Taichi 1.7.4

16, Feb, 2026

Genesis 0.3.4 だとそれぞれのスキームの連成がままならず，Secure なコードでプログラミングもままならないので Genesis のおおもとになっている Taichi のプラットフォームを試してみる．引っ張ったらちぎれるような帯を水面にたたき付けて水がはじけ飛ぶシミュレーション（気液二相流＋弾塑性＆破壊力学の双方向連成）を正しく計算したい人向け．OS は Linux（Ubuntu24 以降）で．GPU がいいけど CPU で．

公式 HP:
https://docs.taichi-lang.org/

User Interface:
https://docs.taichi-lang.org/api/taichi/ui/

API:
https://docs.taichi-lang.org/api/

Genesis と Taichi の関係
基本概念は同じ．Genesis は CG 向け．API が充実している．Taichi は API が充実していないが，genesis が secure にしているコードも自由に書ける．例えば，genesis は計算途中で重力を変更できない．おそらく const になっている．ただ，version 0.4 系で大幅改善するらしい．

主な計算スキームは以下：
MPM (Material Point Method) and SPH (Smoothed Particle Hydrodynamics)
GPU ベースなので粒子系で．粒子法ベースなので力学的正確さは全く期待できず．

(1) 実行環境＆インストール
．bash 環境で以下を実行して環境を作成
> python3 -m venv taichi_env

現在 python3.12.3 だが，python3.11 系の方が安定しているらしい．以下で使っている version が分かる．
> python3 --version
3.11 がインストールしてあれば，
> python3.11 実行ファイル
で実行できる．以下で確認できる．
> python3.11 –version
が，基本，ubuntu の version に対応した python を使うことを推奨しているようだ．なお，venv が無ければ，以下でインストール．

> sudo apt install python3.**-venv

以下を実行して，taichi を activate する．以下，このプロンプトから python コードを実行することになる．

> source taichi_env/bin/activate

以下，pip（Pip Installs Packages: Python のパッケージマネージャ）を使って必要パッケージをインストール

．ここから taichi の環境．まず，pip を upgrade する．現在 pip-26.0.1

> pip install --upgrade pip

taichi をインストール

> pip install taichi

taichi が呼び出す c のコードをメイク．

> python -c "import taichi as ti; print(ti.__version__)"

以下が表示される．

[Taichi] version 1.7.4, llvm 15.0.4, commit b4b956fd, linux, python 3.12.3

(1, 7, 4)

なお，c（g++, gcc）の version もかなり影響する．現在 version 12 を使っている．菊池研 FSI を使う openFOAM を使っている場合，version9 を使っているはずなので注意する．以下で version をチェックできる．

> g++ --version

> gcc --version

描画ライブラリも入れておく

> pip install matplotlib

> pip install imageio imageio-ffmpeg

> pip install tqdm

> pip install open3d

> pip install moviepy

mujoco を使うなら以下もインストールする．

> pip install mujoco-py taichi

CAD（メッシュ）関係のファイルを扱うときには以下もインストールする．

> pip install trimesh

> pip install networkx

点群処理をするなら，以下も入れる．

> pip install plyfile

VTU ファイルを使うなら以下も入れる．

> pip install meshio

(2) サンプルコードを実行

. test.py

```
import taichi as ti

ti.init(arch=ti.cpu)

# スカラー場
x = ti.field(dtype=ti.f32, shape=())

@ti.kernel
def step():
    x[None] += 1.0

for i in range(5):
    step()
    print(f"step {i}: x =", x[None])
```

. bash 環境で taichi を activate して実行.

> source taichi_env/bin/activate

> python3 test.py

```
(taichi_env) kikut@kikut3:~/taichi$ python3 test.py
[Taichi] version 1.7.4, llvm 15.0.4, commit b4b956fd, linux, python 3.12.3
[Taichi] Starting on arch=x64
step 0: x = 1.0
step 1: x = 2.0
step 2: x = 3.0
step 3: x = 4.0
step 4: x = 5.0
```

(3) グラフィックサンプルコードを実行

https://docs.taichi-lang.org/docs/hello_world

. fractal.py

```
import taichi as ti
import taichi.math as tm

ti.init(arch=ti.gpu)

n = 320
pixels = ti.field(dtype=float, shape=(n * 2, n))

@ti.func
def complex_sqr(z):  # complex square of a 2D vector
    return tm.vec2(z[0] * z[0] - z[1] * z[1], 2 * z[0] * z[1])

@ti.kernel
def paint(t: float):
    for i, j in pixels:  # Parallelized over all pixels
        c = tm.vec2(-0.8, tm.cos(t) * 0.2)
        z = tm.vec2(i / n - 1, j / n - 0.5) * 2
```

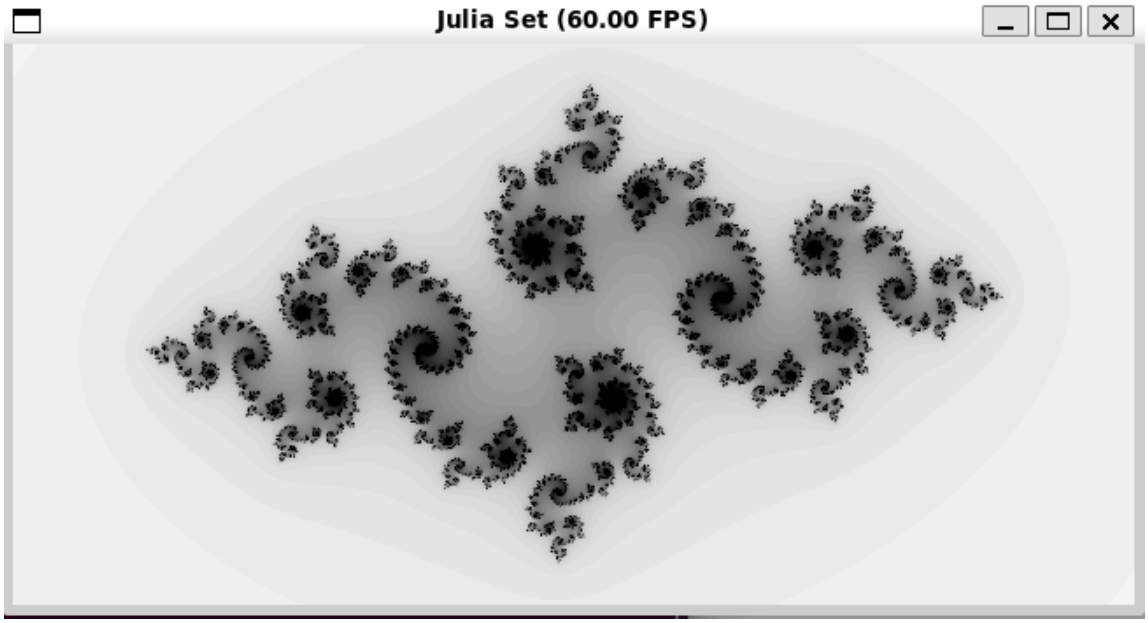
```
            iterations = 0
            while z.norm() < 20 and iterations < 50:
                z = complex_sqr(z) + c
                iterations += 1
            pixels[i, j] = 1 - iterations * 0.02

gui = ti.GUI("Julia Set", res=(n * 2, n))

i = 0
while gui.running:
    paint(i * 0.03)
    gui.set_image(pixels)
    gui.show()
    i += 1
```

実行すると以下の通り

> python3 fractal.py



Taishi 関連覚書

．Vulkan を使っている関数を強引に CPU で描かせるくだりは，ubuntu24.04 系以上でないとできないかも．ubuntu22.04 系で頑張ったができなかった．

1. `@ti.data_oriented`

対象：クラス

Taichi のクラスを「Taichi対応クラス」にするためのデコレータです。

```python
@ti.data_oriented
class A:
    ...
```

これを付けると：

- クラス内で `@ti.kernel` や `@ti.func` を使える
- `self.field` のような Taichi field にアクセスできる
- Taichi の JIT コンパイル対象になる

2. `@ti.kernel`

対象：関数

Taichi の**実行単位（カーネル）**です。

```python
@ti.kernel
def step():
    ...
```

特徴：

- Python から呼び出せる
- JIT コンパイルされる
- 並列実行される
- Taichi のフィールドにアクセス可能
- 引数と戻り値には型指定が必要（例：`ti.f32`）

例：

```python
@ti.kernel
def add():
    for i in x:
        x[i] += 1
```

3. `@ti.func`

対象：関数

カーネル内で使う**インライン関数**です。

```python
@ti.func
def myfunc(a):
    return a * a
```

特徴：

- **Python から直接呼べない**
- `@ti.kernel` の中からのみ呼べる
- インライン展開される
- 追加のカーネル起動はしない

例：

```python
@ti.func
def square(a):
    return a * a

@ti.kernel
def compute():
    x[0] = square(3.0)
```

配列数をあとから設定する方法

方法2：SNode で明示的に配置（MPMなどでよく使う）

```python
self.n = 1000
self.x = ti.Vector.field(self.dim, ti.f32)

ti.root.dense(ti.i, self.n).place(self.x)
```

これで1次元配列として `self.n` 個確保されます。

3次元グリッドなら：

```python
ti.root.dense(ti.ijk, (nx, ny, nz)).place(self.x)
```

なお，dense = dense array allocation

```
grid = ti.root.pointer(ti.ij, (64, 64))
grid.dense(ti.ij, (8, 8)).place(x)
```

これは：

- 64×64 個のブロックを持つ
- 各ブロックは 8×8 の dense 配列
- 実際に使われたブロックだけメモリ確保

なぜ使うか

MPM や SPH では

- 空間全体は大きい
- 粒子が存在する場所は局所的

なので

→ 空間全体を dense にするとメモリが無駄
→ pointer を使って **必要なセルだけ確保**
