Apple Cut by MLS-MPM of Taichi

19, Mar. 2026

MLS-MPM の弾性体粒子のリンゴを三角形メッシュで構成された剛体ナイフで切ってみる．一つの弾性体を二つに分割する（トポロジーを変更する）という意味合いと，粒子法とマルチボディの双方向動力学連成問題という意味合いがある．マルチボディは，CAD の obj ファイルを使えるように汎用化してある．なお，ナイフも粒子で構成した動力学計算は，粒子同士がすり抜け，うまくいかなかった．ナイフの刃幅に対して粒子が 6 個（格子 3 個）あればおそらくすり抜けないが，その解像度では計算時間がかかりすぎて現実的ではない．粒子 3 個ではすり抜けた．

ナイフ：

. 刃渡り 24cm，刃幅 10cm，厚さ 2mm〜4mm,

. 質量 0.2 kg，慣性は適当（入力したが，計算していない）

リンゴ：

https://sketchfab.com/search?q=apple&type=models

ここのどれかからダウンロード

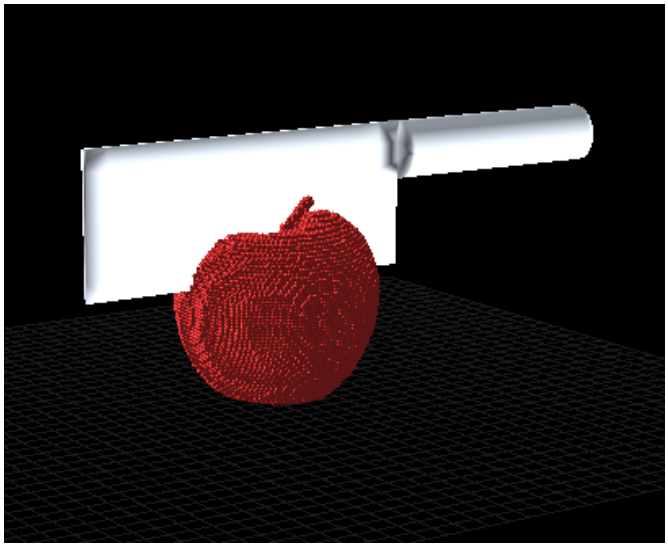
. obj ファイル，テクスチャ付き

. 質量 0.2 kg

計算空間：

0.6m*0.6m*0.6m

解像度 128*128*128



(1) 環境設定

Taichi のインストールなどは，菊池研 HP 参照.

以下を Taichi 環境を activate した状態でさらにインストール．近傍探索などで使っているようだ.

python -m pip install rtree

(2) 実行ファイル

```
apple.obj        # リンゴの CAD obj ファイル,
apple.jpg        # リンゴのテクスチャ画像
knife2.obj       # ナイフの CAD obj ファイル
AppleCut.py      # メインコード
GUI.py           # 実行時の GUI．最終的には無くてよい
motion.py        # ナイフの動力学計算
parameters.py    # シミュレーション条件設定ファイル
VTU.py           # 可視化用 VTU 生成ファイル
```

(2-1) parameters.py

```
import numpy as np                          # python 用の数値計算ライブラリ

SimulationTime = 2.0                        # [s]　シム y レーション実行時間
CalculationSpace = 0.6                      # [m]　立方体の計算空間の一辺
Resolution = 128                            # 計算空間の格子の解像度
gravity = ( 0.0, 0.0, -9.81 )               # 重力
dt = 1e-4                                   # 時間刻み
myu = 0.3                                   # 摩擦係数
airresistance = 1.0                         # c，空気抵抗の意味合い．実際は速度抵抗 v *=e^(c*dt)
VTUinterval = 20                            # VTU saving interval
epsilon = 0.1                               # restitution coefficient　反発係数

apple = {                                   # リンゴの定義
        "fileName": 'apple.obj',
        "translation": [CalculationSpace/2.0, CalculationSpace/2.0, 0.075],      # CAD 座標系から計
算座標系へ
        "scale": [1, 1, 1],                 # x, y, z スケール
        "mass": 0.2,                        # 質量
        }

knife = {
        "fileName": 'knife2.obj',
        "translation": [CalculationSpace/2.0+0.1, CalculationSpace/2.0, CalculationSpace/2.0-0.06], #
CAD 座標系から計算座標系へ
        "scale": [1, 1, 1],
        "COGx": [0, 0, 0],
        "COGv": [0, 0, 0],
        "mass": 0.2,
        "inertia": [[1,0,0],[0,1,0],[0,0,1]],  # 慣性テンソル，使ってない
        }
```

(2-2) AppleCut.py

メインコード，実行ファイル．これが他のファイルを呼び出している．

Python（CPU）で実行するか，Taichi（GPU or GPU 仮想空間）で計算するかで，メモリの確保から数学の関数まで異なるのでコードが煩雑になっている．

```
import taichi as ti                 # おおもとの taichi ライブラリ．ti として使う．
import numpy as np                  # python のメモリや数学の関数
```

```
import os                               # linux のコマンドを使うライブラリ
from PIL import Image
import glob
import trimesh                          # obj ファイルのメッシュを操作するライブラリ

from moviepy import ImageSequenceClip   # 動画関係ライブラリ，使ってない

from engine.mpm_solver import MPMSolver # 粒子法のメインソルバファイルの関数
from VTU import VTUclass                # 可視化用 VTU ファイル生成ライブラリ
from motion import Motionclass          # ナイフの運動学計算クラス
from trimesh.exchange.export import export_mesh    # メッシュ関係保存用ライブラリ

# Parameters
import parameters as para               # パラメータファイルの読み込み．para.*で変数を使う

ti.init(arch=ti.cpu)                    # GPU がないので，CPU 計算に変更
import GUI                              # GUI ファイルのインポート．最終的には使わない

mpm = MPMSolver(res=(para.Resolution, para.Resolution, para.Resolution), size=para.CalculationSpace, max_num_particles=2 ** 21, use_ggui=True)        # 粒子の生成
mpm.add_CADmodel(para.apple["fileName"], offset=para.apple["translation"], material=mpm.material_elastic)    # リンゴ粒子の生成

mpm.set_gravity(para.gravity)           # 重力設定

os.makedirs("VTU", exist_ok=True)       # VTU ディレクトリ生成

vtu = VTUclass()                        # VTU クラスコンストラクタ
knifemotion = Motionclass(para.knife, mpm, para.dt) # 運動クラスコンストラクタ．ナイフ生成

t = 0.0                                 # simulation 時間
cnt = 0                                 # カウンタ
cnt2 = 0                                # カウンタ
#motion.flg = True

faces = np.asarray(knifemotion.mesh.faces.astype(np.int32), dtype=np.int64)    # GUI 表示用メモリ確保

while t<para.SimulationTime:
    print(f"time={t:.4f}")
    nearestx, inside_pid = knifemotion.collision( mpm.x )       # ナイフ衝突粒子判定. trimesh が taichi field で計算できないため，ここで計算し，結果を taichi field に渡すようにしている.
    mpm.step(para.dt, knifemotion, nearestx, inside_pid)        # 粒子ソルバ計算

    if cnt%para.VTUinterval == 0:                               # データ保存＆画像表示ルーチン
        GUI.render(mpm, knifemotion.obj_vertices, knifemotion.indices)
        #window.save_image(f"./image/frame{cnt2:04}.png")
        GUI.window.show()

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

        verts = np.asarray(knifemotion.vertices.to_numpy(), dtype=np.float64)

        mesh = trimesh.Trimesh(vertices=verts, faces=faces, process=False, validate=False, )
        export_mesh( mesh, f"./VTU/MPM{cnt:04d}.stl", file_type="stl", )                    # "stl_ascii"

    t += para.dt
    cnt += 1
```

(2-3)　motion.py
ナイフの動力学計算と，粒子との双方向計算を行っている．コードは長いので略.
クラス内の関数の定義は以下の通り：
Case A: Python（CPU メモリ領域）で計算する場合には,
def A(…):
. . .

Case B: Taichi 環境（GPU or GPU 仮想環境）で計算する場合には,
@ti.kernel
def A(….):
….

Case B-1: Taichi 環境（GPU or GPU 仮想環境）で計算する関数から呼ばれる場合には,
@ti.func
def A(….):
….
となっている．Case A, B では，メモリ確保の関数も，数学のライブラリも異なる．Case A のメモリを Case B に渡す場合には，Case B 用にメモリを確保しなおし，コピーしてから渡す必要がある．三角関数を使う場合にも,
Case A では，math.cos(…)とか numpy を np として使っているなら np.cos(…)とか
Case B では，taichi を ti としているなら，ti.math.cos(….)とか
極めて煩雑なので，流れとしては，気にせずに適当にどっちかを使って書き，chatGPT に修正してもらい，書き換えられた内容を確認して使う，というような方法がよいのかもしれない.

．運動方程式
粒子の質量に対してスケーリングされている．粒子一つの質量当たりの運動方程式になっている.
粒子の並進の式は，以下のように展開される．m は粒子の質量．$\boldsymbol{f}_{\mathrm{p}}$ は粒子一つの質量あたりの力ベクトル.

$$\mathrm{m}\ddot{\mathbf{x}} = \boldsymbol{f}$$

$$\frac{\Delta v}{\Delta t} = \frac{\boldsymbol{f}}{m} = \boldsymbol{f_p}$$

これより，Symplectic Euler 法で時間積分して,

$$\boldsymbol{v} \mathrel{+}= \boldsymbol{f_p}\Delta t$$

$$\boldsymbol{x} \mathrel{+}= \boldsymbol{v}\Delta t$$

となる．剛体（ナイフ）は，粒子と剛体の質量比（実際は比重比）Rmass をもちいて,

$$\mathbf{v}_{obj} \mathrel{+}= \mathrm{R}_{mass} \sum \boldsymbol{f_p}\, \Delta t$$

$$\boldsymbol{x}_{obj} \mathrel{+}= \boldsymbol{v_{obj}}\Delta t$$

回転運動は，粒子にはなく，剛体は以下のようになる．すべて計算空間座標系変数.

$I\dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times (\boldsymbol{I\omega}) = \boldsymbol{\tau}$

モーメント τ は，剛体中心から粒子までの位置ベクトル $\boldsymbol{p}_{ci}$ と粒子にかかる力ベクトルの反力の外積.

$$\boldsymbol{\tau} = \mathrm{R}_{mass} \sum \boldsymbol{p_{ci}} \times (-\boldsymbol{f_p})$$

これより，以下のように展開される.

$$\frac{\Delta \boldsymbol{\omega}}{\Delta t} = I^{-1}(\boldsymbol{\tau} - \boldsymbol{\omega} \times (\boldsymbol{I\omega}))$$

こちらも Symplectic Euler 法で時間積分する．I0 は剛体座標系慣性テンソルで定数.

$I = RI_0R^T$

$\boldsymbol{\omega^{new}} += I^{-1}(\boldsymbol{\tau} - \boldsymbol{\omega} \times (\boldsymbol{I\omega}))\Delta t$

$\Delta \boldsymbol{\theta} = \boldsymbol{\omega^{new}} \Delta t$

$R^{new} = R_n R$

となる．ここで，R は物体の姿勢行列，I_0 は物体の初期慣性テンソル，Rn はロドリゲス回転行列.

$$\mathrm{R}_n = \begin{bmatrix} {n_x}^2(1-cos\alpha)+cos\alpha & n_x n_y(1-cos\alpha)-n_z sin\alpha & n_x n_z(1-cos\alpha)+n_y sin\alpha \\ n_x n_y(1-cos\alpha)+n_z sin\alpha & {n_y}^2(1-cos\alpha)+cos\alpha & n_y n_z(1-cos\alpha)-n_x sin\alpha \\ n_x n_z(1-cos\alpha)-n_y sin\alpha & n_y n_z(1-cos\alpha)+n_x sin\alpha & {n_z}^2(1-cos\alpha)+cos\alpha \end{bmatrix}$$

$$= I_{33} + Ksin\alpha + K^2(1-cos\alpha)$$

where

$$\alpha = \|\Delta\boldsymbol{\theta}\|, n_x = \frac{\theta_x}{\|\Delta\boldsymbol{\theta}\|}, n_y = \frac{\theta_y}{\|\Delta\boldsymbol{\theta}\|}, n_z = \frac{\theta_z}{\|\Delta\boldsymbol{\theta}\|}, \mathrm{I}_{33} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad K = \begin{bmatrix} 0 & -n_z & n_y \\ n_z & 0 & n_x \\ -n_y & n_x & 0 \end{bmatrix}$$

なお，この計算を float（有効数字 7 桁）でやるのは無理がある．実際は，こうした.

．衝突判定とその反力に関して

trimesh(https://trimesh.org/index.html)のレイキャスティングで対象とする点 p が obj の内側なのか，外側なのかを判定している．まず，剛体（ナイフ）の obj ファイルを以下で読み込んで trimesh クラスの変数（mesh）とする.

mesh = trimesh.load_mesh(“knife02.obj”)

その後，以下で点群（points）のある点が obj の内部か外部かをレイキャスティングで判定，内部なら true 外部なら faulse の一次元配列にして mask として返す

mask=mesh.contains(points)

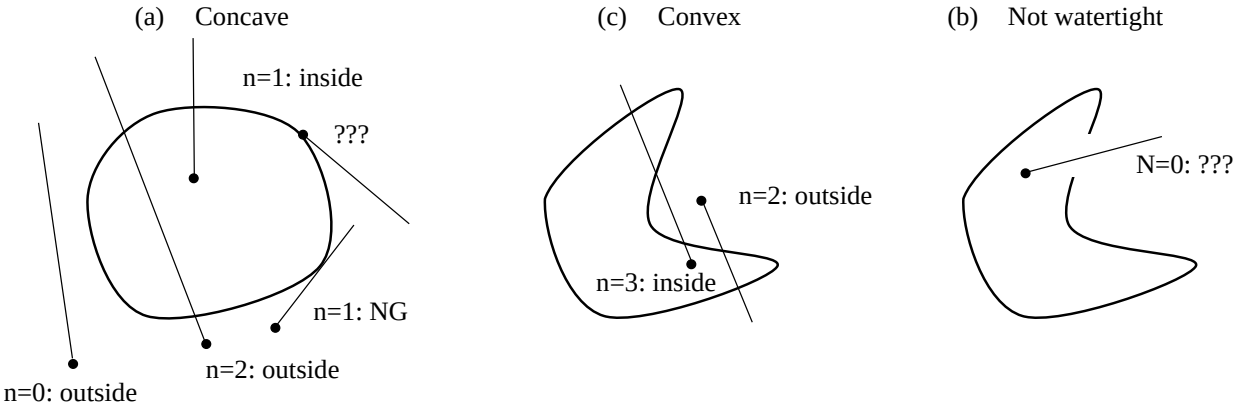
例えば,

points[0]は，内部の点
points[1]は，外部の点
・・・

なら，

mask=[True, False, ….]

となる．ここで点群は，mls-mpm の弾性体粒子，つまりリンゴ．
なお，レイキャスティングは，点から線を伸ばし，obj にぶつかった回数が奇数なら内部，偶数なら外部にするアルゴリズム．穴があいている obj ファイルには使えない．穴があいている場合には，Not watertight というメッセージが出る．実はリンゴにもこのエラーがでているが，枝の部分であると判断し，無視している．



次に，内部の点 $\boldsymbol{x}_i$ を用いて，壁境界上の最も近い点 $\boldsymbol{x}_{\text{nearest}}$ を求め，その方向に反力を与える．ここで，内部の点から最も近い壁境界までの点ベクトル

$$\boldsymbol{p}_i = \boldsymbol{x}_{nearest} - \boldsymbol{x}_i$$

の単位ベクトルは，必ず壁境界面の法線（直角）になる．これを用いて，物体表面に向かう反力，および，壁境界面に平行な摩擦力を求める．

nearestx, dist, tri_id = mesh.nearest.on_surface(inside_point)

で，内部の点からみた最も近い境界の点（nearest point：$\boldsymbol{x}_{\text{nearest}}$）とその距離（dist）と，その obj 上の三角メッシュの番号が配列として計算される．ただし，後者ふたつは結局使っていない．

Case I: 静止している剛体に粒子が衝突

単純化のために，止まっている物体に粒子が衝突した（めり込んだ）状態を考える．粒子を $\boldsymbol{x}_i$，一旦物体が粒子質量に対して十分大きい（粒子の反力で動かない，もしくは一方向連成）とすると，
物体法線方向の反力は，

$$\boldsymbol{f_n} = k\boldsymbol{p_i} + c\left(\frac{\boldsymbol{p}_i}{\|\boldsymbol{p_i}\|}\cdot v_i\right) = \frac{m}{dt}\triangle \boldsymbol{v_n} \cong \frac{m}{dt}[\frac{\boldsymbol{p}_i}{dt} - (\boldsymbol{n}_n \cdot \boldsymbol{v}_i)\boldsymbol{n}_n]$$

ここで，k は物体の剛性でバネ定数，c は粘性抵抗，

$$\mathbf{n}_n = \frac{\boldsymbol{p_i}}{\|\boldsymbol{p}_i\|}$$

である（ベクトルはボールド）．右辺の $\Delta\mathbf{v}$ は，dt 秒後に物体内部にめり込んでいた粒子が物体表面まで移動し，物体表面の法線方向速度を 0 にするための速度である．跳ね返り係数 0 という意味合いに近い．跳ね返り係数 e を考慮するなら，

$$\boldsymbol{f_n} = k\boldsymbol{p_i} + c\left(\frac{\boldsymbol{p}_i}{\|p_i\|}\cdot v_i\right) = \frac{m}{dt}\triangle\boldsymbol{v_n} \cong \frac{m}{dt}[\frac{\boldsymbol{p_i}}{dt} - (1+e)(\boldsymbol{n_n}\cdot\boldsymbol{v}_i)\boldsymbol{n_n}]$$

となる．一般に，k, c を用いた床のモデルはめり込んだ状態で釣り合うが，ここではめり込まないようにしてある．これでどれぐらい現象を正しくモデル化できるか後日検証する必要がある．
dt 秒後も外力が無いなら壁から法線方向に対し$-\mathrm{e}(\boldsymbol{n}_n\cdot\boldsymbol{v}_i)\boldsymbol{n_n}$の速度で離れようとする．これより，

$$\triangle\boldsymbol{v}_n = \frac{\boldsymbol{p}_i}{dt} - (1+e)(\boldsymbol{n_n}\cdot\boldsymbol{v}_i)\boldsymbol{n_n}$$

一方，摩擦力の方向は，$\boldsymbol{p}$ と $\boldsymbol{v}$ ベクトルを含む平面にあり，かつ，壁に平行なので，

$$\boldsymbol{n_t} = \frac{\boldsymbol{p}_i\times(\boldsymbol{p_i}\times\boldsymbol{v}_i)}{\|\boldsymbol{p}_i\times(\boldsymbol{p_i}\times\boldsymbol{v}_i)\|}$$

となり，これより，動摩擦力は，摩擦係数を μ として

$$\boldsymbol{f_t} = \mu\|\boldsymbol{f}_n\|\boldsymbol{n_t} = \frac{m}{dt}\triangle\boldsymbol{v_t}$$

となる．これより，

$$\mu\left\|\frac{m}{dt}\triangle\boldsymbol{v}_n\right\|\boldsymbol{n_t} = \frac{m}{dt}\triangle\boldsymbol{v}_t$$

から，動摩擦による速度変化は，

$$\triangle\boldsymbol{v_t} = \mu\|\triangle\boldsymbol{v_n}\|\boldsymbol{n_t}$$

となる．以上より，dt 秒後の $\boldsymbol{v}_\mathrm{i}$ の速度は，

$$\boldsymbol{v_i}{+}{=}\triangle\boldsymbol{v}_n + \triangle\boldsymbol{v}_t$$

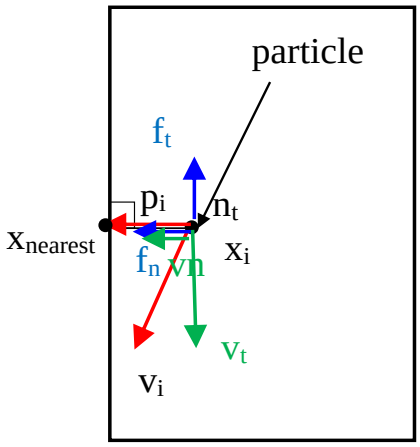
で更新すればいいことになる．今粒子の質量を m_p，物体の質量を m_o として，その比率を求めると，

$$R_{mass} = \frac{m_p}{m_o}$$

となり，粒子から物体への反力により，物体の速度の更新分は，

$$\boldsymbol{v_{obj}}{-}{=} R_{mass}(\triangle\boldsymbol{v_n} + \triangle\boldsymbol{v}_t)$$

となる．



object = knife

Case II: 一般化（物体も粒子も動いている）

動いている物体の場合, めり込んでいる粒子位置の物体の速度 $\mathbf{V}_{obj}$ との相対速度で計算することになる. 上記から,

$$\triangle \boldsymbol{v_n'} = \frac{\boldsymbol{p_i}}{dt} + (\boldsymbol{V}_{obj} \cdot \boldsymbol{n_n})n_n - (\boldsymbol{n_n} \cdot \boldsymbol{v}_i)n_n = \frac{\boldsymbol{p_i}}{dt} - \big(\boldsymbol{n_n} \cdot (\boldsymbol{v}_i - \boldsymbol{V_{obj}})\big)\boldsymbol{n}_n$$

$$\mathbf{n_t}' = \frac{\boldsymbol{p}_i \times (\boldsymbol{p_i} \times (\boldsymbol{v}_i - \boldsymbol{V}_{obj}))}{\|\boldsymbol{p}_i \times (\boldsymbol{p_i} \times (\boldsymbol{v}_i - \boldsymbol{V}_{obj}))\|}$$

$$\triangle \boldsymbol{v_t}' = \mu \| \triangle \boldsymbol{v_n}' \| \boldsymbol{n_t}'$$

$$\boldsymbol{V}_{obj} -= R_{mass}(\triangle \boldsymbol{v_n}' + \triangle \boldsymbol{v_t}')$$

となる. なお, trimesh は, python（CPU）なので, 計算した結果を ti.field にして taichi（GPU）に渡さなければならない. 結局計算は遅い. trimesh の contains や nearest.on_surface を taichi.field の中で書けば早くなるがこれに関しては後日検討. 以下, dynamics のコードのみ抜粋.

```
    @ti.kernel
    def dynamics(self, t: ti.f32, px: ti.template(), pv: ti.template(), nearestx: ti.types.ndarray(dtype=ti.f32, ndim=2), inside_pid: ti.types.ndarray(dtype=ti.i32, ndim=1) ):
        COG0 = self.COGx[None]
        F_m = 600.0*t / self.mass                    # 60.0 [N], force/mass
        dVobj = ti.Vector([0.0,0.0,0.0])             # ナイフの速度変化分

        for i in range(inside_pid.shape[0]):         # ナイフ内部の点
            nearest = ti.Vector([ nearestx[i, 0], nearestx[i, 1], nearestx[i, 2], ])  # xnearest
            p = nearest - px[ inside_pid[i] ]                                         # pi
            nn = p.normalized(1.0e-12)                                                # normal vector
            v0 = p / self.dt
            vi = pv[ inside_pid[i] ] - self.COGv[None]
            k = ti.math.dot(nn,vi)
            vn = v0 - (1.0+self.epsilon) * k * nn

            tempx = ti.math.cross(p,vi)
            tempn = ti.math.cross(p,tempx)
            nt = tempn.normalized(1.0e-12)
            vt = self.myu * vn.norm() * nt
```

```
            pv[ inside_pid[i] ] += vn + vt
            dVobj -= self.massratio * (vn + vt)                    # for reaction force/mass

        self.COGv[None] += ( dVobj + self.gravity[None] * self.dt )

        # knife motion
        if 0.1 < self.COGx[None][2]:          # cutting force
           self.COGv[None][2] -= F_m * self.dt
        else:                                 # if knife is on ground
           if -1.0 < self.COGv[None][1]:      # slide the knife
              self.COGv[None][1] -= F_m/10.0 * self.dt

        self.COGv[None] *= ti.math.exp( - self.airresistance * self.dt )  # air resistance
        self.COGx[None] += self.COGv[None] * self.dt                                    #

        if self.COGx[None][2] < 0.1:
            self.COGx[None][2] = 0.1
            self.COGv[None][2] = 0.0

        dx = self.COGx[None] - COG0

        for i in self.vertices:
            self.vertices[i] += dx

        # for graphic
        for i in self.obj_vertices:
            self.obj_vertices[i] += dx
```

(2-4)　mpm_solver.py

mls-mpm のソルバ．この粒子速度更新の時に，上記動力学に基づく速度変化を加えている．mpm は，固定された構造格子（Eulerian description）と計算空間以内を動く粒子（Lagrange description）の両方を使って計算を行っている．格子の力学情報に基づいて粒子の運動を更新し，粒子の運動により格子の力学情報を更新している．詳細は，菊池研 HP の Taichi 参照.

ソルバクラスコンストラクタ：

```
@ti.data_oriented
class MPMSolver:
    material_water = 0
    material_elastic = 1
    material_snow = 2
    material_sand = 3
    material_stationary = 4
    material_elastic1 = 5              # 物性の異なる弾性体を計算できるようにした
    material_elastic2 = 6
    materials = {
        'WATER': material_water,
        'ELASTIC': material_elastic,
        'SNOW': material_snow,
        'SAND': material_sand,
        'STATIONARY': material_stationary,
        'ELASTIC1': material_elastic1,
        'ELASTIC2': material_elastic2,
    }

    # Surface boundary conditions
```

```
    # Stick to the boundary
    surface_sticky = 0
    # Slippy boundary
    surface_slip = 1
    # Slippy and free to separate
    surface_separate = 2

    surfaces = {
        'STICKY': surface_sticky,
        'SLIP': surface_slip,
        'SEPARATE': surface_separate
    }

    def __init__(
            self,
            res,
            size=1,
            max_num_particles=2**30,      # Max 1 G particles
            padding=3,
            unbounded=False,
            dt_scale=1,
            use_bls=True,
            CFLC=0.9,                             # 0.0 for no CFL condition

            water_SG=1.0,                 # water specific gravity
            Ewater = 1.0e6,               # scaled Young's modulus for elastic
            nuwater = 0.2,                # Poisson's ratio for elastic

# ***_SG は，specific gravity の略．水に対する比重を登録して，浮力などを反映できるようにした．
Elastic = 1.0e6 を変更すると，リンゴの剛性を変えられるが，硬いほど dt を細かくしなければ発散する

            elastic_SG=1.0,               # elastic specific gravity
            Eelastic = 1.0e6,             # scaled Young's modulus for elastic
            nuelastic = 0.3,              # Poisson's ratio for elastic

            snow_SG=0.5,                  # snow specific gravity
            Esnow = 1.0e6,                # scaled Young's modulus for elastic
            nusnow = 0.2,                 # Poisson's ratio for elastic

            sand_SG=2.5,          # 2.5,          # sand specific gravity
            Esand = 1.0e6,                # scaled Young's modulus for elastic
            nusand = 0.2,                 # Poisson's ratio for elastic

            elastic1_SG=1.2,              # elastic specific gravity
            Eelastic1 = 1.0e6,            # scaled Young's modulus for elastic
            nuelastic1 = 0.3,             # Poisson's ratio for elastic

            elastic2_SG=1.2,              # elastic specific gravity
            Eelastic2 = 1.0e6,            # scaled Young's modulus for elastic
            nuelastic2 = 0.3,             # Poisson's ratio for elastic

            support_plasticity=True,      # Support snow and sand materials
            use_adaptive_dt=False,
            use_ggui=False,
            use_emitter_id=False
    ):
        self.dim = len(res)
```

これがメインコードから呼び出される関数.

```
    def step(self, frame_dt, motion, nearestx, inside_pid, print_stat=False, smry_writer=None):
        begin_t = time.time()
        begin_substep = self.total_substeps

        substeps = int(frame_dt / self.default_dt) + 1

        dt = frame_dt / substeps
        frame_time_left = frame_dt
        if print_stat:
            print(f'needed substeps: {substeps}')
        while frame_time_left > 0:
            print('.', end='', flush=True)
            self.total_substeps += 1
            if self.use_adaptive_dt:
                max_grid_v = self.compute_max_grid_velocity(self.grid_v)
                cfl_dt = self.CFLC * self.dx / (max_grid_v + 1e-6)
                dt = min(dt, cfl_dt, frame_time_left)
            frame_time_left -= dt
            self.grid.deactivate_all()
            self.build_pid(self.pid, self.grid_m, 0.5)
            self.p2g(dt)                                    # particle to grid　粒子情報から格子情報を更新
            self.grid_normalization_and_gravity(dt, self.grid_v, self.grid_m)
            for p in self.grid_postprocess:
                p(self.t, dt, self.grid_v)
            self.t += dt
            self.g2pV(dt)                       # grid to particle 格子情報から粒子速度を更新

            if motion.flg:                      #  ナイフの力学に基づき，粒子情報を更新
                motion.dynamics( self.t, self.x, self.v, nearestx, inside_pid )

            self.g2pX(dt)                       # grid to particle  粒子の速度情報に基づき位置を更新

            cur_frame_velocity = self.compute_max_velocity()
            if smry_writer is not None:
                smry_writer.add_scalar("substep_max_CFL",  cur_frame_velocity  *  dt  /  self.dx,
self.total_substeps)
            self.all_time_max_velocity = max(self.all_time_max_velocity, cur_frame_velocity)
```

CAD の obj ファイルを trimesh を使って粒子情報に変換

```
    # ******** for CAD model ************
    def add_CADmodel(self, file_name, offset, scale=(1.0,1.0,1.0),
                     material=material_elastic,
                     color=0xFFFFFF,
                     sample_density=None,
                     velocity=None, size_ratio=0.5):
....
```

(3) 実行

Taichi 環境をアクティベイトして python で自身のディレクトリに移動し，メインコードを実行する.

```
> source taichi_enc/bin/activate
```

> cd ******

> python3 AppleCut.py

(4) Paraview による可視化

CAD のファイル（knife）の obj のメッシュを時系列で stl として保存し，粒子は VTU として時系列で保存してある. 応力など内部情報を可視化するときには格子データも保存すべき. とりあえず今は CG. なお，obj ファイルは時系列で読めないという paraview の恐ろしい仕様により，わざわざ stl に変更している．obj はテキストファイルのみ．stl はテキストとバイナリを選択出るので，バイナリで保存してある．ナイフはそのまま stl を出力．リンゴはテクスチャを張り付けることもできるが，リンゴが切れた時（トポロジーが変化したとき）にも平滑化してテクスチャをはるので切れているように見えない．ということで，リンゴは Gaussian resampling と contour で表示．詳細は，研究室 HP を参照.