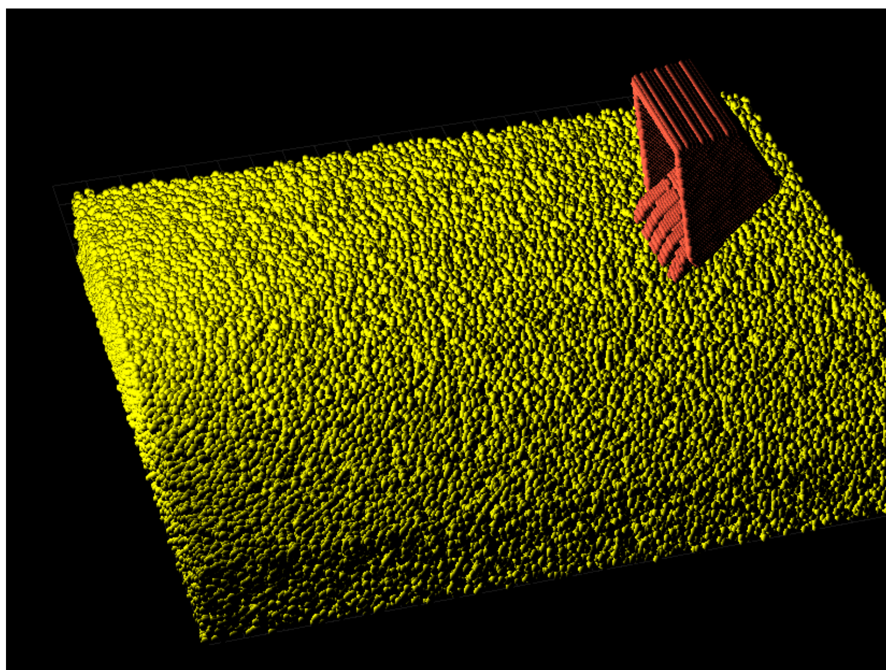
Excavation by MLS-MPM(Moving-Least Squares Material Point Method)

17 Feb. 2026

Taichi の MPM で, ショベルで土を掘る一方向連成シミュレーションをしてみる. 土は MPM の material の砂（sand），ショベルのバケットは material の弾性体（elastic）. 弾性体は CAD モデルで，運動学により外力の影響を受けずに動くので実質剛体.

計算範囲は 4m*3.2m*4m
掘削範囲は 4m*3.2m*深さ 0.5m（grid は 4m/64=6.25cm）
バケットは，1/10 モデルを 10 倍（grid は砂の 1/4）
30 秒の掘削＆排土作業を 2 サイクル（サイクル＝移動，掘削，排土，原点復帰の 4 工程）



(1) インストール
以下を参考に，mlsmpm がインストールされていることが前提.
https://www.kikulab.chibatech.ac.jp/wordpress/wp-content/uploads/2026/02/MPM.pdf

このファイル群の以下を書き換えた.

engine/mpm_solver.py	# CAD の obj ファイルから粒子を生成できるようにした
VTU.py	# paraview 用に VTU ファイルを吐き出すようにした.
motion.py	# ショベルの運動定義
mlsmpm08.py	# メインプログラム

(2) CAD モデル
某共同研究先の 1/10 ショベル．CAD データを obj として保存しておく.
001bucket.obj

粒子径と肉厚の関係は検討する必要がある．一般論として粒子系のシミュレーションは，粒子二つ以上じゃないと衝突の際，突き抜ける．MPM がそうかどうかは要検討．MPM の粒子同士はそもそもぶつかった状態になっている．感覚的には，計算の dt が粗いと土粒子がショベル粒子を突き抜ける（砂がバケットからすり抜けて落ちる）．

(3) ファイルの作成

以下二つは，魚の方を参照.

https://www.kikulab.chibatech.ac.jp/wordpress/wp-content/uploads/2026/02/fishMPM.pdf

. mpm_solver.py
. VTU.py

ショベルの運動は以下の通り．ただの回転と平行移動で移動，掘削，排土作業を 1 サイクルとして行っている．

```
import taichi as ti              # taichi を ti としてインポート
import math                      # 数学ライブラリをインポート

@ti.data_oriented                # ti データ関数として Motionclass を定義
class Motionclass:               # クラス名
    def __init__(self, bucket, mpm: ti.template(), dt: ti.f32):
        # bucket がメインから送られるバケットのデータリスト，mpm は粒子変数，dt は時間刻み
        self.bucketCenter = bucket["translation"]               # バケット初期位置
        self.scale = bucket["scale"]                            # バケットサイズ，10 倍に
        self.cycle = bucket["cycle"]            # [s]           # 1 サイクルの時間
        self.sideStep = bucket["sideStep"]   # [m] y-direction offset intercal # 各サイクルの移動距離
        self.xLocomotion =   bucket["xLocomotion"] # [m]    # x 方向前進距離
        self.zLocomotion =   bucket["zLocomotion"] # [m]    # z 方向距離
        self.relax =   bucket["relax"]                  # relaxation time  # 緩和時間：使ってない
        self.dt = dt                                    # 時間刻み
```

1 サイクル 30 秒で 4m 動いたとして，0.13m/s．dt が 1ms だとすると dt 間に 0.13mm しか動かないので，粒子径 6.25cm の 0.2%しか動いていないので十分なはず.

```
        self.bucket_particle_num = 0;                   # count elastic material number
        for i in range(mpm.x.shape[0]):                 # 弾性体（バケット）粒子のみカウント
            if mpm.material[i] == mpm.material_elastic:
                self.bucket_particle_num += 1;

        # バケット粒子初期位置＆番号保存
        self.bucketX0 = ti.Vector.field(3, ti.f32, shape=self.bucket_particle_num)
        self.particleID = ti.field(dtype=ti.i32, shape=self.bucket_particle_num)
        j = 0
        for i in range(mpm.x.shape[0]):
            if mpm.material[i] == mpm.material_elastic:
                self.bucketX0[j] = mpm.x[i]
                self.particleID[j] = i
                j += 1

    @ti.kernel     # *** One way FSI ***   ti.func を呼ぶために kernel にする.
    def excavation( self, mpm: ti.template(), tt: ti.f32 ):           # 掘削関数
        PI = ti.math.pi                                               # π
        center = self.bucketCenter                                    # いろいろ変数保存
        xradius = self.xLocomotion
        zradius = self.zLocomotion
```

```
        cycle = self.cycle  # [s]
        subcycle = cycle/4.0          # [s]                          # cyubcycle＝工程が 4 回で 1cycle
        sideStep = self.sideStep      # [m]                          # cycle 毎平行移動

        cycleID = int(tt/cycle)
        offsetY = float(cycleID) * sideStep
        t = tt - cycleID* cycle                              #  実時間 tt を，各サイクルの時間 t に変換

        #  1 サイクルは 4 工程
        # posture control  姿勢制御
        pitch0 =-PI / 4.0
        if t < subcycle:                       # start point to excavation point
            pitch0 = -PI/4.0
        elif t < ( 2.0*subcycle ):             # excavation and move to loading point
            pitch0 = -PI/4.0 + PI/2.0 * (t-subcycle)/subcycle
        elif t < ( 3.0*subcycle ):             # loading
            pitch0 = PI/4.0 - 3.0*PI/4.0 * (t-2.0*subcycle)/subcycle
        else:                                          # back to start point
            pitch0 = -PI/2.0 + 2.0 * PI/4.0 * (t-3.0*subcycle)/subcycle
            if -PI/4.0 < pitch0: pitch0 = -PI/4.0

        pitch1 =-PI / 4.0
        if (t+self.dt) < subcycle:             # start point to excavation point
            pitch1 = -PI/4.0
        elif (t+self.dt) < ( 2.0*subcycle ):   # excavation and move to loading point
            pitch1 = -PI/4.0 + PI/2.0 * ((t+self.dt)-subcycle)/subcycle
        elif (t+self.dt) < ( 3.0*subcycle ):   # loading
            pitch1 = PI/4.0 - 3.0*PI/4.0 * ((t+self.dt)-2.0*subcycle)/subcycle
        else:                                          # back to start point
            pitch1 = -PI/2.0 + 2.0 * PI/4 * ((t+self.dt)-3.0*subcycle)/subcycle
            if   -PI/4.0 < pitch1: pitch1 = -PI/4.0

        Ry0 = ti.Matrix([ [ ti.cos(pitch0), 0.0, ti.sin(pitch0)],
                          [ 0.0,            1.0, 0.0           ],
                          [-ti.sin(pitch0), 0.0, ti.cos(pitch0)] ])
        Ry1 = ti.Matrix([ [ ti.cos(pitch1), 0.0, ti.sin(pitch1)],
                          [ 0.0,            1.0, 0.0           ],
                          [-ti.sin(pitch1), 0.0, ti.cos(pitch1)] ])

        # locomotion control  移動制御
        displacement0 = ti.Vector([0.0,0.0,0.0])
        theta0 = 0.0
        if t < subcycle:                       # start point to excavation point
            theta0 =-PI/2.0 * t/subcycle
            displacement0 = [-xradius + xradius * ti.math.cos(theta0), offsetY, zradius *
ti.math.sin(theta0)]
        elif t < ( 2.0*subcycle ):             # loading
            theta0 =-PI/2.0 - PI/2.0 * (t-subcycle)/subcycle
            displacement0 = [-xradius + xradius * ti.math.cos(theta0), offsetY, zradius *
ti.math.sin(theta0)]
        elif t < ( 3.0*subcycle ):             # loading
            displacement0 = [-2.0*xradius, offsetY, 0.0]
        else:
            displacement0 = [-2.0*xradius+2.0*xradius*(t-3.0*subcycle)/subcycle,
offsetY+sideStep*(t-3.0*subcycle)/subcycle, 0.0]

        displacement1 = ti.Vector([0.0,0.0,0.0])
        theta1 = 0.0
        if (t+self.dt) < subcycle:                     # start point to excavation point
            theta1 =-PI/2.0 * (t+self.dt)/subcycle
```

```
                displacement1 = [-xradius + xradius * ti.math.cos(theta1), offsetY, zradius *
ti.math.sin(theta1)]
            elif (t+self.dt) < ( 2.0*subcycle ):                # loading
                theta1 =-PI/2.0 - PI/2.0 * ((t+self.dt)-subcycle)/subcycle
                displacement1 = [-xradius + xradius * ti.math.cos(theta1), offsetY, zradius *
ti.math.sin(theta1)]
            elif (t+self.dt) < ( 3.0*subcycle ):                # loading
                displacement1 = [-2.0*xradius, offsetY, 0.0]
            else:
                displacement1 = [-2.0*xradius+2.0*xradius*((t+self.dt)-3.0*subcycle)/subcycle,
offsetY+sideStep*((t+self.dt)-3.0*subcycle)/subcycle, 0.0]

            for i in range(self.bucket_particle_num):       # バケット粒子位置更新
                pos = ( self.bucketX0[i] - center )
                p0 = Ry0 @ pos + displacement0 + center
                p1 = Ry1 @ pos + displacement1 + center
                mpm.x[ self.particleID[i] ] = p0                    # ここで実際の粒子位置書き換え
                mpm.v[ self.particleID[i] ] = ( p1 - p0 ) / self.dt  # 速度書き換え
```

. mlsmpm08.py
実行ファイル

```
import taichi as ti
import numpy as np                  # number 関連の py を np としてインポート
import utils                        # これなんだっけ？
import os                           # linux コマンドをつかう．mkdir 用
import glob                         #

from moviepy import ImageSequenceClip      # 動画

from engine.mpm_solver import MPMSolver # engine/mpm_solver.py のファイルの MPMSolver クラスをインポート
from VTU import VTUclass                     # VTU 用
from motion import Motionclass               # Motionclass 用

# Parameters
SimulationTime = 60        # [s]    シミュレーション時間
CalculationSpace = 4       # [m]    計算空間 4*4*4m^3 この後変更
Resolution = 64            # 粒子解像度，4m/64=6.25cm 程度，DEM が 10cm なのでそれよりちょっと小さいぐらい
dt = 2e-4                  # [s]計算刻み．粗いと砂粒子がバケットを突き抜ける．
fileName = "001bucket.obj"          # CAD ファイル
VTUinterval = 100                   # VTU saving interval:  png と VTU の保存間隔
dt*VTUinterval=2ms 毎

bucket = {                          # バケットパラメータ
        "translation": [3.2, 1, 1.4], # 配置位置
        "scale": [10.0, 10.0, 10.0], # 倍率．1/10 モデルなので x, y, z 方向に 10 倍
        "cycle": 30.0,              # [s]：  1 サイクル時間＝4 工程
        "sideStep": 1.0,    # [m] y-direction offset intercal  並進距離
        "xLocomotion": 1.1,         # [m]  掘削方向移動量
        "zLocomotion": 0.9,         # [m]  上下方向移動量
        "relax": 0.1                # 緩和係数，使ってない
        }

ti.init(arch=ti.cpu)                # GPU も Vulcan も使わず昭和漢は CPU で！

# 粒子化．グリット化．引数は，
解像度 vector：これは実際の計算空間を決めている
計算空間サイズ(size)：粒子径 dx=size/reso[0]を決めるのに使っているだけ，
```

```
粒子数条件: 2^30 乗まで設定できるようになっているが，このプログラムで 1G なので実質ムリ．
mpm = MPMSolver(res=(Resolution, (int)(0.8*Resolution), Resolution), size=CalculationSpace,
max_num_particles=2 ** 19, use_ggui=True)

# 砂粒子の作成．引数は，
立方体の一つの頂点 vector：
立方体のサイズ vector：
材質：砂．他には，雪，水，静止体，弾性体がある．
mpm.add_cube(lower_corner=[0, 0, 0], cube_size=[4, 3.2, 0.5], material=MPMSolver.material_sand)

# CAD モデルを読み込んで連続体の粒子にする．詳細は，上記ファイル参照．
size_ratio=0.5 は粒子が常に半分ぶつかっている状態．これだと砂粒子が突き抜けるので 0.25 にして
ある．感覚的には，0.5 でバケットの肉厚に対して粒子 2 個．0.25 でバケット肉厚に対して粒子 4 個．
mpm.add_CADmodel(fileName, offset=bucket["translation"], scale=bucket["scale"],
material=MPMSolver.material_elastic, size_ratio=0.25)

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

@ti.kernel                     # 色設定
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, 1.0])
        for d in ti.static(range(3)):
            color_4d[d] = material_color[material_id, d]
        ti_color[I] = color_4d

res = (1024, 760)              # ディスプレイ設定
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(-2, -10, 10)         # カメラ位置
camera.lookat(1, 1, 1)               # 視点
camera.up(0, 0, 1)                   # 画面方向
camera.fov(16)                       # 視野範囲
particles_radius = CalculationSpace / Resolution / 4.0

# ground lines                       # 地面の描画
N_ground = 20
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():                        # 粒子の描画
    camera.track_user_inputs(window, movement_speed=0.03, hold_key=ti.ui.RMB)
    scene.set_camera(camera)
    scene.ambient_light((0, 0, 0))
    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():                    # GUI. 使ってない
    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([   # 材料の色設定：RGB+透明度
    [0.1, 0.1, 1.0, 0.8],                                           # 水
    [236.0 / 255.0, 84.0 / 255.0, 59.0 / 255.0, 1.0],               # 弾性体＝ショベル
    [1.0, 1.0, 1.0, 1.0],                                           # 雪
    [1.0, 1.0, 0.0, 1.0] ] )                                        # 砂

os.makedirs("image", exist_ok=True)                                 # ディレクトリ作成
# os.makedirs("VTU", exist_ok=True)

vtu = VTUclass()                                                    # 使ってない.
motion = Motionclass(bucket, mpm, dt)                               # 運動クラス

t = 0.0
cnt = 0
cnt2 = 0

while t<SimulationTime:
    print(f"time={t:.3f}")
    motion.excavation(mpm, t)                                       # これが運動計算
    mpm.step(dt)                                                    # 粒子のなーんちゃって動力学

    if cnt%VTUinterval == 0:                                        # 描画関連
        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) 実行

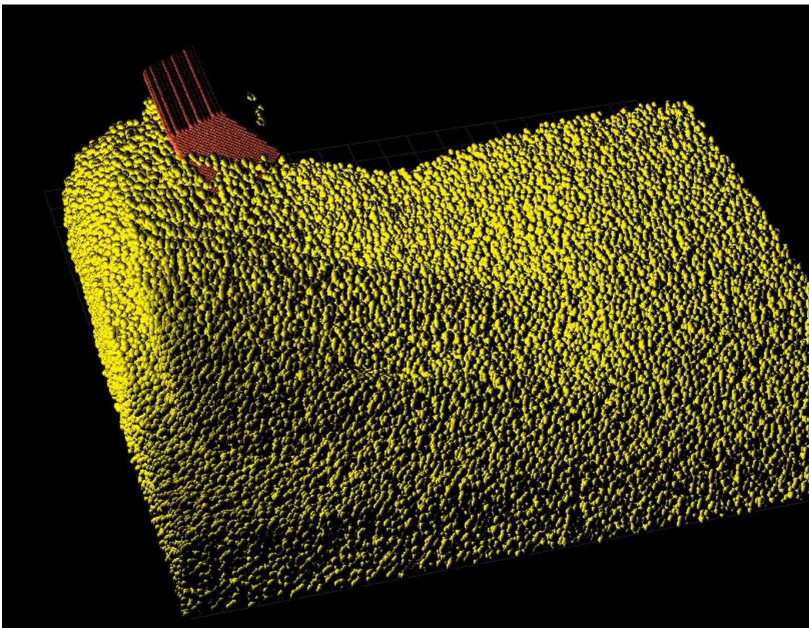
taichi を activate して実行

```
> source taichi_env/bin/activate
> python3 mlsmpm08.py
```

数時間かかる．粒子径は，6.25cm であるが，倍重なっている．よって，計算空間が，4*3.2*0.5m^3 だとして，4*3.2*0.5/(0.0625^3)*8=0.2M 個ぐらいある．位置ベクトルの他，力ベクトル，速度ベクトル，色（これもベクトルか...），グリッド，材料もあるとして，おそらく 1.0G ぐらい．しかし，実際は，タスクマネージャーをみると 2G 使っている．描画を辞めればあるいは半分になるのかも．

施工現場が 8m*8m*1m で 8 倍として 16G ぐらいメモリがあれば計算できるか... いや，どうでしょう？

DEM を 10cm*10cm で切ったとして，0.1*0.1/(0.0625^2)*4=10 個あるので十分確率的にも精度は保てそう．



Paraview による可視化は，別資料参照．なお，時系列はただただでかい．ご注意を．

(5) 資料

print(dir(mpm))で定義されている mpm のメンバ変数を見ると以下の通り．SPH と違って，初期値は保存していないのでメモリ的には少し小さいのかも．

['C', 'E', 'F', 'F_bound', 'Jp', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_data_oriented', '_seed_from_ndarray', 'add_CADmodel', 'add_bounding_box', 'add_cube', 'add_ellipsoid', 'add_mesh', 'add_ngon', 'add_particles', 'add_sphere_collider', 'add_surface_collider', 'add_texture_2d', 'all_time_max_velocity', 'alpha', 'block', 'block_offset', 'build_pid', 'clear_grid_postprocess', 'clear_particles', 'color', 'color_with_alpha', 'compute_max_grid_velocity', 'compute_max_velocity', 'copy_dynamic', 'copy_dynamic_nd', 'copy_ranged', 'copy_ranged_nd', 'default_dt', 'dim', 'dx', 'g2p', 'g2p2g', 'g2p2g_allowed_cfl', 'gravity', 'grid', 'grid_bounding_box', 'grid_m', 'grid_normalization_and_gravity', 'grid_postprocess', 'grid_size', 'grid_v', 'input_grid', 'inv_dx', 'lambda_0', 'last_time_final_particles', 'leaf_block_size', 'material', 'material_elastic', 'material_sand', 'material_snow', 'material_stationary', 'material_water', 'materials', 'max_num_particles', 'mu_0', 'n_particles', 'nu', 'num_grids', 'offset', 'p2g', 'p_mass', 'p_rho', 'p_vol', 'padding', 'particle', 'particle_info', 'pid', 'quant', 'random_point_in_unit_polygon', 'random_point_in_unit_sphere', 'read_restart', 'recover_from_external_array', 'res', 'sand_projection', 'seed', 'seed_ellipsoid', 'seed_from_external_array', 'seed_from_voxels', 'seed_particle', 'seed_polygon', 'set_gravity', 'set_source_velocity', 'source_bound', 'source_velocity', 'stencil_range', 'step', 'support_plasticity', 'surface_separate', 'surface_slip', 'surface_sticky', 'surfaces', 't', 'total_substeps', 'unbounded', 'use_adaptive_dt', 'use_bls', 'use_emitter_id', 'use_g2p2g', 'use_ggui',
'v',
'v_clamp_g2p2g', 'voxelizer', 'voxelizer_super_sample', 'water_density', 'write_particles', 'write_particles_ply', 'writers',
'x'
]

力学関係覚書：

. mpm_solver.py
に力学計算が書かれているが，解析はしていない．材料の物性値に関しては SPH と同様，サイズでスケーリングされている．SI 単位系はほぼ無意味．また，密度，質量といった概念がない．砂が水に沈まない．いろいろ修正しなければいけないところがあるが，リアルにすればするほど計算時間がかかり，Taichi の良さがなくなってくるのでどうするか考えもの．この後，mujoco などの剛体シミュレータとの連成が必要になるが，MPM で完結させる手もある．
砂と連続体のパラメータ関係一部抜粋：

```
        # Young's modulus and Poisson's ratio
        self.E, self.nu = 1e6 * size * E_scale, 0.2
```

$E = 1.0*10^6 * size * Escale$ # size = 計算空間のサイズ, E_scale=1

$\nu=0.2$ # 普通は 0.3 あたり

Lame parameters　弾性変形で，圧縮方向とせん断方向を両方考えるやつ

self.mu_0, self.lambda_0 = self.E / (2 * (1 + self.nu)), self.E * self.nu / ((1 + self.nu) * (1 - 2 * self.nu))

$\mu = \frac{E}{2(1+\nu)}$　　　# ラメの第二定数で横弾性係数 G の式と同じ．等方性じゃないと変えたりするようだ．Lamé parameters は以下

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}$$

In homogeneous and isotropic materials, these define Hooke's law in 3D,

$$\boldsymbol{\sigma} = 2\mu\boldsymbol{\varepsilon} + \lambda\,\mathrm{tr}(\boldsymbol{\varepsilon})I,$$

where $\boldsymbol{\sigma}$ is the stress tensor, $\boldsymbol{\varepsilon}$ the strain tensor, I the identity matrix, and tr the trace function. Hooke's law may be written in terms of tensor components using index notation as

$$\sigma_{ij} = 2\mu\varepsilon_{ij} + \lambda\delta_{ij}\varepsilon_{kk},$$

where δ_{ij} is the Kronecker delta.

https://en.wikipedia.org/wiki/Lam%C3%A9_parameters

https://ja.wikipedia.org/wiki/%E3%83%A9%E3%83%A1%E5%AE%9A%E6%95%B0

# Sand parameters
friction_angle = math.radians(45)
sin_phi = math.sin(friction_angle)
self.alpha = math.sqrt(2 / 3) * 2 * sin_phi / (3 - sin_phi)

$$\alpha = 2\sqrt{\frac{2}{3}}\frac{\sin\frac{\pi}{4}}{3-\sin\frac{\pi}{4}} = 0.5036$$

．chaty とのやりとりでの格言

ti.field は「一度だけ作る」もの．呼び出している関数で何度も実行するとメモリが足りなくなる．ガーベージコレクト（gc.collect()）してもダメ

core dump は，ubuntu で off にしても wsl が吐く．しかも以下に．

C:¥Users¥kouki¥AppData¥Local¥Temp¥wsl-crashes

プリントデバックは今でも有効．昭和は偉大なり．

print("check", flush=True)