.05, 0.1, 0.2, 0.10)
    window.GUI.text(f"Current particle number {mpm.n_particles[None]}")
    particles_radius = window.GUI.slider_float("particles radius ", particles_radius, 0, 0.1)
    window.GUI.end()

    window.GUI.begin("Camera", 0.05, 0.3, 0.3, 0.16)
    camera.curr_position[0] = window.GUI.slider_float("camera pos x", camera.curr_position[0], -10, 10)
    camera.curr_position[1] = window.GUI.slider_float("camera pos y", camera.curr_position[1], -10, 10)
    camera.curr_position[2] = window.GUI.slider_float("camera pos z", camera.curr_position[2], -10, 10)

    camera.curr_lookat[0] = window.GUI.slider_float("camera look at x", camera.curr_lookat[0], -10, 10)
    camera.curr_lookat[1] = window.GUI.slider_float("camera look at y", camera.curr_lookat[1], -10, 10)
    camera.curr_lookat[2] = window.GUI.slider_float("camera look at z", camera.curr_lookat[2], -10, 10)

    window.GUI.end()

material_type_colors = np.array([
    [0.1, 0.1, 1.0, 1.0],       # water
    [0.9, 0.9, 0.9, 1.0],       # elastic
    [1.0, 1.0, 1.0, 1.0],       # snow
    [1.0, 1.0, 0.0, 1.0],         # sand
    [0.0, 0.0, 0.8, 1.0],       # stationary
    [0.8, 0.8, 0.8, 1.0],       # elastic1
    [0.7, 0.7, 0.7, 1.0] ],     # elastic2
    dtype=np.float32 )

os.makedirs("image", exist_ok=True)
# os.makedirs("VTU", exist_ok=True)

vtu = VTUclass()
motion = Motionclass(fish, mpm, dt, particles_radius)


# precalculation
t = 0.0
time = 0.1
cnt = 0
motion.flg = False

while t<time:
    print(f"time={t:.4f}")
    motion.move( t, mpm.x, mpm.v, time )
    mpm.step(dt, motion)

    if cnt%10 == 0:
        render()
        window.show()

    t += 10.0*dt
    cnt +=1

t = 0.0
cnt = 0
cnt2 = 0
```

```
motion.flg = True

while t<SimulationTime:
        print(f"time={t:.4f}")
        mpm.step(dt, motion)

        if cnt%VTUinterval == 0:
              render()
              # show_options()
              window.save_image(f"./image/frame{cnt2:04}.png")
              window.show()

              # vtu.write_vtuMPMASCII(f"./VTU/SPH{cnt:04}.vtu", mpm )
              cnt2 += 1

        t += dt
        cnt += 1

# for movie
files = sorted(glob.glob("./image/frame*.png"))
if len(files) > 0:
      clip = ImageSequenceClip(files, fps=60)
      clip.write_videofile("out.mp4", codec="libx264", audio=False)
```

(4) **MLS-MPM スキーム概説**

論文はこれ

https://dl.acm.org/doi/10.1145/3197517.3201293

A Moving Least Squares Material Point Method with Displacement Discontinuity and Two-Way Rigid Body Coupling

YUANMINGHU†,MITCSAIL

ACMTrans. Graph., Vol. 37, No. 4, Article 150. Publication date: August 2018.

変位不連続を有する MLS-MPM と剛体双方向連成

Galerkin-style weak form discretiza tion

dynamic open boundaries

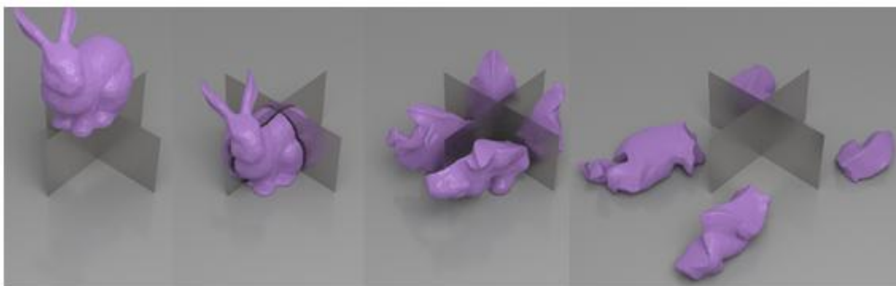


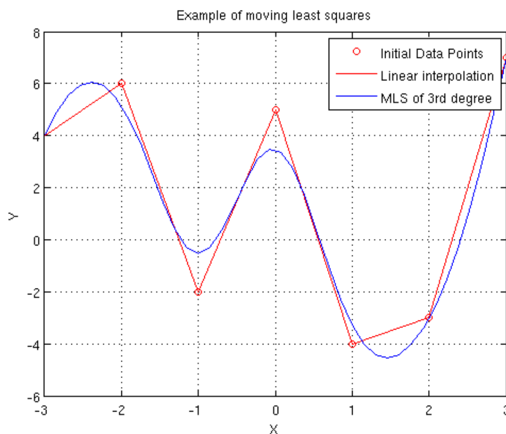
Fig. 4. An elastic bunny is split by two intersecting thin plates.

コアアルゴリズムは，**MLS-MPM with CPIC**

Moving Least Square – Material Point Method with Compatible Particle-In-Cell

. **Moving least squares(MLS):** データ点群から滑らかな曲面や関数を近似的に再構成する手法．各計算点において、近くのデータには重みを大きく、遠いデータには小さくして、加重最小二乗法で局所的な近似を行うため、局所的な形状変化を柔軟に表現でき、コンピュータグラフィックスや物理シミュレーション（SPH 法など）で表面再構成に利用される．評価点に近接するデータ点の影響力を高くしている．

https://graphics.stanford.edu/courses/cs468-05-fall/slides/niloy_levin_fall_05.pdf



Consider a function $f : \mathbb{R}^n \to \mathbb{R}$ and a set of sample points $S = \{(x_i, f_i) | f(x_i) = f_i\}$. Then, the moving least square approximation of degree m at the point x is $\tilde{p}(x)$ where $\tilde{p}$ minimizes the weighted least-square error

$$\sum_{i \in I} (p(x_i) - f_i)^2 \theta(\|x - x_i\|)$$

over all polynomials p of degree m in $\mathbb{R}^n$. $\theta(s)$ is the weight and it tends to zero as $s \to \infty$.

Typical example

- Gaussian kernel $\theta(d) = \exp(-d^2/h^2)$

i はグリッドノード，p は粒子量

Variable	Type	Meaning
u	any	any continuous function approximated with MLS
$\mathbf{x}_i$	vector	the location of sample/node i
$\mathbf{x}_p$	vector	the location of particle p
$\mathbf{z}, \mathbf{x}$	vector	an arbitrary continuous location
$\mathbf{P}(\mathbf{x})$	vector	the polynomial basis
$\mathbf{c}(\mathbf{x})$	vector	all basis coefficients
$\mathbf{M}(\mathbf{x})$	matrix	the moment matrix
$\mathbf{M}_p$	matrix	$\mathbf{M}(\mathbf{x}_p)$
$\xi_i(\mathbf{x})$	scalar	weighting function centered at $\mathbf{x}_i$
$\Phi_i(\mathbf{x})$	scalar	MLS shape function centered at $\mathbf{x}_i$
$N_i(\mathbf{x})$	scalar	B-spline basis function centered at $\mathbf{x}_i$
$\rho(\mathbf{x}, t)$	vector	the continuous density field
$\mathbf{v}(\mathbf{x})$	vector	the continuous velocity field
m_i	scalar	mass of node i
$\mathbf{v}_i$	vector	velocity of node i
$\mathbf{v}_i^n$	vector	velocity of node i at time n over domain Ω^{t^n}
$\hat{\mathbf{v}}_i^n$	vector	velocity of node i at time $n+1$ over domain Ω^{t^n}
m_p	scalar	mass of particle p
$\mathbf{v}_p$	vector	velocity of particle p
$\mathbf{C}_p$	matrix	affine matrix of particle p
$\mathbf{q}$	vector	test function in the weak form
$q_{\alpha,\beta}$	scalar	derivative of q_α wrt. x_β
$\boldsymbol{\sigma}$	matrix	Cauchy stress
$\mathbf{F}_p$	matrix	the deformation gradient on particle p
$\mathbf{f}_i$	vector	the force on grid node i

Element-free Galerkin(EFG)のようなメッシュレス手法として説明する．

$\mathbf{x}_i$ を固定された位置の集合として，MLS を想定した連続の式 u（多項式最小二乗式）を使って，近似点は，

$u_i = u(\mathbf{x}_i)$

となる．u からどの地点の近似位置も計算できる．p は粒子

$$u(\mathbf{z}) = \boldsymbol{P}^T(\boldsymbol{z}-\boldsymbol{x})\boldsymbol{c}(\boldsymbol{x}) = [p_0(\boldsymbol{z}-\boldsymbol{x}), p_1(\boldsymbol{z}-\boldsymbol{x}), \ldots . p_l(\boldsymbol{z}-\boldsymbol{x})]\begin{bmatrix} c_0(\boldsymbol{x}) \\ c_1(\boldsymbol{x}) \\ \\ c_l(\boldsymbol{x}) \end{bmatrix}$$

EFG 法では，c は以下の式を最小化するお文字最小二乗式により評価される．

$$J_{\mathbf{x}}(\mathbf{c}) = \sum_{i \in B_{\mathbf{x}}} \xi_i(\mathbf{x}) \left(\mathbf{P}^T(\mathbf{x}_i - \mathbf{x})\mathbf{c}(\mathbf{x}) - u_i\right)^2$$

ここで，ξ は xi 周りの局所重み関数．Bx は ξ≠0 を満たす指標集合．これより，

$c(\mathbf{x}) = \boldsymbol{M}^{-1}(\boldsymbol{x})\boldsymbol{b}(\boldsymbol{x})$

となる．ここで，

where $\mathbf{b}(\mathbf{x}) = \sum_{i \in B_{\mathbf{x}}} \xi_i(\mathbf{x})\mathbf{P}(\mathbf{x}_i - \mathbf{x})u_i$ and $\mathbf{M}(\mathbf{x}) = \sum_{i \in B_{\mathbf{x}}} \xi_i(\mathbf{x})\mathbf{P}(\mathbf{x}_i - \mathbf{x})\mathbf{P}^T(\mathbf{x}_i - \mathbf{x})$. Note that when **P** only contains linear polynomials, a

以上より，

$$u(\mathrm{z}) = \sum_{i \in B_{\mathbf{x}}} \xi_i(\mathbf{x})\mathbf{P}^T(\mathrm{z}-\mathbf{x})\mathbf{M}^{-1}(\mathbf{x})\mathbf{P}(\mathbf{x}_i - \mathbf{x})u_i, \quad (3)$$

$$u(\mathbf{z}) = \sum_{i \in B_{\mathbf{x}}} \Phi_i(\mathbf{z})u_i,$$

となり，φ は nodal shape function of xi

実際には，

定数バイアスに対して，$\mathrm{P(x)} = [1]^T$

線形バイアスに対して，$\mathrm{P(x)} = [1, x, y]^T$

2 次バイアスに対して，$\mathrm{P(x)} = [1, x, y, xy, x^2, y^2]^T$

$$\sum_i \Phi_i(\mathbf{x}) = 1.$$

m 次元で l=m

3.1.1 線形多項式バイアスのケース

$$\begin{bmatrix} \hat{u} \\ \nabla \hat{u} \end{bmatrix} = \mathbf{M}^{-1}(\mathbf{x})\mathbf{Q}^T \Xi(\mathbf{x}) \begin{bmatrix} u_1 \\ \vdots \\ u_N \end{bmatrix}, \quad (4)$$

N はサンプルデータ数，Ξ(x)は対角成分が ξi(x)の対角重みマトリクス．Ξ は ξ の大文字.

diagonal weighting matrix with $\Xi_{ii} = \xi_i(\mathbf{x})$, and $\mathbf{Q}(\mathbf{x}) = [\mathbf{P}(\mathbf{x}_1 - \mathbf{x}), \ldots, \mathbf{P}(\mathbf{x}_N - \mathbf{x})]^T$, $\mathbf{M}(\mathbf{x}) = \mathbf{Q}^T \Xi \mathbf{Q}$.

．形状関数としての格子（grid）（i）：

質量：mi

速度：vi

．求積点（quadrature points）としての粒子（p）：

粒子は

質量：m_p

位置：x_p

速度：v_p

変形勾配（deformation gradient）：F_p

材料変数：**

を持っている．

時間刻み毎，粒子は質量と速度を格子に送る．格子速度は時間積分されて粒子に戻される．

オイラーアプローチにおける(5)圧縮性の連続の式（質量保存則）と(6)NS 方程式は，以下の通り

3.3.1 Governing equations. We start with the Eulerian governing equations:

$$\frac{D\rho}{Dt} + \rho\nabla \cdot \mathbf{v} = 0 \quad \text{(conservation of mass),} \qquad (5)$$

$$\rho\frac{D\mathbf{v}}{Dt} = \nabla \cdot \boldsymbol{\sigma} + \rho\mathbf{g} \quad \text{(conservation of momentum),} \qquad (6)$$

σ は，応力テンソル（Cauchy stress）

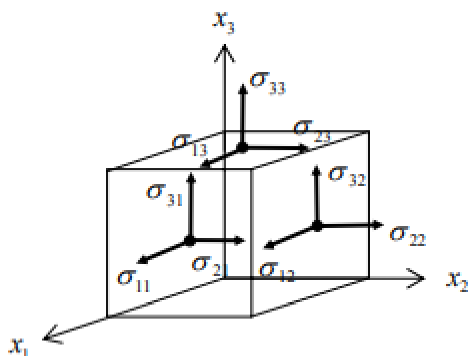
Cauchy's Law

Cauchy's Law states that there exists a **Cauchy stress tensor σ** which maps the normal to a surface to the traction vector acting on that surface, according to

$$\boxed{\mathbf{t} = \boldsymbol{\sigma}\,\mathbf{n}, \qquad t_i = \sigma_{ij} n_j} \qquad \textbf{Cauchy's Law} \qquad (3.3.4)$$

or, in full,

$$\begin{aligned} t_1 &= \sigma_{11}n_1 + \sigma_{12}n_2 + \sigma_{13}n_3 \\ t_2 &= \sigma_{21}n_1 + \sigma_{22}n_2 + \sigma_{23}n_3 \\ t_3 &= \sigma_{31}n_1 + \sigma_{32}n_2 + \sigma_{33}n_3 \end{aligned} \qquad (3.3.5)$$



(6)式の弱形式（weak form）：

$$\frac{1}{\Delta t}\int_{\Omega^{t^n}} \rho(\mathbf{x}, t^n)\left(\hat{v}_\alpha^{n+1}(\mathbf{x}) - v_\alpha^n(\mathbf{x})\right) q_\alpha(\mathbf{x}, t^n) d\mathbf{x}$$
$$= \int_{\partial\Omega^{t^n}} q_\alpha(\mathbf{x}, t^n)\mathcal{T}_\alpha(\mathbf{x}, t^n) ds - \int_{\Omega^{t^n}} q_{\alpha,\beta}(\mathbf{x}, t^n)\sigma_{\alpha\beta}(\mathbf{x}, t^n) d\mathbf{x}, \tag{7}$$

ここで,

Rd：ディリクレ境界$\partial\Omega_D$で消える（？vanish）任意のベクトル値テスト関数（=FEM の形状関数）

q(x,t)：時刻 t における位置 x の粒子場

T(x,t)：境界上の traction field

$\mathbf{v}^n$：現在の n 時刻（ステップ）におけるオイラー速度

$\mathbf{v}^{\wedge n+1}$：外力を計算した後更新された速度場

α：次元=x, y, z

MPM と MLS-MPM の違いは，形状関数．MPM は B-spline，MLS-MPM は上記 nodal 形状関数 φi(x)

．Traditional MPM の離散化

B-spline 基底関数（basis function）Ni(x)によって離散化．この辺は FEM の格子と同じ．

α=x, y, z の任意の時間 t^n と任意の位置 **x** における粒子場 $q_\alpha(\mathbf{x},t^n)=Ni(\mathbf{x})q^n{}_{i\alpha}$

同様に速度場は，$\mathbf{v}^n{}_\alpha(x)=N_j(\mathbf{x})v^n{}_{j\alpha}$

外力更新後の速度場は，$v^{\wedge n+1}{}_\alpha(\mathbf{x},t^n)=N_j(\mathbf{x})\ v^{\wedge n+1}$

lamped mass は，$m_i^n = \sum_p N_i(\mathbf{x}_p^n) m_p$

. MLS-MPM の運動量項

まず，(7)式左の項の積分の離散化積分は，以下の通り．

$$\int_{\Omega^{t^n}} \rho(\mathbf{x}, t^n) v_\alpha^n(\mathbf{x}) q_\alpha(\mathbf{x}, t^n) d\mathbf{x}$$
$$= \sum_p \int_{\Omega_p^{t^n}} \rho(\mathbf{x}, t^n) v_\alpha^n(\mathbf{x}) q_\alpha(\mathbf{x}, t^n) d\mathbf{x}.$$

このとき，時刻 t^n における粒子位置 **x** が求積点となり，求積係数 φ(**x**)として以下のように積分している．ガウスの求積法を見よ．

$$v_\alpha^n(\mathbf{x}) = \sum_j \Phi_j(\mathbf{x}) v_{j\alpha}^n \tag{8}$$

$$q_\alpha(\mathbf{x}, t^n) = \sum_i \Phi_i(\mathbf{x}) q_{i\alpha}^n, \tag{9}$$

$$\Phi_i(\mathbf{x}) = \xi_i(\mathbf{x}_p^n)\mathbf{P}^T(\mathbf{x} - \mathbf{x}_p^n)\mathbf{M}^{-1}(\mathbf{x}_p^n)\mathbf{P}(\mathbf{x}_i - \mathbf{x}_p^n). \tag{10}$$

よって，運動量 mv は,

$$\sum_p \int_{\Omega_p^{t^n}} \rho(\mathbf{x}, t^n) v_\alpha^n(\mathbf{x}) q_\alpha(\mathbf{x}, t^n) d\mathbf{x} = \sum_{p,i,j} q_{i\alpha}^n v_{j\alpha}^n m_{ij}, \qquad (11)$$

となり，ここで，格子 i の質量マトリクス m_{ij} は，最終的に lamped mass として対角成分だけが加算され，以下のスカラー質量値となる．

$$m_i^n = \sum_p \int_{\Omega_p^{t^n}} \rho(\mathbf{x}, t^n) \Phi_i(\mathbf{x}) d\mathbf{x} \approx \sum_p m_p \Phi_i(\mathbf{x}_p^n) = \sum_p m_p N_i(\mathbf{x}_p^n),$$

時刻 n の i 番目の格子の質量

外力による更新速度は，

$$\hat{v}_\alpha^{n+1}(\mathbf{x}) = \mathbf{P}^T(\mathbf{x} - \mathbf{x}_p^n) \mathbf{c}_{\hat{v}_\alpha^{n+1}}(\mathbf{x}_p^n),$$

であり，$\mathbf{c}_{\hat{v}_\alpha^{n+1}}$ は，格子―粒子変換

. MLS-MPM の応力項

ここが本手法の key contribution

ノイマン（勾配）境界条件は 0 として仮定．

離散化積分などは上記と概念は同じ．

$$\begin{aligned} &-\int_{\Omega^{t^n}} q_{\alpha,\beta}(\mathbf{x}, t^n) \sigma_{\alpha\beta}(\mathbf{x}, t^n) d\mathbf{x} \\ &= -\sum_p \int_{\Omega_p^{t^n}} q_{\alpha,\beta}(\mathbf{x}, t^n) \sigma_{\alpha\beta}(\mathbf{x}, t^n) d\mathbf{x}. \qquad (12) \end{aligned}$$

Ng は全グリッド数

i=グリッド番号，α=x，y，ｚのとき 1，それ以外は 0??? グリッド i にかかる x, y, z 方向の力 $F_{\hat{i}\hat{\alpha}}$ に対応させているのか？

$$q_{i\alpha}^n = \delta_{i\hat{i}} \delta_{\alpha\hat{\alpha}} = \begin{cases} 1 & \text{if } \alpha = \hat{\alpha} \text{ and } i = \hat{i} \\ 0 & \text{otherwise} \end{cases}$$

$$q_\alpha(\mathbf{x}, t^n) = \mathbf{P}^T(\mathbf{x} - \mathbf{x}_p^n) \mathbf{M}^{-1}(\mathbf{x}_p^n) \xi_{\hat{i}}(\mathbf{x}_p^n) \mathbf{P}(\mathbf{x}_{\hat{i}} - \mathbf{x}_p^n) \delta_{\alpha\hat{\alpha}} \qquad (13)$$

. 力の離散化

q の導関数が必要なので，(13)式を微分して，

$$q_{\alpha,\beta}(\mathbf{x}, t^n) = \frac{\partial \mathbf{P}^T(\mathbf{x} - \mathbf{x}_p^n)}{\partial x_\beta} \mathbf{M}^{-1}(\mathbf{x}_p^n) \xi_{\hat{i}}(\mathbf{x}_p^n) \mathbf{P}(\mathbf{x}_{\hat{i}} - \mathbf{x}_p^n) \delta_{\alpha\hat{\alpha}}. \qquad (14)$$

計算を簡単にするために，線形多項式空間である

$$\mathbf{P}^T(\mathbf{x} - \mathbf{x}_p^n) = [1, x_1 - x_{p1}^n, x_2 - x_{p2}^n, x_3 - x_{p3}^n].$$

を使って，重みには，２次／３次 B スプラインになる（M^{-1} が定数になる）形状関数

$$\xi_{\hat{i}} = N_{\hat{i}}$$

を使って，(14)式を以下のように単純化する．

$$q_{\alpha,\beta}(\mathbf{x}, t^n) = M_p^{-1} N_i(\mathbf{x}_p^n)(x_{\hat{i}\beta} - x_{p\beta})\delta_{\alpha\hat{\alpha}}, \tag{15}$$

ここで,

$M_p = \frac{1}{4}\Delta x^2$ for quadratic $N_i(\mathbf{x})$ and $\frac{1}{3}\Delta x^2$ for cubic $N_i(\mathbf{x})$.

となる．これを(12)式に代入すると，格子 i^上の力は,

$$\begin{aligned} f_{\hat{i}\hat{\alpha}} &= -\sum_p \int_{\Omega_p^{t^n}} q_{\alpha,\beta}(\mathbf{x}, t^n)\sigma_{\alpha\beta}(\mathbf{x}, t^n) d\mathbf{x} \\ &\approx -\sum_p V_p^n M_p^{-1} \sigma_{p\,\hat{\alpha}\beta}^{\;n} N_{\hat{i}}(\mathbf{x}_p^n)(x_{\hat{i}\beta}^n - x_{p\beta}^n), \end{aligned} \tag{16}$$

となる．ここで，Vnp は，時刻 n における粒子 p の現在の体積．この近似は一点求積による近似．

．変形勾配（deformation gradient）と力

変形勾配 F（勾配を F にしているので注意！）

$$\mathbf{F} = \frac{\partial \mathcal{Z}}{\partial \mathbf{X}}$$

X：材料空間

Z（X, t)：変形マップ

$$\frac{\partial}{\partial t}\mathbf{F}(\mathbf{X}, t) = \frac{\partial \mathbf{v}}{\partial \mathbf{x}}(\mathcal{Z}(\mathbf{X}, t), t)\mathbf{F}(\mathbf{X}, t),$$

$\frac{\partial \mathbf{v}}{\partial \mathbf{x}}$ は，グリッド上で離散化されたオイラー速度勾配

粒子方向速度 Fp は，以下で更新される．

$$\mathbf{F}_p^{n+1} = \left(\mathbf{I} + \Delta t \frac{\partial \hat{\mathbf{v}}^{n+1}}{\partial \mathbf{x}}(\mathbf{x}_p^n)\right)\mathbf{F}_p^n,$$

ここで，traditional な MPM は,

$$\frac{\partial \hat{\mathbf{v}}^{n+1}}{\partial \mathbf{x}}(\mathbf{x}_p^n) = \sum_i \hat{\mathbf{v}}_i^{n+1} \nabla N_i(\mathbf{x}_p^n)^T.$$

を使っているが，MLS-MPM では,

$$\frac{\partial \hat{\mathbf{v}}^{n+1}}{\partial \mathbf{x}} = \mathbf{C}_p^{n+1} \quad \text{and} \quad \mathbf{F}_p^{n+1} = \left(\mathbf{I} + \Delta t \mathbf{C}_p^{n+1}\right)\mathbf{F}_p^n, \tag{17}$$

を使っている．ここで,

$\mathbf{C}_p^{n+1}$ は，affine velocity matrix from APIC. Hyperelasticity（超弾性）を仮定するなら,

total potential energy を $E = \sum_p V_p^0 \Psi_p(\mathbf{F}_p)$ （ここで，V_p^0 は粒子初期体積，Ψ_p をエネルギー密度関数）として，力（エネルギーの空間微分）は，

$$\mathbf{f}_i = -\frac{\partial E}{\partial \mathbf{x}_i} = -\sum_p N_i(\mathbf{x}_p^n) V_p^0 M_p^{-1} \frac{\partial \Psi}{\partial \mathbf{F}}(\mathbf{F}_p^n) \mathbf{F}_p^{n\,T}(\mathbf{x}_i^n - \mathbf{x}_p^n),$$

$$\boldsymbol{\sigma} = \frac{1}{\det(F)} \frac{\partial \Psi}{\partial \mathbf{F}} \mathbf{F}^T \text{ and } \det(\mathbf{F}) V_p^0 = V_p^n.$$

この辺の単純化が，MLS-MPM を traditional MPM より計算コストを小さくしている

. 陰積分

$$-\delta \mathbf{f}_i = \sum_p V_p^0 \mathbf{A}_p \mathbf{F}_p^{n\,T} M_p^{-1} N_i(\mathbf{x}_p^n)(\mathbf{x}_i^n - \mathbf{x}_p^n), \qquad (19)$$

accumulating δfi

これは，粒子を格子へ配置するステップ：particle to grid scatter step

$$\mathbf{A}_p = \frac{\partial^2 \Psi}{\partial \mathbf{F} \partial \mathbf{F}} : \sum_j M_p^{-1} N_j(\mathbf{x}_p^n) \delta \mathbf{u}_j (\mathbf{x}_j^n - \mathbf{x}_p^n)^T \mathbf{F}_p^n.$$

computing Ap

で，これは格子から粒子へ集めるステップ：grid-to-particle gather step

. MLS-MPM steps:

(1): Particles to grid

粒子から格子へ，質量と運動量を転送

(2) Update grid momentum

symplectic Euler もしくは，backward Euler を使って，格子運動量を更新．ここが MPM と異なる

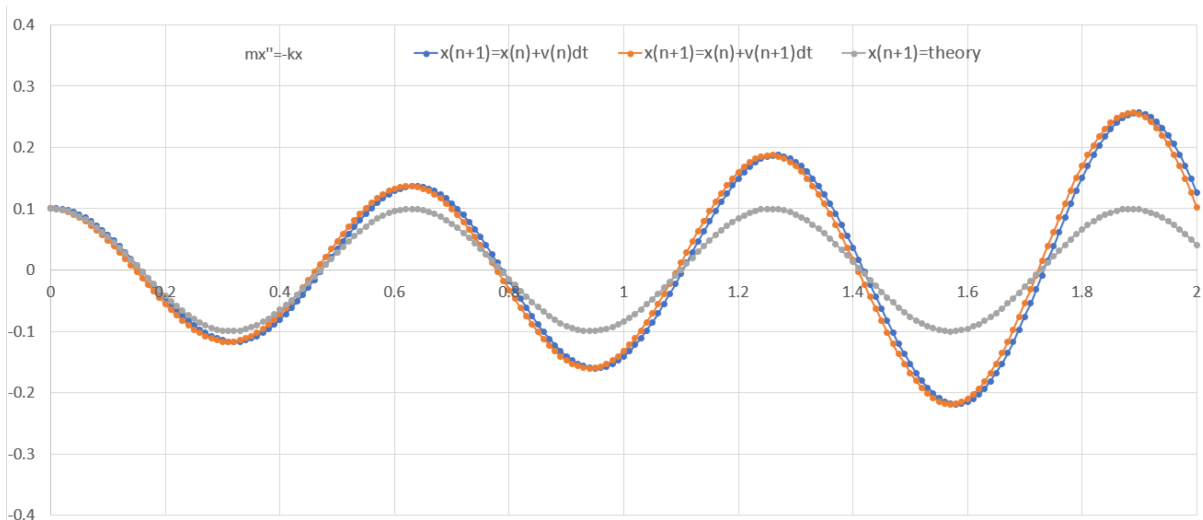
シンプレクティック数値積分法：

https://ja.wikipedia.org/wiki/%E3%82%B7%E3%83%B3%E3%83%97%E3%83%AC%E3%82%AF%E3%83%86%E3%82%A3%E3%83%83%E3%82%AF%E6%95%B0%E5%80%A4%E7%A9%8D%E5%88%86%E6%B3%95

シンプレクティック数値解法 (symplectic numerical method) とは，ハミルトン系，すなわちエネルギーが保存される系の状態を記述する微分方程式のシンプレクティック構造を保存する数値解法のこと

$$v_{n+1} = v_n + \Delta t\, a(x_n)$$
$$x_{n+1} = x_n + \Delta t\, v_{n+1}$$

のように解く．位置 x の更新に v^n ではなく，v^{n+1} を使っている．試しに振動の問題を解いてみると，誤差は変わらないが，周波数のずれはちいさいようだ．



Symplectic Euler 法は，以下のように展開できる．

$$v_{n+1} = v_n - \frac{k}{m} x_n dt$$

$$x_{n+1} = x_n + v_{n+1} dt = x_n + \left(v_n - \frac{k}{m} x_n dt\right) dt = x_n + v_n dt - \frac{k}{m} x_n dt^2$$

よって，Euler 法に対して，$-\frac{k}{m} x_n dt^2$だけ，位置を戻しているようになっている．行列式を求めると 1 になることが分かる．

$$\begin{bmatrix} v_{n+1} \\ x_{n+1} \end{bmatrix} = \begin{bmatrix} 1 & -\frac{k}{m} dt \\ dt & 1 - \frac{k}{m} dt^2 \end{bmatrix} \begin{bmatrix} v_n \\ x_n \end{bmatrix}$$

$$\begin{vmatrix} 1 & -\frac{k}{m} dt \\ dt & 1 - \frac{k}{m} dt^2 \end{vmatrix} = 1 - \frac{k}{m} dt^2 + \frac{k}{m} dt^2 = 1$$

よって，面積を不変に保つ写像，つまり，シンプレクティック写像になっている．なお，誤差が拡大していかないことと，行列式が 1 になることは必ずしも一致しない．

(1) Grid to particles

格子から粒子へ，APIC もしくは，PolyPIC を使って速度と affine/polynomical coefficients を転送

(2) Particle deformation gradient

速度勾配に対し MLS 近似を使って粒子変形勾配を更新．ここが MPM と異なる． C_p^{n+1} の再利用により計算コストは低くなっている．

(3) Update particle plasticity

もしあれば，可塑性（plasticity）のためにプロジェクト粒子変形勾配

このプロセスが，弾性体が切れたりするやつか？？？？

(4) Particle advection

新しい（計算した）速度により，粒子の位置を更新.

. Material discontinuity のための Compatible Particle-in-cell: CPIC

p, q は MPM 粒子

r, s は剛体指標

i, j は格子ノード指標

5.1 Rigid-rigid collision 剛体間同士の衝突

剛体 r の（中間）速度は, $\mathbf{v}_r^* \leftarrow \mathbf{v}_r^n$, 角速度は, $\omega_r^* \leftarrow \omega_r^n$. これを使って, $\mathbf{v}_r^{n+1}$ and ω_r^{n+1} に更新される．5.6 参照

アスタリスク*は中間速度だった？？

Particle color persistence and penalty force

粒子が剛体を突き抜けた時の反力：**f**=kΔx***n** と同じ

$$\mathbf{f}_p^{P,n} = -k_h d_p \mathbf{n}_p, \qquad (22)$$

kh: penalty stiffness parameter 壁反力の跳ね返り係数的なものかな．．．．

np: 壁法線方向ベクトル

dp: 距離

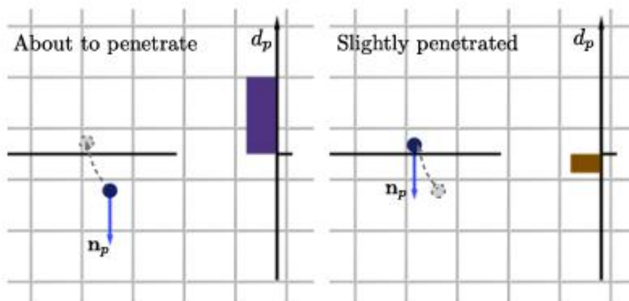


Fig. 13. Here we show the motion of a particle slightly penetrating a boundary due to numerical advection error. In this case, our method robustly maintains a persistent particle color (A_{pr}, T_{pr}) and normal $\mathbf{n}_p$. The reconstructed distance becomes negative when penetration happens. This allows us to apply a weak penalty force as explained in §5.3.3.

Particle grid compatibility

Si: 0 ではない Air をもつ境界面の集合．格子ノード i

Sp: 0 ではない Air をもつ境界面の集合．粒子 p

粒子と格子ノードによってすべての面が分かち合われるなら,

$$(T_{ir} = T_{pr}, \forall r \in S_i \cap S_p)$$

5.4 Particle-to-grid transfer 粒子格子転送

i^{p+}: 粒子 p と compatible な格子ノード i

i^{p-}: じゃない格子ノード i

p^{i+}: 格子ノード i と compatible な粒子 p

p^{i-}: じゃない粒子 p

剛体境界面近傍で，粒子は compatible な格子ノードにのみ転送される．以下，格子ノード i の時刻 n の質量と運動量

$$m_i^n = \sum_{q \in \{p^{i+}\}} N_i(\mathbf{x}_q^n) m_q, \tag{23}$$

$$(m\mathbf{v})_i^n = \sum_{q \in \{p^{i+}\}} N_i(\mathbf{x}_q^n) m_q \left(\mathbf{v}_q^n + \mathbf{C}_q^n (\mathbf{x}_i - \mathbf{x}_q^n) \right). \tag{24}$$

Velocity projection operator: 速度投影オペレータ

それぞれの剛体 r に対して，位置 **x** での世界空間速度（world-space velocity）は，

$$\mathbf{V}_r^n(\mathbf{x}) = \mathbf{v}_r^n + \omega_r^n \times (\mathbf{x} - \mathbf{x}_r^n).$$

となる．並進速度と回転による速度の和

Velocity projection and impulses on rigid bodies

粒子 p，剛体 r として，incompatible な格子ノード i への速度貢献は,

$$\mathbf{Proj}_r(\mathbf{v}_p^n, \mathbf{n}_p, \mathbf{x}_i) = \mathbf{V}_r^n(\mathbf{x}_i) + \mathbf{Proj}(\mathbf{v}_p^n - \mathbf{V}_r^n(\mathbf{x}_i), \mathbf{n}_p, B_r, \mu_r)$$

Br: boundary type

μr：剛体 r の摩擦係数

CPIC においては,

$j \in i^{p-}$ ，つまり，incompatible な格子ノード j の impulse（力積）は,

$$m_p(\mathbf{v}_p^n - \mathbf{Proj}_{r^*}(\mathbf{v}_p^n, \mathbf{n}_p, \mathbf{x}_j)) N_j(\mathbf{x}_p^n)$$

として最も近い剛体粒子 $r^*(\mathbf{x}_j)$ に作用する．

. MLS-MPM grid momentum update 格子運動量更新

i 番目の格子の運動量は以下の式で更新される．

$$(m\hat{\mathbf{v}})_i^{n+1} = (m\mathbf{v})_i^n + \Delta t \left(m_i^n \mathbf{g} + \mathbf{f}_i^n \right),$$

速度は運動量を質量で割って更新される．

$$\hat{\mathbf{v}}_i^{n+1} = (m\hat{\mathbf{v}})_i^{n+1}/m_i^n.$$

5.5 CPIC grid-to-particle transfer: CPIC 格子粒子転送

a ghost velocity approach

粒子 p と incompatible な格子ノードにおいては速度は粒子と同じ？？？

$$\mathbf{v}_j = \mathbf{v}_p^n \quad j \in i^{p-},$$

$$\mathbf{v}_p^{n+1} = \sum_{j \in i^{p-}} N_j(\mathbf{x}_p^n)\tilde{\mathbf{v}}_p + \sum_{j \in i^{p+}} N_j(\mathbf{x}_p^n)\hat{\mathbf{v}}_j^{n+1}, \tag{27}$$

$$\mathbf{C}_p^{n+1} = D_p^{-1}\left(\sum_{j \in i^{p-}} N_j(\mathbf{x}_p^n)\tilde{\mathbf{v}}_p \mathbf{z}_{jp}^{n\ T} + \sum_{j \in i^{p+}} N_j(\mathbf{x}_p^n)\hat{\mathbf{v}}_j^{n+1}\mathbf{z}_{jp}^{n\ T}\right), \tag{28}$$

$$\mathbf{z}_{jp}^n = \mathbf{x}_j - \mathbf{x}_p^n.$$

Dp の特異性を避けるために，incompatible ノード上で ghost velocity を使っている．

衝突している粒子速度は,

$$\tilde{\mathbf{v}}_p = \mathbf{Proj}_r(\mathbf{v}_p^n, \mathbf{n}_p^n, \mathbf{x}_j) + \Delta t c \mathbf{n}_p,$$

r は粒子 p に最も近い剛体の境界．c は境界から押し戻されるときには 0 じゃない．

粒子 p の移動は,

$$\mathbf{x}_p^{n+1} = \mathbf{x}_p^n + \Delta t \mathbf{v}_p^{n+1}.$$

粒子変形勾配は，以下で更新される．

$$\mathbf{F}_p^{n+1} = (\mathbf{I} + \Delta t \mathbf{C}_p^{n+1})\mathbf{F}_p^n.$$

式(22)のペナルティ力積 $\Delta t \mathbf{f}_p^{P,\,r\,n}$ ，を粒子に適用し，剛体にはその反力積を適用する．

5.6 Rigid body advection: 剛体の移動（この分野は，剛体の移動にも advection を使う）

$$\mathbf{v}_r^{n+1} \leftarrow \mathbf{v}_r^*$$

$$\omega_r^{n+1} \leftarrow \omega_r^*,$$

5.1 参照

6 実装

P2G: transfer operation from particle to grid

G2P: transfer operation from grid to particle

従来の MPM ではこの変換に計算コストの 85%を使ってた

MLS-MPM では，SPGrid をつかい，blocked transfer と lock-free multi-threading のための eight-colored P2G を使っている．

高速化のために以下の式を使っている．

into $N_i(\mathbf{x}_p^n)\mathbf{Q}_p(\mathbf{x}_i - \mathbf{x}_p^n)$, where

$$\mathbf{Q}_p = \Delta t V_p^0 M_p^{-1} \frac{\partial \Psi}{\partial \mathbf{F}}(\mathbf{F}_p^n)\mathbf{F}_p^{nT} + m_p \mathbf{C}_p,$$

8*10^6 の粒子でベンチマークしてみた．8M 粒子≒2^23 粒子

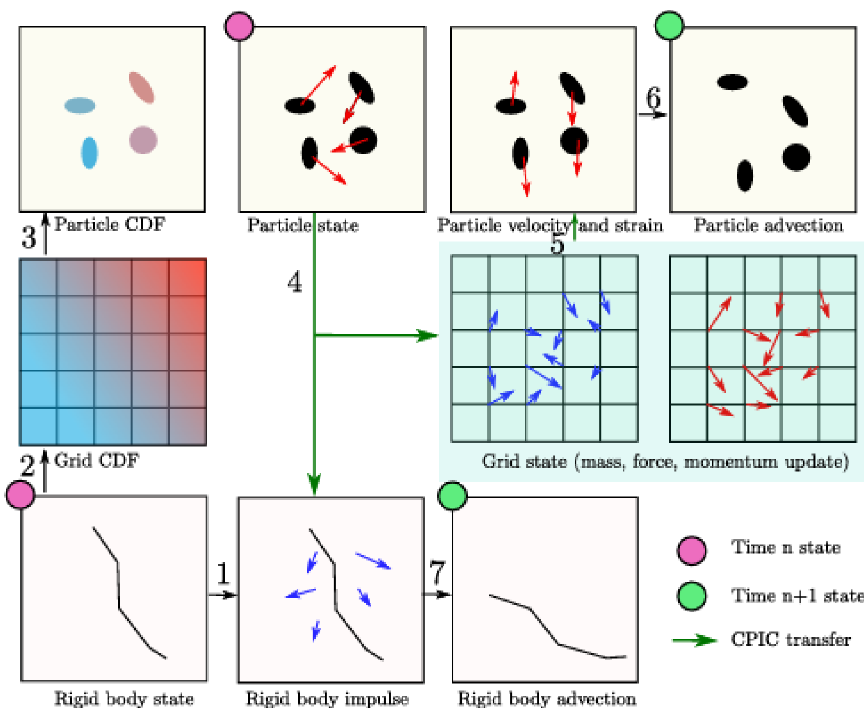


Fig. 10. Algorithm overview from time t^n to t^{n+1} for MLS-MPM with CPIC. Steps: (1) Rigid-rigid collision and rigid body articulation update rigid body velocities (§5.1); (2) Splat rigid body to grid CDF (§5.2); (3) Reconstruct particle CDF from grid CDF (§5.3); (4) CPIC particle-to-grid transfer and rigid body impulses (§5.4); (5) CPIC grid-to-particle transfer (§5.5); (6) MPM particle advection (§5.5); (7) Rigid body advection (§5.6).

. Affine Particle In-Cell (APIC)

C. Jiang, C. Schroeder, A. Selle, J. Teran, and A. Stomakhin. 2015. The affine particle-in cell method. ACM Trans Graph 34, 4 (2015), 51:1–51:10.

https://dl.acm.org/doi/10.1145/2766996

Grid と Particle

Compatible Particle-In-Cell (CPIC)

. Particle in-Cell (PIC)

. Fluid Implicit Particle (FLIP)

. Polynomial Particle-In-Cell
APolynomial Particle-In-Cell Method

. eXtendedFiniteElementMethod(XFEM)
Robust eXtended finite elements for complex cutting of deformables. ACM Trans Graph 36, 4 (2017), 55.

. VirtualNode Algorithm (VNA)
Molino et al. 2005

Material Point Method (MPM):
is a hybrid Lagrangian/ Eulerian discretization scheme for solid mechanics and a generalization of the FLIP

FLIP-based fluids

element-free Galerkin (EFG)

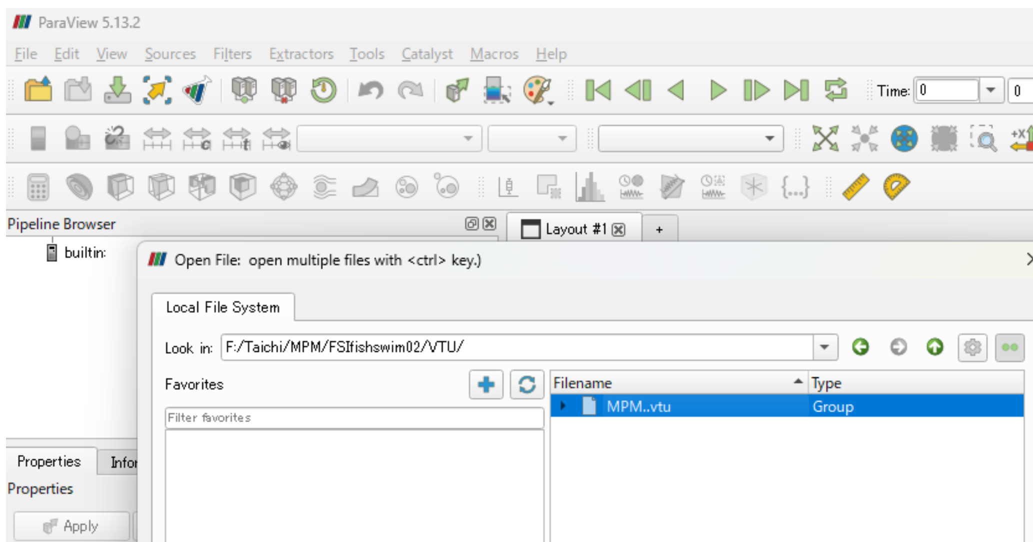
. Smoothed Particle Hydrodynamics (SPH)
position-based dynamics (PBD)

(5) **実行**
Taichi をアクティベイトし，メインプログラムがあるディレクトリ時で実行

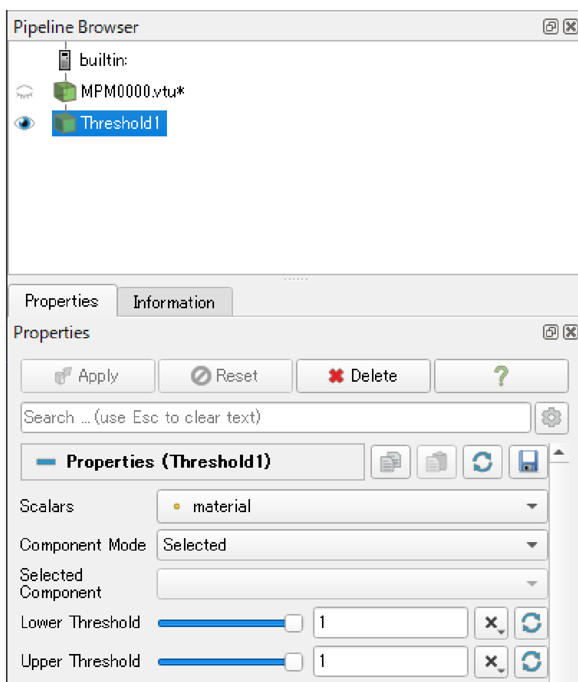
```
> source taichi_env/bin/activate
> python3 FSIswim01.py
```

Taichi の GUI で描いた画像は，image ディレクトリに，
VTU ファイルは VTU ディレクトリに保存される．

(6) **可視化**
VTU ファイルを paraview の gaussian resampling で表示してみる．File->Open から折りたたまれている VTU ファイルを読むと時系列で読み込める．



Apply したあと，Threshold で Lower Threshold=1, Upper Threshold=1 にすると弾性体＝魚が選択される．



Filters の gaussian resampling を選択し，Resampling Grid を x, y, z すべて 128 にし，魚が動くであろう領域を Extent to Resampling に座標で入力（min, ,max の順番になっており，0,0 にすると初期値に戻される．よって，x, y, z すべて入力する）

x 範囲：　0 to 0.4 [m]

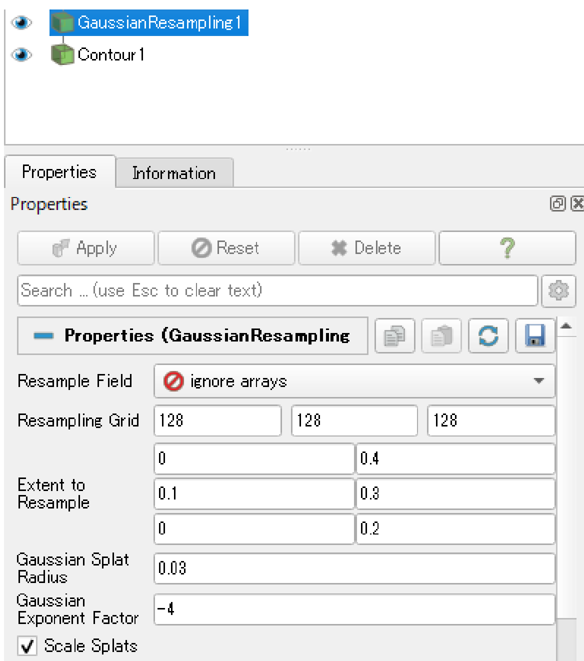
y 範囲：　0.1 to 0.3 [m]

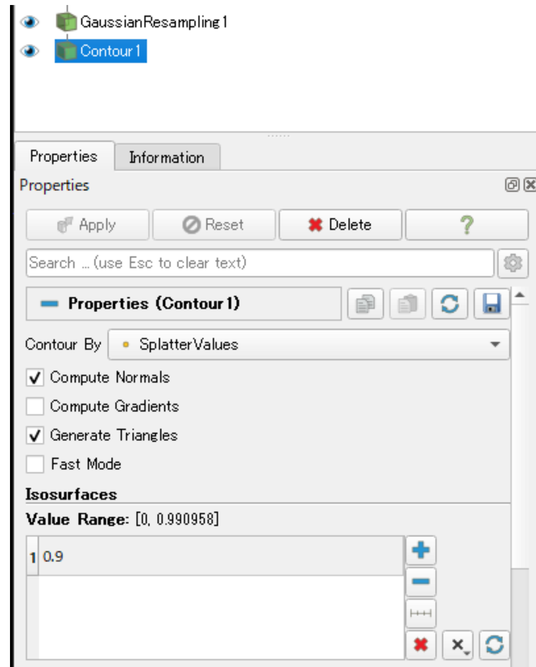
z 範囲：　0 to 0.2 [m]

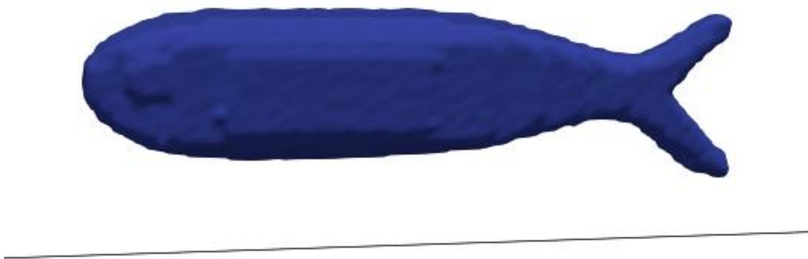
Gaussian Splat radius: 0.03 [m]　（格子サイズ 0.0031 なので 10 倍ぐらい？）

Gaussian Exponent Factor: -4

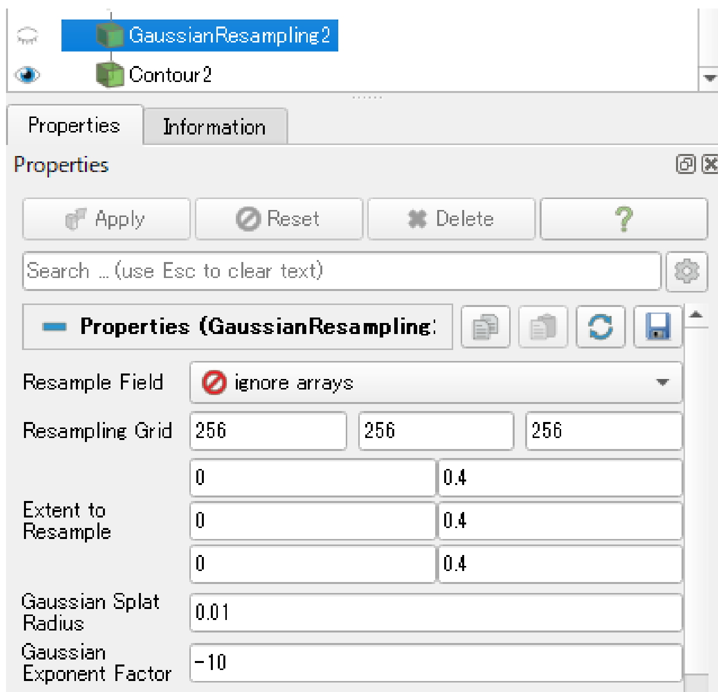
その後，コンタ図にする．material=1 が弾性体なので 0.9 ぐらい．

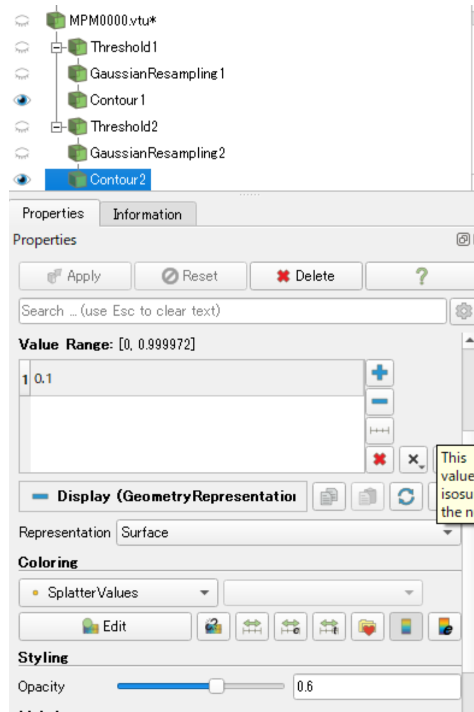


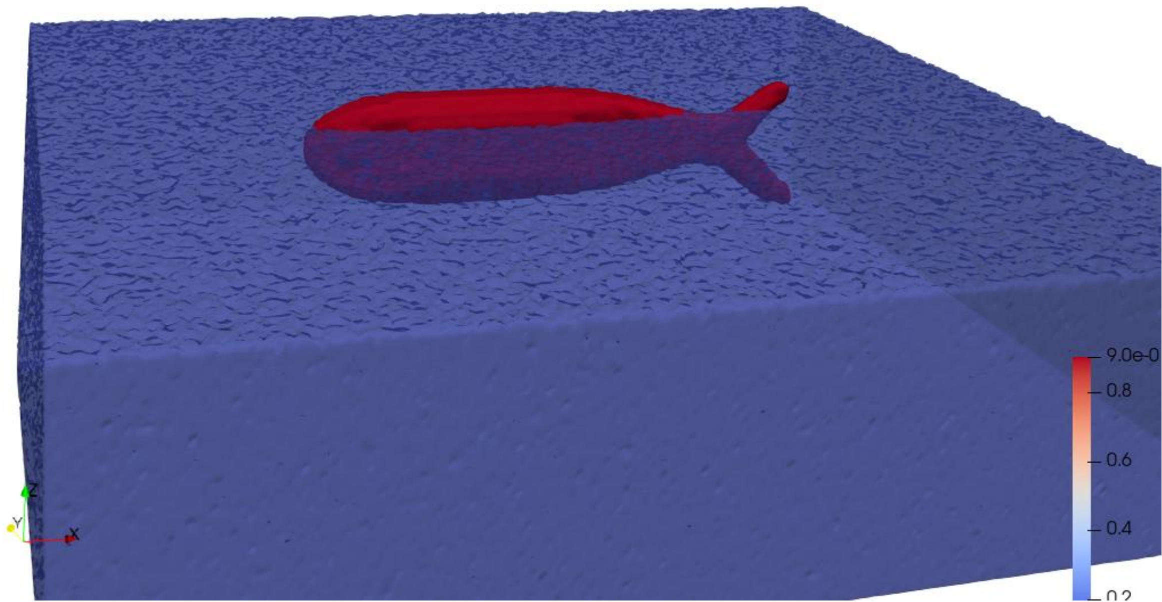




水に関しても同様に表示する．Threshold の material=0 が水．contour の Value range は，水が 0 なので，0.1 ぐらい．







このあと動画にする．File->animation で frame rate を設定する．

Save Animation Options

Search ... (use Esc to clear text)

Size and Scaling

Image Resolution 1124 x 738

Coloring

Override Color Palette: No change

Transparent Background

File Options

Format: MP4 Writer

File Name

Bit Rate: 10000000

Animation Options

Frame Rate: 15

Cancel OK