PyAFV Documentation

Release 0.4.15

Wei Wang

Jun 09, 2026

CONTENTS

1 Getting started 3
1.1 Installation . . . 3
1.1.1 Install using pip/conda . . . 3
1.1.2 Install from source . . . 4
Required prerequisites . . . 4
Windows MinGW GCC . . . 4
1.1.3 Install offline . . . 4
1.1.4 Install using Docker . . . 4
1.2 A simple example . . . 5

2 Examples 9
2.1 Relaxation to mechanical equilibrium . . . 9
2.2 Active finite Voronoi (AFV) dynamics . . . 11
2.3 Connectivity between cells . . . 13
2.4 Custom plotting . . . 14
2.4.1 A standalone plotting utility . . . 17
2.5 Periodic boundary conditions . . . 19
2.6 Varying cell target areas from cell to cell . . . 20

3 Multiprocessing 23
3.1 Basic usage . . . 23
3.2 Repeated build steps . . . 24
3.3 Halo width . . . 25
3.4 Decomposition method . . . 25
3.5 Visualization . . . 25
3.6 A complete example . . . 25
3.7 Running on clusters . . . 26
3.7.1 Running on a single node . . . 26
3.7.2 Multi-node parallelism: MPI . . . 26

4 Performance 31
4.1 Measuring performance . . . 31
4.2 Benchmarking backends . . . 31
4.3 Benchmarking parallel build . . . 32
4.4 Benchmarking hybrid parallel build: MPI + Python multiprocessing . . . 33

5 Calibration 37
5.1 How to calibrate against the DP model . . . 37
5.2 Usage of the DP simulator . . . 38

6 Citing PyAFV 41

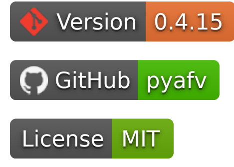
7 Contributing to PyAFV 43
- 7.1 Code of Conduct 43
- 7.2 How to contribute 43
  - 7.2.1 Create a fork 43
  - 7.2.2 Set up your development environment 43
  - 7.2.3 Create a feature branch and start development 44
  - 7.2.4 Keeping your fork up to date 44
- 7.3 Coding standards 45
- 7.4 Documentation requirements 45
- 7.5 Writing tests 45
  - 7.5.1 Testing strategies (in order of preference) 46
  - 7.5.2 Benchmarking 46
- 7.6 Featured examples 46
- 7.7 Submitting a pull request 47
  - 7.7.1 Pull request process 47
- 7.8 Reporting issues 47
- 7.9 Code review process 48
- 7.10 Questions? 48

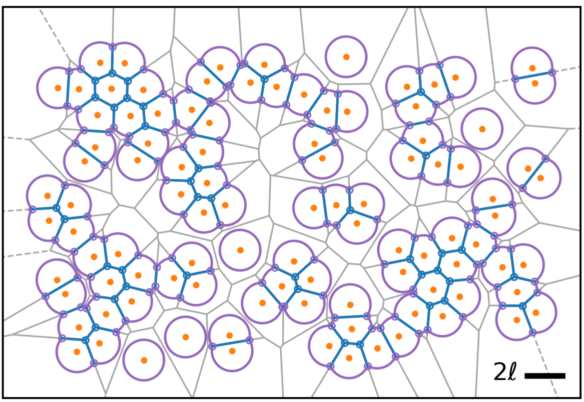
8 API reference 49
- 8.1 pyafv 49
- 8.2 pyafv.PhysicalParams 50
  - 8.2.1 pyafv.PhysicalParams.get_steady_state 51
  - 8.2.2 pyafv.PhysicalParams.with_optimal_radius 51
  - 8.2.3 pyafv.PhysicalParams.replace 52
- 8.3 pyafv.FiniteVoronoiSimulator 52
  - 8.3.1 pyafv.FiniteVoronoiSimulator.preferred_areas 53
  - 8.3.2 pyafv.FiniteVoronoiSimulator.build 53
  - 8.3.3 pyafv.FiniteVoronoiSimulator.plot_2d 54
  - 8.3.4 pyafv.FiniteVoronoiSimulator.update_params 54
  - 8.3.5 pyafv.FiniteVoronoiSimulator.update_positions 55
  - 8.3.6 pyafv.FiniteVoronoiSimulator.update_preferred_areas 55
  - 8.3.7 pyafv.FiniteVoronoiSimulator._build_voronoi_with_extensions 55
  - 8.3.8 pyafv.FiniteVoronoiSimulator._per_cell_geometry 56
- 8.4 pyafv.ParallelFiniteVoronoiSimulator 57
  - 8.4.1 pyafv.ParallelFiniteVoronoiSimulator.preferred_areas 58
  - 8.4.2 pyafv.ParallelFiniteVoronoiSimulator.build 59
  - 8.4.3 pyafv.ParallelFiniteVoronoiSimulator.update_params 59
  - 8.4.4 pyafv.ParallelFiniteVoronoiSimulator.update_positions 60
  - 8.4.5 pyafv.ParallelFiniteVoronoiSimulator.update_preferred_areas 60
- 8.5 pyafv.visualize_2d 61
- 8.6 pyafv.visualize_2d_parallel 62
- 8.7 pyafv.target_delta 62
- 8.8 pyafv.calibrate 63
  - 8.8.1 pyafv.calibrate.auto_calibrate 63
  - 8.8.2 pyafv.calibrate.DeformablePolygonSimulator 64
  - 8.8.3 pyafv.calibrate tools 66
- 8.9 Experimental APIs 67

Bibliography 71

Python Module Index 73

Index 75





PyAFV is a Python implementation of the **active finite Voronoi (AFV) model** in 2D.

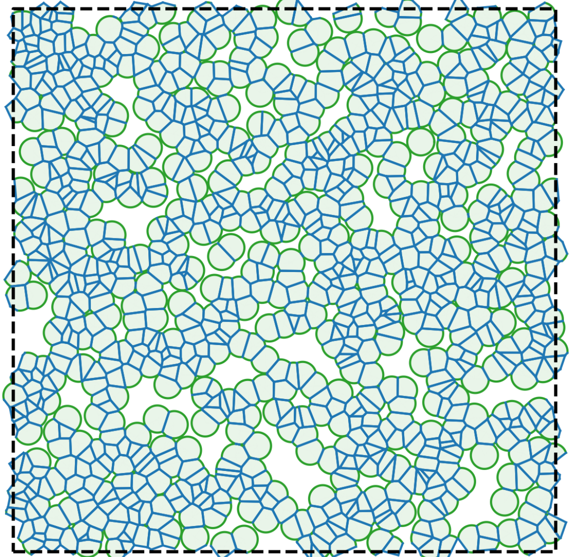
The AFV framework was introduced and developed in, for example, Refs. [1, 2, 3].

Check out the *Getting started*, *Examples*, *Multiprocessing*, *Performance*, *Calibration*, *Citing PyAFV*, *Contributing to PyAFV*, and *API reference* sections for further information, including how to *install* the package, usage examples, multi-process domain decomposition, benchmarks, local development, and the complete API reference.

- *Explore a collection of usage examples in Jupyter notebooks on Google Colab.*
- *See also an interactive simulation demo using PyAFV on Prof. Dapeng (Max) Bi's homepage!*

Note

For the latest updates, see the **GitHub** repository.



Contents

CHAPTER

ONE

GETTING STARTED

1.1 Installation

PyAFV supports *Python* >= 3.10, < 3.15, and has been tested on major operating systems including *Linux*, *macOS*, and *Windows*, for both *x86-64* and *ARM64* architectures.

Tests on all platforms passing

Note

- Python 3.14t, the **free-threaded** build that runs without the *Global Interpreter Lock (GIL)*, is also supported starting with **PyAFV** v0.3.8.
- Python 3.10 and 3.14t (free-threaded) on *Windows ARM64* (not *x86-64*) are the only untested configurations. The builds succeed and the wheels are available on **PyPI**, but automated testing is unavailable due to the absence of a supported **GitHub Actions** runners for these configurations.

1.1.1 Install using pip/conda

The package is available on **PyPI**, so you should be able to install it using *pip* directly:

```
(.venv) $ pip install pyafv
```

After installation, verify that it was successful by importing the package in *Python* and checking the version:

```
>>> import pyafv
>>> pyafv.__version__
'0.4.15'
```

Note

On some HPC clusters, global Python path can contaminate the runtime environment. You may need to clear it explicitly using `unset PYTHONPATH` or prefixing the *pip* command with `PYTHONPATH=""`.

As an alternative, you can install **PyAFV** via *conda* from the **conda-forge** channel:

```
(.venv) $ conda install -c conda-forge pyafv
```

If you go this route, note that for Python 3.14 the package currently supports only the GIL-enabled build.

1.1.2 Install from source

Installing from source can be necessary if *pip* installation does not work. First, download the source code of *pyafv* from **GitHub** or **PyPI**.

Required prerequisites

In general, you do not need to manually install the dependencies, as *pip* will handle them automatically. We list the required packages and minimum versions below for reference:

Package	Minimum version	Usage
numpy	1.26.4	Numerical computations
scipy	1.13.1	Miscellaneous scientific functions
matplotlib	3.8.4	Plotting and visualization

Unzip the downloaded source code and navigate to the root directory of the package. Then, run the following command to install:

```
(.venv) $ pip install .
```

Note

A **C/C++** compiler is required if you are building from source, since some components of **PyAFV** are implemented in **Cython** for performance optimization.

Windows MinGW GCC

If you are using **MinGW GCC** (rather than **MSVC**) on *Windows*, to build from the source code, add a `setup.cfg` file at the repository root before running `pip install .` with the following content:

```
# setup.cfg
[build_ext]
compiler=mingw32
```

1.1.3 Install offline

If you need to install **PyAFV** on a machine without internet access, you can download the corresponding wheel file from **PyPI** on another machine with internet access. Transfer the wheel file to the target machine, and then run the following command to install it using *pip* (make sure the required prerequisites listed above are already installed):

```
(.venv) $ pip install pyafv-<version>-<platform>.whl
```

Alternatively, you can build **PyAFV** from source as described in the previous section. In this case, in addition to the required prerequisites listed above, the build-time dependencies **hatchling** and **hatch-cython** must also be available.

1.1.4 Install using Docker

Pull the Docker image from **Docker Hub**:

```
(.venv) $ docker pull wwang721/pyafv:latest
```

It's also available in the **GitHub Container Registry (GHCR)** under **GitHub Packages**; use `ghcr.io/wwang721/pyafv` to pull from GHCR instead. Then run Python scripts with *pyafv* using:

```
(.venv) $ docker run --rm -v $(pwd):/app wwang721/pyafv python <script_name>.py
```

Use `${PWD}` on Windows PowerShell instead of `$(pwd)`.

1.2 A simple example

Now that you have installed **PyAFV**, here is a simple example to get you started (click the **Google Colab** badge above to run the notebook directly). Begin by importing the required libraries and generating 100 random points in two dimensions:

```
import numpy as np
import pyafv

N = 100                                             # number of cells
pts = np.random.rand(N, 2) * 10                     # initial positions
```

Next, create a *pyafv.PhysicalParams* object to specify the physical parameters of the simulation:

```
params = pyafv.PhysicalParams(r=1.0)                # use default parameter values
```

class PhysicalParams(*r=1.0*, *A0=3.141592653589793*, *P0=4.8*, *KA=1.0*, *KP=1.0*, *Lambda=0.2*, *delta=None*)

Physical parameters for the active finite Voronoi (AFV) model.

> **Warning**
>
> - **Frozen dataclass** is used for *PhysicalParams* to ensure immutability of instances.
> - Do not set `delta` unless you know what you are doing.

Parameters

- **r** (*float*) – Radius (maximal) of the Voronoi cells, sometimes denoted as ℓ.
- **A0** (*float*) – Preferred area of the Voronoi cells.
- **P0** (*float*) – Preferred perimeter of the Voronoi cells.
- **KA** (*float*) – Area elasticity constant.
- **KP** (*float*) – Perimeter elasticity constant.
- **Lambda** (*float*) – Tension difference between non-contacting edges and contacting edges.
- **delta** (*float | None*) – Contact truncation threshold to avoid singularities in computations; if None, set to 0.45*r.

Finally, initialize the simulator by constructing a *pyafv.FiniteVoronoiSimulator* instance and visualize the resulting Voronoi diagram:

```
sim = pyafv.FiniteVoronoiSimulator(pts, params)     # initialize the simulator
sim.plot_2d(show=True)                              # visualize the Voronoi diagram
```

The plotting routine `plot_2d()` is provided by:

`FiniteVoronoiSimulator.plot_2d(ax=None, show=False, **kw)`

Build the finite Voronoi structure and render a 2D snapshot. The plotting style follows `scipy.spatial.voronoi_plot_2d()`.

This method is basically a wrapper of `_build_voronoi_with_extensions()` and `_per_cell_geometry()` functions + plot.

Parameters

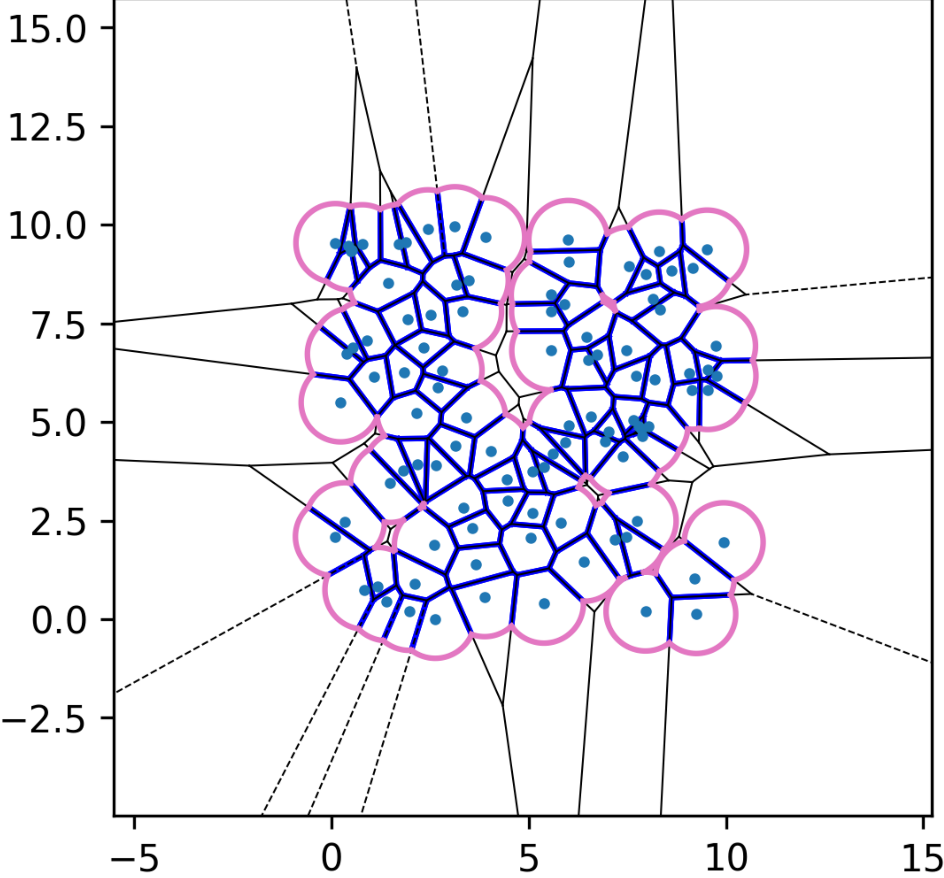
- **ax** (*Axes | None*) – If provided, draw into the axes; otherwise get the current axes.
- **show** (*bool*) – Whether to call `plt.show()` at the end.
- **show_points** (*bool, optional*) – Add cell center points to the plot, default *True*.
- **point_size** (*float, optional*) – Specifies the marker size for the points, default 2.
- **show_inner_vertices** (*bool, optional*) – Add inner vertices to the plot, default *False*.
- **show_outer_vertices** (*bool, optional*) – Add outer vertices to the plot, default *False*.
- **line_color_in** (*str, optional*) – Specifies the color for contact edges (and inner vertices), default 'b'.
- **line_color_out** (*str, optional*) – Specifies the color for non-contact edges (and outer vertices), default 'C6'.
- **line_width** (*float, optional*) – Specifies the line width for cell boundaries, default 1.5.
- **line_alpha** (*float, optional*) – Specifies the line alpha for cell boundaries, default 1.0.
- **show_voronoi** (*bool, optional*) – Add the Voronoi edges to the plot, default *True*.

Returns

The matplotlib axes containing the plot.

Return type

Axes



To compute the conservative forces and extract detailed geometric information (e.g., cell areas, vertices, and edges), call:

```
diag = sim.build()                                          # compute forces and geometry
```

The returned object `diag` is a Python `dict` containing these quantities.

`FiniteVoronoiSimulator.`**`build`**`(`*connect=True*`)`

Build the finite Voronoi structure and compute forces, returning a dictionary of diagnostics.

Do the following:

- Build Voronoi (+ extensions)
- Get cell connectivity
- Compute per-cell quantities and derivatives
- Assemble forces

Parameters

`connect` (*`bool`*) – Whether to compute cell connectivity information. Setting this to `False` saves some computation time (though very marginal) when connectivity is not needed.

Returns

A dictionary containing forces and geometric properties with keys:

- **forces**: (N,2) array of forces on cell centers
- **areas**: (N,) array of cell areas
- **perimeters**: (N,) array of cell perimeters
- **arclens**: (N,) array of non-contacting edge (arc) lengths per cell
- **coord_nums**: (N,) integer array of coordination numbers per cell
- **vertices**: (M,2) array of all Voronoi + extension vertices
- **edges_type**: List-of-lists of edge types per cell (1=straight, 0=circular arc)
- **regions**: List-of-lists of vertex indices per cell
- **connections**: (K,2) array of connected cell index pairs

Return type

dict[*str*, *object*]

For more examples and detailed usage instructions, please refer to the *Examples* and *API reference* sections.

CHAPTER

TWO

EXAMPLES

Note

To run the examples below, you need to install the **tqdm** package for progress bars. The *Jupyter Notebooks* in examples/jupyter/ additionally require **jupyter** and **ipywidgets** as well.

2.1 Relaxation to mechanical equilibrium

The following example shows how 100 cells relax to mechanical equilibrium from a squeezed initial configuration using the **PyAFV** package.

```
import numpy as np
import matplotlib.pyplot as plt
import tqdm
import pyafv as afv


np.random.seed(42)

N = 100          # number of cells
radius = 1.0     # maximal radius
mu = 1.0         # mobility
dt = 0.01        # time step

# Parameter set
phys = afv.PhysicalParams(
    r=radius,
    A0=np.pi*(radius**2),
    P0=4.8*radius,
    KA=1.0,
    KP=1.0,
    Lambda=0.2
)

# Do not set delta unless you know what you are doing.
# We set it to zero here for comparison with the our primitive results.
phys = phys.replace(delta=0.0)

# Initial positions
```

(continues on next page)

(continued from previous page)

```
pts = np.random.rand(N, 2)*0.3 + 0.35  # shape (N,2)
pts *= 25.

# Initialize simulator
sim = afv.FiniteVoronoiSimulator(pts, phys)

# Plot initial configuration
fig, ax = plt.subplots()
sim.plot_2d(ax=ax)
plt.show()

# Relaxation to mechanical equilibrium
for _ in tqdm.tqdm(range(1000), desc="Relaxation"):
    diag = sim.build()
    forces = diag["forces"]
    pts += mu * forces * dt
    sim.update_positions(pts)

# Plot relaxed configuration
fig, ax = plt.subplots()
sim.plot_2d(ax=ax)
plt.show()
```

See the plotted figures below:

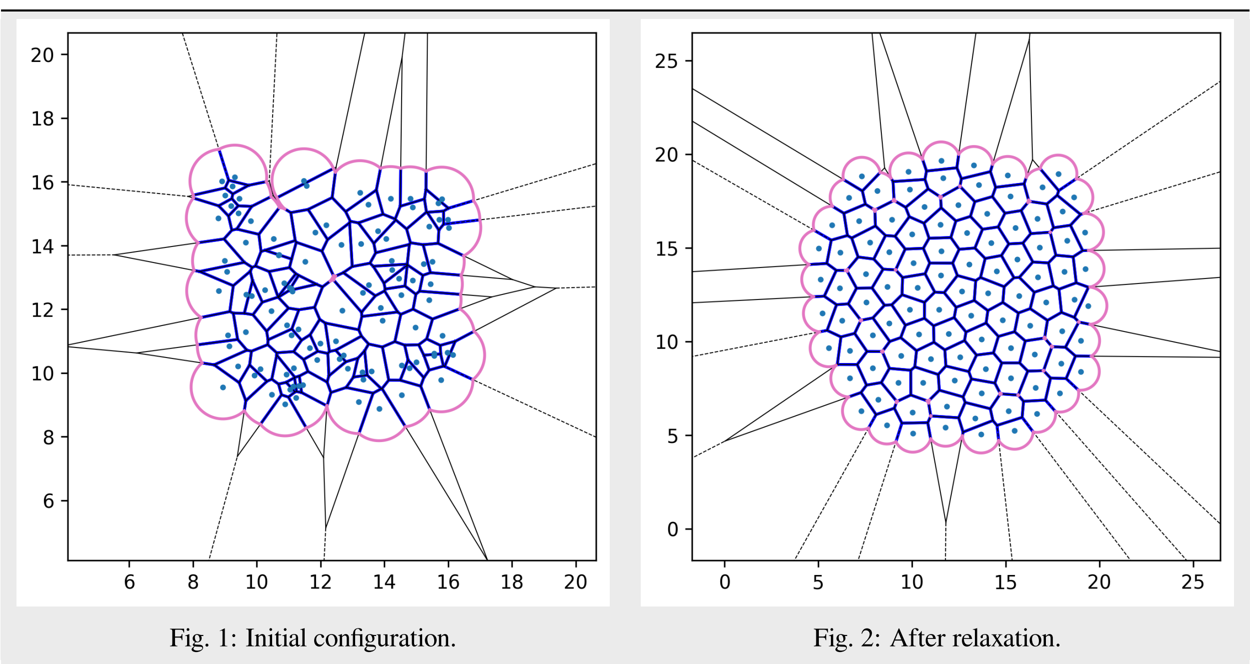


Fig. 1: Initial configuration.

Fig. 2: After relaxation.

2.2 Active finite Voronoi (AFV) dynamics

We can incorporate self-propulsion (active-Brownian-like dynamics) for each cell to model active-matter systems. The resulting equation of motion is

$$\dot{\mathbf{r}}_i = -\mu \boldsymbol{\nabla}_i E + v_0 \mathbf{n}_i,$$

where μ is the mobility, the interaction force on cell $\mathbf{F}_i = -\boldsymbol{\nabla}_i E$, and $\mathbf{n}_i = (\cos\theta_i, \sin\theta_i)$ is a unit orientation vector. The orientation evolves according to

$$\dot{\theta}_i = \sqrt{2D_r}\, \eta_i(t),$$

where the noise satisfies $\langle \eta_i(t) \rangle = 0$ and $\langle \eta_i(t)\eta_j(t') \rangle = \delta_{ij}\, \delta(t - t')$.

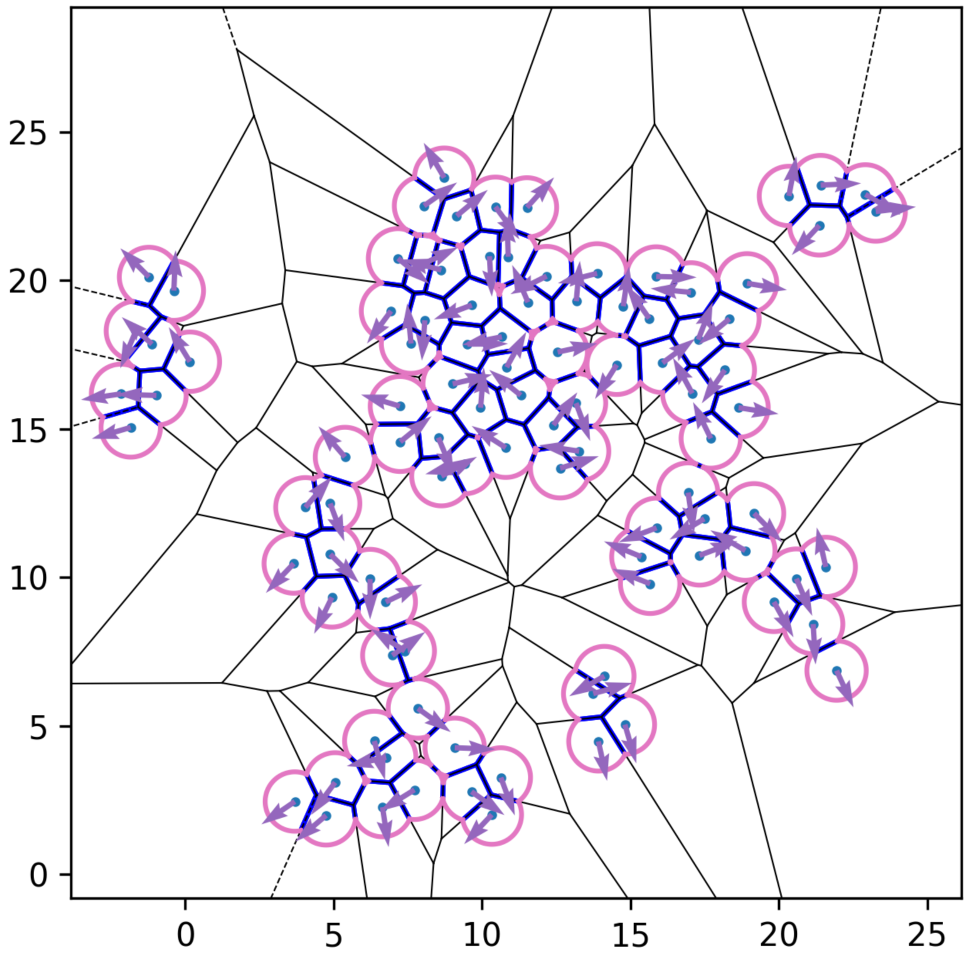
```
import numpy as np
import matplotlib.pyplot as plt
import tqdm
import pyafv as afv


np.random.seed(42)

N = 100          # number of cells
radius = 1.0     # maximal radius
mu = 1.0         # mobility
v0 = 2.4         # self-propulsion speed
Dr = 0.3         # rotational diffusion constant
dt = 0.01        # time step

# Parameter set
phys = afv.PhysicalParams(
    r=radius,
    A0=np.pi*(radius**2),
    P0=4.8*radius,
    KA=1.0,
    KP=1.0,
    Lambda=0.2
)

# Do not set delta unless you know what you are doing.
# We set it to zero here for comparison with the our primitive results.
phys = phys.replace(delta=0.0)

# Initial positions and orientations
pts = np.random.rand(N, 2)*0.3 + 0.35   # shape (N,2)
pts *= 25.
theta = 2. * np.pi * np.random.rand(N) - np.pi

# Initialize simulator
sim = afv.FiniteVoronoiSimulator(pts, phys)

# Relaxation to mechanical equilibrium
for _ in tqdm.tqdm(range(200), desc="Relaxation"):
    diag = sim.build()
```

(continues on next page)

(continued from previous page)

```
    pts += mu * diag["forces"] * dt
    sim.update_positions(pts)

# Active dynamics
for _ in tqdm.tqdm(range(1000), desc="Active dynamics"):
    diag = sim.build()
    forces = diag["forces"]

    active_velocity = v0 * np.column_stack((np.cos(theta), np.sin(theta)))

    pts += (mu * forces + active_velocity) * dt

    # Gaussian white noise
    theta += np.sqrt(2 * Dr * dt) * np.random.randn(N)

    sim.update_positions(pts)


fig, ax = plt.subplots()
sim.plot_2d(ax=ax)
# Plot cell orientations
ax.quiver(pts[:, 0], pts[:, 1], np.cos(theta),
          np.sin(theta), color='C4', scale=20, zorder=3)
plt.show()
```

See the plotted figure below:



2.3 Connectivity between cells

PyAFV can directly output the cell-cell connectivity from the finite Voronoi diagram, where any two connected cells share a straight Voronoi edge.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection
import pyafv as afv


np.random.seed(42)

N = 200          # number of cells
radius = 1.0     # maximal radius

# Parameter set
```

(continues on next page)

(continued from previous page)

```
phys = afv.PhysicalParams(
    r=radius,
    A0=np.pi*(radius**2),
    P0=4.8*radius,
    KA=1.0,
    KP=1.0,
    Lambda=0.2
)

# Initial positions
pts = np.random.rand(N, 2)*0.3 + 0.35  # shape (N,2)
pts *= 70.

# Initialize simulator
sim = afv.FiniteVoronoiSimulator(pts, phys)
diag = sim.build()
connect = diag["connections"]

# Plot initial configuration
fig, ax = plt.subplots()
sim.plot_2d(ax=ax)

# Plot the connections between cells
num_connections = connect.shape[0]
if num_connections > 0:
    i_masked = connect[:, 0]
    j_masked = connect[:, 1]
    # Build list of segments (line endpoints) for visualization, shape: (num_pairs, 2,
↪ 2)
    segments = np.stack([pts[i_masked], pts[j_masked]], axis=1)
    # Create LineCollection
    lc = LineCollection(segments, colors="C7", linewidths=1.5, zorder=0)
    ax.add_collection(lc)

plt.show()
```

2.4 Custom plotting

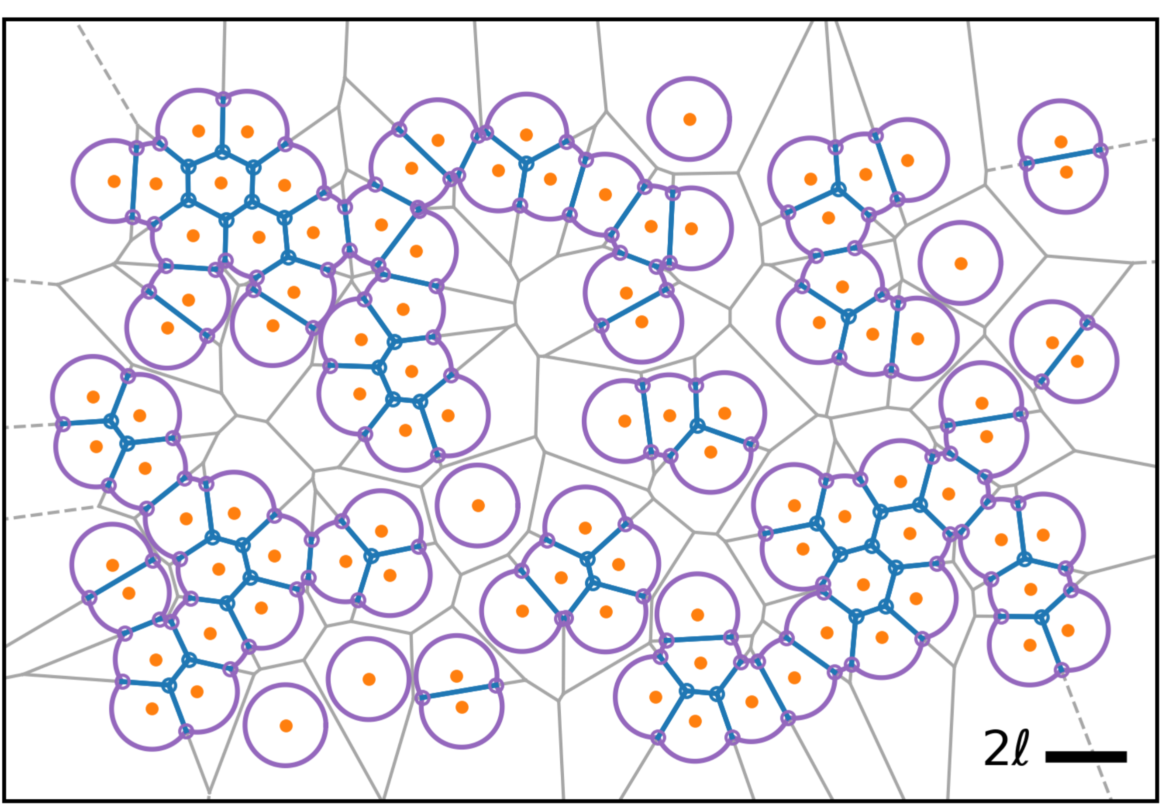
Open in Colab

See examples/jupyter/custom_plot.ipynb for an example of custom plotting using **PyAFV**, or you can run the notebook directly on **Google Colab** by clicking the badge above.

This example, together with the example in the *section using periodic boundary conditions*, show how to use `pyafv.FiniteVoronoiSimulator.build()` returned `dict` to plot the Voronoi diagram with custom styling, including coloring cells by their area and customizing edge colors and widths. Both a **serial** version (for *illustration*) and a **vectorized** version (⚡*much faster*) of the *custom plotting routine* are provided in the notebooks.

Important

Starting from v0.4.10, we also provide a standalone utility function `pyafv.visualize_2d()`, which wraps the **vectorized** plotting routine for easier reuse (see the *next section*). This function is generally preferred over the original `pyafv.FiniteVoronoiSimulator.plot_2d()` method.



This notebook also shows how to access additional internal information via `pyafv.FiniteVoronoiSimulator._build_voronoi_with_extensions()` and `pyafv.FiniteVoronoiSimulator._per_cell_geometry()` for advanced plotting. The public `pyafv.FiniteVoronoiSimulator.build()` method serves as a higher-level wrapper around these two and other lower-level routines.

FiniteVoronoiSimulator.**_build_voronoi_with_extensions**(*joggle=False*)

Build standard Voronoi structure for current points.

For N<=2, emulate regions. For N>=3, extend infinite ridges, add extension vertices, and update regions accordingly. Return the augmented structures.

Warning

This is an internal method. Use with caution.

Parameters

joggle (*bool*) – Whether to joggle input points (N>=3) slightly to avoid precision issues (e.g., collinearities, co-circularities).

Returns

A *tuple* containing:

- **vor**: SciPy Voronoi object for current points with extensions.
- **vertices_all**: (M,2) array of all Voronoi vertices including extensions.
- **ridge_vertices_all**: (R,2) array of vertex indices for each ridge, including extensions.
- **num_vertices**: Number of Voronoi vertices before adding extension.
- **vertexpair2ridge**: *dict* mapping vertex index pairs to ridge index.
- **vertex_points**: *dict* mapping vertex index to list of associated point indices.

Return type

tuple[*Voronoi*, *ndarray*, *ndarray*, *int*, *dict*[*tuple*[*int*,*int*], *int*], *dict*[*int*, *list*[*int*]]]

FiniteVoronoiSimulator.**_per_cell_geometry**(*vor*, *vertices_all*, *ridge_vertices_all*, *num_vertices*, *vertexpair2ridge*)

Build the finite Voronoi per-cell geometry and energy contributions.

Iterate each cell to:

- sort polygon/arc vertices around each cell
- classify edges (1 = straight Voronoi edge; 0 = circular arc)
- compute area/perimeter for each cell
- accumulate derivatives w.r.t. vertices (dA_poly/dh, dP_poly/dh)
- register "outer" vertices created at arc intersections and track their point pairs

> **Warning**
>
> This is an internal method. Use with caution.

Parameters

- **vor** (*Voronoi*) – SciPy Voronoi object for current points with extensions.
- **vertices_all** (*ndarray*) – (M,2) array of all Voronoi vertices including extensions.
- **ridge_vertices_all** (*ndarray*) – (R,2) array of vertex indices for each ridge.
- **num_vertices** (*int*) – Number of Voronoi vertices before adding extension.
- **vertexpair2ridge** (*dict[tuple[int, int], int]*) – *dict* mapping vertex index pairs to ridge index.

Returns

A diagnostics dictionary containing:

- **vertex_in_id**: *set* of inner vertex ids.
- **vertex_out_id**: *set* of outer vertex ids.
- **vertices_out**: (L,2) array of outer vertex coordinates.
- **vertex_out_points**: (L,2) array of point index pairs associated with each outer vertex.

- **vertex_out_da_dtheta**: Array of dA/dtheta for all outer vertices.
- **vertex_out_dl_dtheta**: Array of dL/dtheta for all outer vertices.
- **dA_poly_dh**: Array of dA_polygon/dh for each vertex.
- **dP_poly_dh**: Array of dP_polygon/dh for each vertex.
- **area_list**: Array of polygon areas for each cell.
- **perimeter_list**: Array of polygon perimeters for each cell.
- **arclen_list**: Array of non-contacting arc lengths for each cell.
- **point_edges_type**: List of lists of edge types per cell.
- **point_vertices_f_idx**: List of lists of vertex ids per cell.
- **num_vertices_ext**: Number of vertices including infinite extension vertices.
- **coord_nums**: (N,) integer array of coordination numbers per cell.

Return type

dict[*str*, *object*]

2.4.1 A standalone plotting utility

Inspired by the custom plotting examples used in daily research, we have implemented a **standalone** plotting utility `pyafv.visualize_2d()` that can be used to plot the finite Voronoi diagram with *color-filled cells* and *customizable styling*. A simple example usage is shown below:

```
import numpy as np
import matplotlib.pyplot as plt
import pyafv as afv

N = 100                                         # number of cells
pts = np.random.rand(N, 2) * 10                 # initial positions
params = afv.PhysicalParams(r=1.0)              # use default parameter values
sim = afv.FiniteVoronoiSimulator(pts, params)   # initialize the simulator
diag = sim.build()                              # compute forces and geometry

fig, ax = plt.subplots()
afv.visualize_2d(pts, diag, r=1.0, ax=ax)        # visualize the FV diagram
plt.show()
```

This `pyafv.visualize_2d()` function is designed to be **flexible** and **efficient** (vectorized), and it should satisfy most common plotting needs for 2D finite Voronoi diagrams.

visualize_2d(*pts*, *diag*, *r*, *ax=None*, * *(Keyword-only parameters separator (PEP 3102))*, *selected=None*, ***kw*)

Visualize a 2D snapshot using the diagnostic dictionary *diag* generated by `pyafv.FiniteVoronoiSimulator.build()`. This is basically a wrapper around the vectorized custom plotting functions from the example notebooks and generally preferred over the original `pyafv.FiniteVoronoiSimulator.plot_2d()` method. The plotting style follows `scipy.spatial.voronoi_plot_2d()`.

> **Note**
>
> If you are visualizing a *diag* from `ParallelFiniteVoronoiSimulator.build()`, you should use `visualize_2d_parallel()` instead.

Parameters

- **pts** (*ndarray*) – An (N, 2) array of point coordinates.
- **diag** (*dict[str, object]*) – A diagnostic *dict* containing Voronoi diagram information.
- **r** (*float*) – Maximum radius (or denoted as ℓ) used for drawing arcs.
- **ax** (*Axes | None*) – If provided, draw into the axes; otherwise create a new one.
- **selected** (*array-like | None, optional*) – Cells to draw. This can be either a one-dimensional array of integer indices or a boolean mask with length equal to `len(pts)`. If *None*, draw all cells.
- **cell_colors** (*color or list, optional*) – A single color or a sequence of colors for filling cells, default 'C2'. Use *None* for no fill. If *selected* is provided, full-length per-cell color sequences are sliced to the selected cells.
- **fill_alpha** (*float, optional*) – Specifies the alpha for cell fills, default 0.1.
- **fill_zorder** (*float, optional*) – Specifies the z-order for cell fills, default 0.
- **show_points** (*bool, optional*) – Add cell center points to the plot, default *False*.
- **point_size** (*float, optional*) – Specifies the marker area for the points, default 4. If *selected* is provided, full-length per-cell sequences are sliced to the selected cells.
- **point_colors** (*color or list, optional*) – A single color or a sequence of colors for the points, default 'C0'. If *selected* is provided, full-length per-cell color sequences are sliced to the selected cells.
- **point_zorder** (*float, optional*) – Specifies the z-order for the points, default 3.
- **straight_colors** (*color | None, optional*) – Color for straight contact edges, default 'C0'. Use *None* to skip drawing straight edges.
- **straight_lw** (*float, optional*) – Line width for straight edges, default 1.0.
- **straight_alpha** (*float, optional*) – Alpha for straight edges, default 1.0.
- **straight_capstyle** (*str, optional*) – Cap style for straight edges, default 'butt'.
- **straight_zorder** (*float, optional*) – Z-order for straight edges, default 2.
- **arc_colors** (*color | None, optional*) – Color for arc non-contact edges, default 'C2'. Use *None* to skip drawing arc edges.
- **arc_lw** (*float, optional*) – Line width for arc edges, default 1.0.
- **arc_alpha** (*float, optional*) – Alpha for arc edges, default 1.0.
- **arc_capstyle** (*str, optional*) – Cap style for arc edges, default 'butt'.
- **arc_zorder** (*float, optional*) – Z-order for arc edges, default 1.
- **auto_adjust_bounds** (*bool, optional*) – Whether to automatically adjust the plot bounds to fit the diagram, default *True*.

Returns

The matplotlib figure object representing the entire canvas.

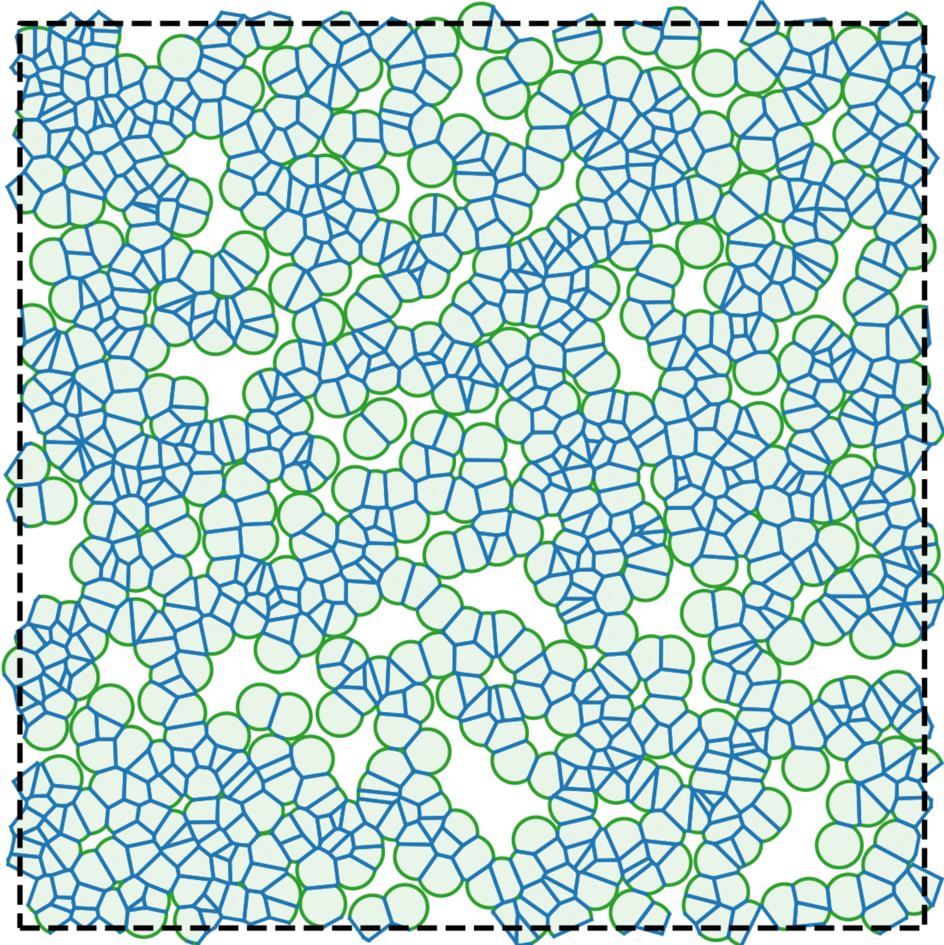
Return type
Figure

2.5 Periodic boundary conditions

PyAFV uses open boundary conditions in 2D by default, but it is also possible to implement *periodic boundary conditions* via a tiling of the edge regions. See examples/jupyter/periodic_plotting.ipynb for an example (or you can run the notebook directly on **Google Colab** by clicking the badge above), and the generated figure is shown below:

Note

Starting from v0.4.10, the tiling routine is also wrapped as a utility function `pyafv.tile_pbc()` (experimental).



2.6 Varying cell target areas from cell to cell

Starting from **PyAFV** v0.3.5, the simulator supports cell-specific preferred areas, allowing the target area A_0 to vary from cell to cell, i.e., $\{A_{0,i}\}$.

A new read-only property *pyafv.FiniteVoronoiSimulator.preferred_areas* has been added. It returns the current preferred areas for all cells:

FiniteVoronoiSimulator.**preferred_areas**

Preferred areas $\{A_{0,i}\}$ for all cells (read-only).

Returns

A copy of the internal preferred area array.

Return type

ndarray

To modify the preferred areas, the method *pyafv.FiniteVoronoiSimulator.update_preferred_areas()* is provided:

FiniteVoronoiSimulator.**update_preferred_areas**(*A0*)

Update the preferred areas for all cells.

Parameters

A0 (*float | ndarray*) – New preferred area(s) for all cells, either as a scalar or as an array of shape (N,).

Raises

ValueError – If *A0* does not match cell number.

Return type

None

Here is an example usage:

```
import numpy as np
from pyafv import FiniteVoronoiSimulator, PhysicalParams

# Initialize simulator
N = 100
pts = np.random.rand(N, 2) * 10
phys = PhysicalParams(r=1.0, A0=np.pi)
sim = FiniteVoronoiSimulator(pts, phys)

# Set varying preferred areas per cell
varying_A0 = np.pi + 0.2 * np.random.randn(N)
sim.update_preferred_areas(varying_A0)

# Access via property
print(sim.preferred_areas)  # shape: (N,)

# Run simulation...
```

In addition, the *pyafv.FiniteVoronoiSimulator.update_positions()* method now accepts an optional second argument to update the preferred areas:

`FiniteVoronoiSimulator.`**`update_positions`**`(`*pts*, *A0=None*`)`

Update cell center positions.

Note

If the number of cells changes, the preferred areas for all cells are reset to the default value—defined either at simulator instantiation or by *`update_params()`*—unless *A0* is explicitly specified.

Parameters

- **`pts`** (*`ndarray`*) – New cell center positions.
- **`A0`** (*`float | ndarray | None`*) – Optional, set new preferred area(s).

Raises

- **`ValueError`** – If *pts* does not have shape (N,2).
- **`ValueError`** – If *A0* is an array and does not have shape (N,).

Return type

None

And the *`pyafv.FiniteVoronoiSimulator.update_params()`* method will also re-initialize the preferred areas for all cells using the supplied value of *A0* in *phys*:

`FiniteVoronoiSimulator.`**`update_params`**`(`*phys*`)`

Update physical parameters.

Parameters

`phys` (`PhysicalParams`) – New *`PhysicalParams`* object.

Raises

`TypeError` – If *phys* is not an instance of *`PhysicalParams`*.

Return type

None

Warning

This also resets all preferred cell areas to the new value of *A0*.

CHAPTER

THREE

MULTIPROCESSING

Starting in v0.4.12, PyAFV provides *pyafv.ParallelFiniteVoronoiSimulator* for domain-decomposed AFV simulations using **Python multiprocessing** (CPU parallelism). The simulator splits the full point set into rectangular owned domains, adds halo points around each domain, builds a local finite Voronoi diagram for each subdomain, and merges the owned-cell diagnostics back into the global point ordering.

This feature is intended for large systems ($N \gtrsim 10^4$) where the cost of local Voronoi builds is high enough to offset the overhead of domain decomposition, inter-process data transfer, and duplicated halo work. The crossover depends on hardware, point density, and the chosen domain grid. For small systems, *pyafv.FiniteVoronoiSimulator* may still be faster.

Note

See *Benchmarking parallel build* for a build-time benchmark comparing *pyafv.FiniteVoronoiSimulator* with *pyafv.ParallelFiniteVoronoiSimulator*. The benchmark shows that multiprocessing is generally faster once the system is not too small, especially for large systems.

3.1 Basic usage

The interface is similar to *pyafv.FiniteVoronoiSimulator*, but the domain grid shape and number of worker processes are supplied when the simulator is created:

```
import numpy as np
import pyafv

points = np.random.default_rng(42).random((10_000, 2)) * 100.0
phys = pyafv.PhysicalParams(r=1.0)

sim = pyafv.ParallelFiniteVoronoiSimulator(
    points,
    phys,
    grid_shape=(4, 4),
    n_workers=16,
)

diag = sim.build()
```

PyAFV decomposes the domain into an `a`-by-`b` grid of subdomains set by `grid_shape=(a, b)`. The number of subdomains is therefore ab. `n_workers` is the number of worker processes to use. In practice, set `n_workers` to the

number of CPU cores available to the Python job, but no larger than the number of subdomains; additional workers would remain idle anyway.

By default, *pyafv.ParallelFiniteVoronoiSimulator.build()* uses `connect=False`. This differs from *pyafv.FiniteVoronoiSimulator.build()*, where `connect=True` by default. This default avoids connectivity work during runs where only forces are needed.

Tip

Decomposing the whole system into smaller domains can also improve the accuracy of `scipy.spatial.Voronoi` for large systems, since Qhull's floating-point tolerance scales with the system span; see issue #38.

3.2 Repeated build steps

For repeated calls with `n_workers > 1`, put the time-stepping loop inside the context manager. This creates the worker processes once and reuses them across build steps:

```
dt = 0.01
n_steps = 100

with sim:
    for step in range(n_steps):
        diag = sim.build()
        points += diag["forces"] * dt
        sim.update_positions(points)
```

If the context manager is not used, each call to `build` creates and shuts down a new process pool. That is usually slower in a loop.

Important

When using multiprocessing in a Python script, put the executable code behind the standard Python guard:

```
def main():
    # Initialize points, phys, and n_steps here.
    sim = pyafv.ParallelFiniteVoronoiSimulator(points, phys, (4, 4), 16)
    with sim:
        for step in range(n_steps):
            diag = sim.build()
            # followed by time-stepping code...

if __name__ == "__main__":
    main()
```

This guard is required when Python uses the `spawn` multiprocessing start method. This includes **Windows** and modern **macOS** by default; **Linux** usually defaults to `fork`, but the guard is still recommended for portable scripts.

In Jupyter notebooks, the parallel simulator may still work without this guard, but long production runs are usually more robust when launched from a script.

3.3 Halo width

Each owned domain is expanded by `halo_width` in every direction before the local Voronoi calculation is built. If `halo_width` is not specified, PyAFV uses `4.01 * phys.r` ($> 4\ell$). This should be large enough that the geometry and force for an owned cell are not affected by missing neighboring cells outside the local domain.

3.4 Decomposition method

The low-level helper *pyafv.decompose_points()* and the parallel simulator both support three halo-collection methods:

```
sim = pyafv.ParallelFiniteVoronoiSimulator(
    points,
    phys,
    grid_shape=(4, 4),
    n_workers=16,
    decomposition_method="dense",
)
```

- `"dense"` is the default. It builds a dense domain-by-point mask and is often faster for moderate systems.
- `"binned"` reuses the owned-domain bins to check fewer candidate halo points. It can be faster for large systems with many domains.
- `"sorted_x"` avoids the dense temporary mask by sorting points along the `x`-axis and querying candidate halo ranges. It uses less temporary memory, but can be slower for typical moderate-sized systems.

3.5 Visualization

Parallel plotting diagnostics are local to each domain and should be requested explicitly:

```
import matplotlib.pyplot as plt

diag = sim.build(plot_mode=True)
fig, ax = plt.subplots()
pyafv.visualize_2d_parallel(points, diag, r=phys.r, ax=ax)
plt.show()
```

Use *pyafv.visualize_2d_parallel()* for diagnostics from *pyafv.ParallelFiniteVoronoiSimulator.build()*; `plot_mode` must be set to `True`. Use *pyafv.visualize_2d()* for diagnostics from *pyafv.FiniteVoronoiSimulator.build()*.

3.6 A complete example

The following code provides a complete example that simulates 10,000 cells using a `3 x 3` domain decomposition and 9 worker processes:

```
import numpy as np
import pyafv as afv
from tqdm import tqdm
import matplotlib.pyplot as plt

```

(continues on next page)

(continued from previous page)

```
def main():
    radius = 1.0
    points = np.random.default_rng(42).random((10_000, 2)) * 100.0
    phys = afv.PhysicalParams(r=radius)

    sim = afv.ParallelFiniteVoronoiSimulator(points, phys, (3, 3), n_workers=9)

    dt = 0.01
    with sim:
        for _ in tqdm(range(1000)):
            diag = sim.build()
            points += diag["forces"] * dt
            sim.update_positions(points)

        diag = sim.build(plot_mode=True)

    fig, ax = plt.subplots()
    afv.visualize_2d_parallel(points, diag, r=radius, ax=ax)
    plt.show()


if __name__ == "__main__":
    main()
```

Run the code as you would a standard Python script. If your system has 9 or more CPU cores available, the parallel implementation should run substantially faster than the single-process version.

3.7 Running on clusters

3.7.1 Running on a single node

Python multiprocessing runs worker processes on the **same node** as the main Python process. It does not distribute work across multiple nodes. On a Slurm cluster, use one task with multiple CPUs, for example:

```
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=16
```

Then use the same number of workers in Python:

```
sim = pyafv.ParallelFiniteVoronoiSimulator(points, phys, (4, 4), 16)
```

3.7.2 Multi-node parallelism: MPI

For multi-node domain decomposition, use an **MPI**-based (**M**essage **P**assing **I**nterface) implementation instead of Python multiprocessing. PyAFV does **not** currently provide an MPI implementation. However, it exposes the point-based domain-decomposition function `pyafv.decompose_points()` as a low-level helper:

decompose_points(*points*, *grid_shape=(2, 2)*, *halo_width=0.0*, ***, *domain_bounds=None*, *method='dense'*)

Decompose points into owned grid domains plus halo/local points.

Points on internal owned-domain boundaries are assigned to exactly one domain by using right-sided binning and clipping at the outermost boundary. Halo/local domains include points on all halo box edges.

> **Tip**
>
> This function is independent of the finite Voronoi model. It only computes global/local point-index bookkeeping.

Parameters

- **points** (*ndarray*) – (N,2) array of point coordinates.
- **grid_shape** (*tuple[int, int]*) – Number of owned domains in the x and y directions.
- **halo_width** (*float*) – Width added to each side of every owned domain to collect local halo points.
- **domain_bounds** (*tuple[tuple[float, float], tuple[float, float]] | None*) – Optional domain bounds as `((xmin, xmax), (ymin, ymax))`. If *None*, bounds are inferred from *points*.
- **method** (*Literal['dense', 'binned', 'sorted_x']*) – Method used to collect halo points. `"dense"` builds a dense domain-by-point mask and is usually faster for moderate systems. `"binned"` reuses the owned-domain bins to reduce the number of candidate points checked for each halo and is usually faster for many domains. `"sorted_x"` uses less temporary memory.

Returns

One list of `DomainDecomposition` objects in row-major order, where each element stores the local and global point information for a single domain.

Return type

list[DomainDecomposition]

Raises

- **ValueError** – If *points* does not have shape (N,2).
- **ValueError** – If *points* contains non-finite values.
- **ValueError** – If *grid_shape*, *halo_width*, *domain_bounds*, or *method* is invalid.

Using this function, users can build an MPI wrapper around the parallel simulator with mpi4py. The following code shows a minimal example with two MPI ranks. First, the full system is decomposed into a `2 x 1` grid across MPI ranks. Then, each rank further decomposes its local domain into `2 x 2` subdomains for multiprocessing, using four worker processes per rank ($2 \times 4 = 8$ workers in total).

```python
# MPI_wrap.py

import numpy as np
import pyafv as afv


def main():
    # ========================================================
    # Put MPI setup inside main() so spawned multiprocessing
    # workers do not import/initialize MPI at top level.
    from mpi4py import MPI

    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
```

(continues on next page)

(continued from previous page)

```
size = comm.Get_size()    # should be 2 for this test
# ==========================================================

radius = 1.0
points = np.random.default_rng(42).random((10_000, 2)) * 100.0
phys = afv.PhysicalParams(r=radius)

# 2x1 domain decomposition for the two MPI ranks
if rank == 0:
    domains = afv.decompose_points(points, (2, 1), halo_width=4.01*radius)

domains = comm.bcast(domains if rank == 0 else None, root=0)

# Each rank processes its own domain
domain = domains[rank]
local_points = domain.local_pts

# Each rank creates its own simulator with 4 workers
sim = afv.ParallelFiniteVoronoiSimulator(local_points, phys, (2, 2), n_workers=4)

dt = 0.01
with sim:
    diag = sim.build()

    # gather diagnostics from all ranks
    diag_all = comm.gather(diag, root=0)

    # combine diagnostics on rank 0
    if rank == 0:
        forces = np.zeros_like(points, dtype=float)
        areas = np.zeros(points.shape[0], dtype=float)
        perimeters = np.zeros(points.shape[0], dtype=float)
        # ... add more diagnostics as needed ...

        for mpi_domain, diag in zip(domains, diag_all):
            owned_local_ids = mpi_domain.owned_local_ids
            owned_global_ids = mpi_domain.local_global_ids[owned_local_ids]

            forces[owned_global_ids] = diag["forces"][owned_local_ids]
            areas[owned_global_ids] = diag["areas"][owned_local_ids]
            perimeters[owned_global_ids] = diag["perimeters"][owned_local_ids]

        diag_combined = {
            "forces": forces,
            "areas": areas,
            "perimeters": perimeters,
        }

        # Update points
        points += forces * dt

        # upate domain decomposition with new points
```

(continues on next page)

(continued from previous page)

```
            domains = afv.decompose_points(points, (2, 1), halo_width=4.01*radius)

        domains = comm.bcast(domains, root=0)
        domain = domains[rank]
        local_points = domain.local_pts
        sim.update_positions(local_points)

    print(f"Rank {rank} finished simulation.")


if __name__ == "__main__":
    main()
```

On Linux clusters, it may be necessary to explicitly set the multiprocessing start method to `spawn` for better compatibility with MPI:

```
import multiprocessing as mp

# ... define main() here ...

if __name__ == "__main__":
   mp.set_start_method("spawn", force=True)
   main()
```

To run the above code:

```
(.venv) $ mpiexec -n 2 python MPI_wrap.py
```

On a Slurm cluster, request 2 MPI tasks and 4 CPUs per task:

```
#SBATCH --ntasks=2
#SBATCH --cpus-per-task=4

mpiexec -n "$SLURM_NTASKS" \
  --map-by "slot:PE=$SLURM_CPUS_PER_TASK" \
  --bind-to core \
  python MPI_wrap.py
```

See *Benchmarking hybrid parallel build: MPI + Python multiprocessing* for a benchmark of this approach on the **Rockfish** HPC cluster at Johns Hopkins University.

Caution

MPI ranks can be placed on different nodes, while the worker processes created by each rank run on that rank's node. Therefore, `--cpus-per-task` must not exceed the number of CPU cores available on a single node.

Visualization of MPI-based parallel simulations is also possible. Simply loop over each rank's domain and call `pyafv.visualize_2d_parallel()` using the diagnostics from each rank (`plot_mode=True` must be passed to the `build` method to enable plotting).

```
if rank == 0:       # use only one rank to plot
   fig, ax = plt.subplots()
```

(continues on next page)

(continued from previous page)

```
for idx in range(size):
    diag = diag_all[idx]
    domain = domains[idx]
    local_points = domain.local_pts

    afv.visualize_2d_parallel(local_points, diag, r=radius, ax=ax,
           selected=domain.owned_local_ids)  # remove halo points for each rank's␣
↪domain

ax.set_xlim(-10, 110)
ax.set_ylim(-10, 110)
plt.show()
```

However, if the number of cells is large enough to require multiple compute nodes ($N \gtrsim 10^6$), visualizing the full system is usually not very informative anyway. In such large-scale simulations, the finite Voronoi structures are often difficult to distinguish visually, and it is generally more effective to visualize cells simply as points.

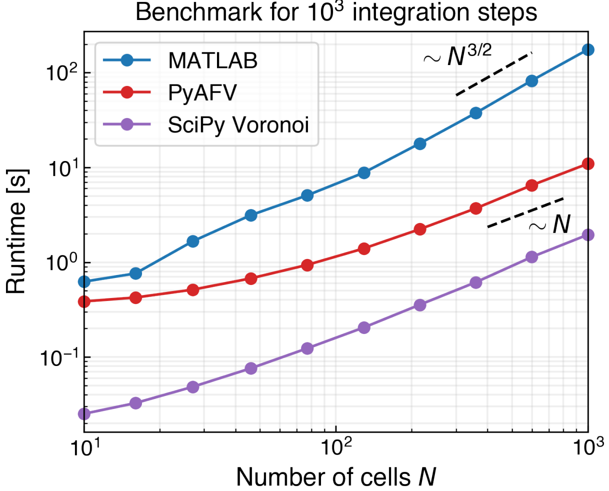
CHAPTER

FOUR

PERFORMANCE

4.1 Measuring performance

PyAFV has been benchmarked against the **MATLAB** implementation of the active finite Voronoi model from Ref. [1] by measuring the wall-clock runtime for simulations of varying system sizes. The results are shown in the figure; each data point corresponds to 10^3 integration steps, averaged over three independent runs. The results show that *PyAFV* exhibits **near-linear** scaling, approximately $\mathcal{O}(N)$—comparable to the scaling behavior of **SciPy**'s Voronoi implementation `scipy.spatial.Voronoi`—whereas the original *MATLAB* code scales more steeply, at roughly $\mathcal{O}(N^{3/2})$. This difference will lead to a significant speedup, particularly for large systems ($N \gtrsim 10^3$).



Note

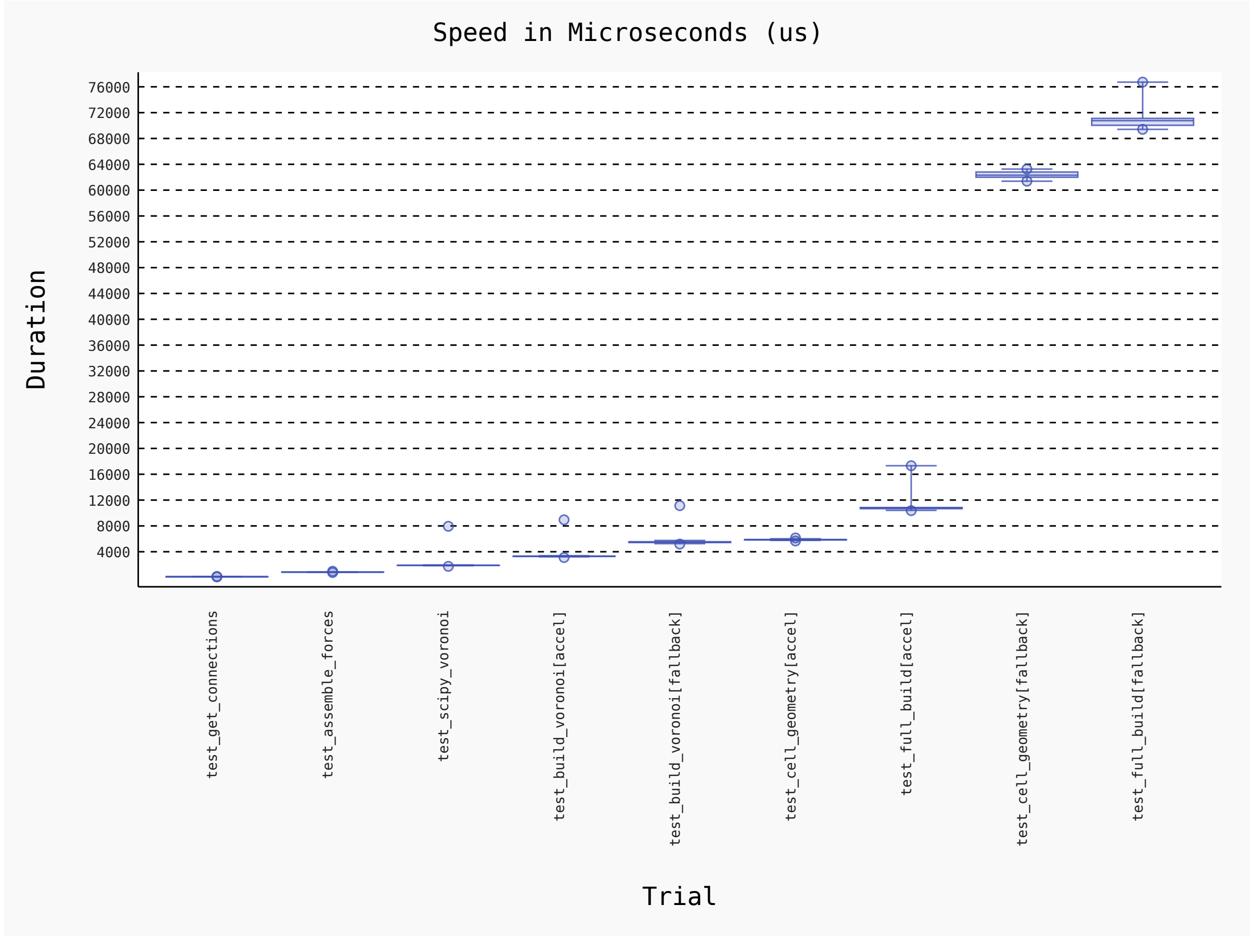
All benchmark results were obtained on a MacBook Pro (14-in, 2024) equipped with an Apple M4 Pro chip (12-core) and 24 GB of RAM, running macOS 15.6. The *MATLAB* implementation was executed using **MATLAB R2025a**, while *PyAFV* was run using **Python 3.13.5** with the **PyAFV v0.4.3** default Cython backend (**PyAFV v0.4.12** for *parallel build benchmark*).

4.2 Benchmarking backends

In addition, there is a set of lightweight benchmarks in `tests` using **pytest-benchmark**, e.g., `test_bench_build.py` compares the runtimes of the Cython and pure-Python backends . To run it:

```
(.venv) $ uv run pytest tests/test_bench_build.py --benchmark-only --benchmark-warmup=on --benchmark-histogram
```

This will display the benchmark results and generate an interactive SVG histogram file (click to see the detailed timing results for each method):



The histogram above summarizes the runtimes of the core routines invoked by *pyafv.FiniteVoronoiSimulator.build()* for a system of $N = 1000$ cells. The `test_scipy_voronoi` benchmark measures the execution time of **SciPy**'s Voronoi tessellation, which serves as *a baseline for comparison*. This SciPy routine is called internally by *pyafv.FiniteVoronoiSimulator._build_voronoi_with_extensions()*, corresponding to the `test_build_voronoi` benchmark shown in the histogram. From this comparison, we see that SciPy's Voronoi computation accounts for approximately 60% of the total runtime of that method.

Hint

The suffixes `[accel]` and `[fallback]` in the benchmark names indicate whether the Cython backend or the pure-Python fallback implementation was used.

The remaining dominant cost arises from the additional per-cell processing performed in *pyafv.FiniteVoronoiSimulator._per_cell_geometry()*. As shown in the histogram, the Cython-backed implementation substantially reduces the runtime of this step, bringing it down to a level comparable to that of SciPy's Voronoi tessellation.

4.3 Benchmarking parallel build

This figure shows the cost of a single *pyafv.FiniteVoronoiSimulator.build()* call with `connect=False` against the domain-decomposed multiprocess implementation. For each system size, the same ten randomly generated point sets were used for all methods; the bars show the mean build time, while the right panel shows the speedup relative to *pyafv.*

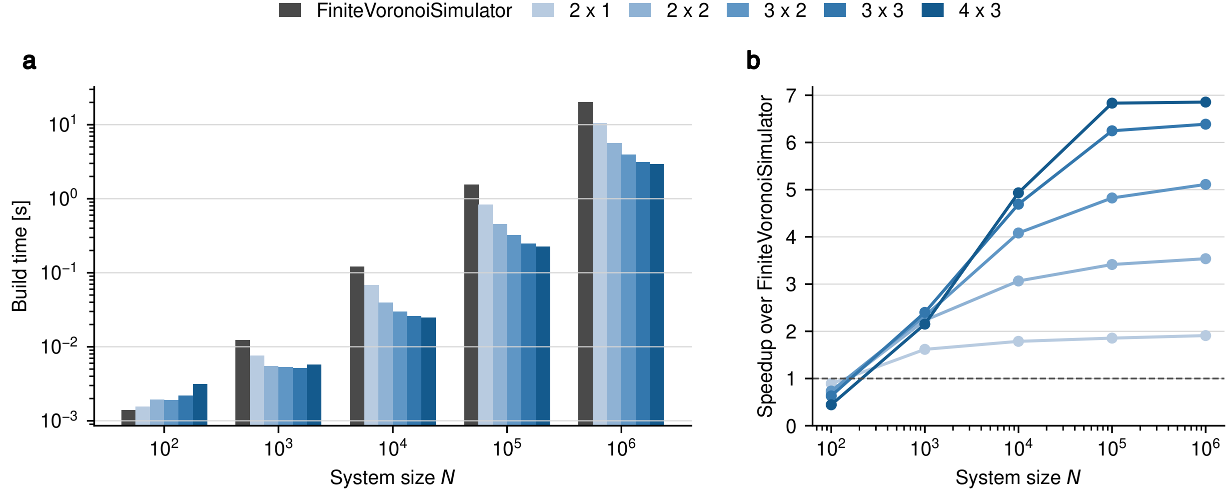


Fig. 1: Build-time benchmark for `pyafv.FiniteVoronoiSimulator` and `pyafv.ParallelFiniteVoronoiSimulator`.

`FiniteVoronoiSimulator`. Parallel timings were measured with a persistent worker pool and three unmeasured warm-up builds, so the reported times do not include one-time worker startup. The number of workers is set equal to the number of domains.

For very small systems, multiprocessing overhead dominates. In this benchmark, the parallel implementation is slower than the single-process simulator at $N = 100$, but becomes faster by $N = 1000$. For larger systems, local domain decomposition gives substantial speedups: the `4 x 3` setup reaches about $4.9\times$ at $N = 10^4$, $6.8\times$ at $N = 10^5$, and $6.9\times$ at $N = 10^6$. The speedup is not perfectly linear in the number of workers, likely because the benchmark was run on a laptop with 8 performance cores and 4 efficiency cores rather than on a uniform multi-core CPU.

The following figures show benchmark runs on the **Rockfish** HPC cluster at Johns Hopkins University. The first figure corresponds to runs on the **shared** partition (32 cores/node), while the second shows results from the **parallel** partition (48 cores/node). The serial `pyafv.FiniteVoronoiSimulator` benchmark was allocated 16 GB of memory to avoid out-of-memory (OOM) failures; on Rockfish, this allocation corresponded to 5 CPUs on both partitions. For the parallel simulator runs, both the number of allocated CPUs and the number of workers were set equal to the number of domains. Compared with the laptop benchmark, the parallel speedup scales more cleanly with the number of workers.

The optimal decomposition depends on the number of points and the CPU resources available on the machine. In this benchmark, using more domains generally helps over the tested range, but the best choice should still be checked for each workload because worker scheduling, inter-process data transfer, result merging, and CPU affinity all depend on the hardware and launch configuration.

4.4 Benchmarking hybrid parallel build: MPI + Python multiprocessing

The hybrid benchmark uses MPI for a coarse domain decomposition and `pyafv.ParallelFiniteVoronoiSimulator` inside each rank for a second local multiprocessing decomposition; see *Multi-node parallelism: MPI*. A label such as `(2 x 2) x (4 x 3)` means that the full system is first decomposed into a `2 x 2` MPI-rank grid, and each rank then decomposes its local point set into a `4 x 3` worker grid.

The hybrid timing includes the full wrapper step: coarse domain decomposition on rank 0, MPI broadcast of the domains, local multiprocessing builds, MPI gather of owned-cell forces, and force assembly on rank 0. For the hybrid benchmark runs, the number of Slurm tasks was set to the number of coarse MPI domains, and `--cpus-per-task` was set to the

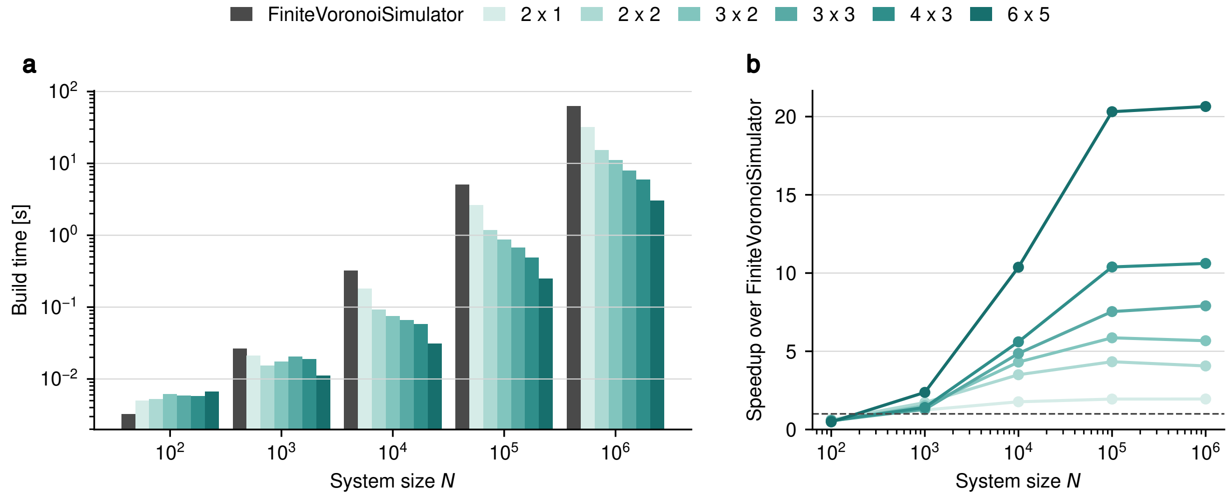


Fig. 2: JHU Rockfish **shared** partition build-time benchmark.

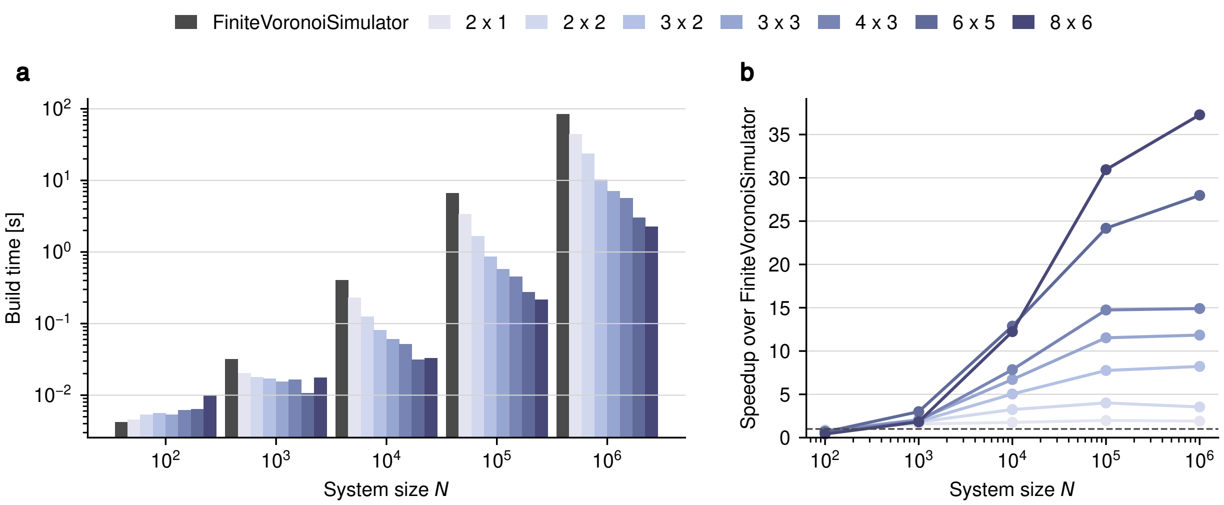


Fig. 3: JHU Rockfish **parallel** partition build-time benchmark.

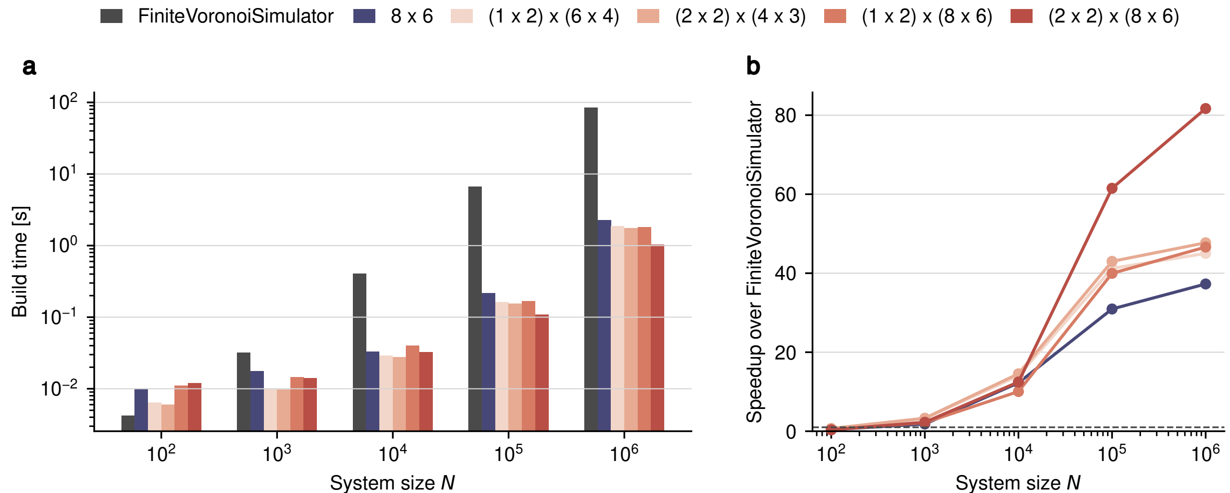


Fig. 4: Benchmark for a hybrid Python multiprocessing + MPI wrapper on JHU Rockfish **parallel** partition.

number of local multiprocessing subdomains. For example, the `(2 x 2) x (4 x 3)` case was launched with 4 MPI ranks and 12 CPUs (12 workers) per rank, for a total of 48 workers.

The equal-total-CPU comparisons are `8 x 6`, `(1 x 2) x (6 x 4)`, and `(2 x 2) x (4 x 3)`, all of which use 48 CPUs (48 workers) in total. The hybrid versions are slightly faster for larger systems in this benchmark. This may be explained by the fact that the pure multiprocessing run has one parent process managing 48 workers, while the hybrid runs distribute that Python multiprocessing coordination across two or four MPI ranks. The hybrid benchmark also gathers and merges only owned-cell forces, while *pyafv.ParallelFiniteVoronoiSimulator.build()* returns a fuller diagnostic dictionary.

The `(1 x 2) x (8 x 6)` case is less efficient than might be expected from its 96 total workers, corresponding to 96 allocated CPUs. One likely reason is the launch layout: the **parallel** partition has 48 cores per node, so this configuration requires two nodes and therefore cross-node MPI communication. By contrast, the 48-worker `(1 x 2) x (6 x 4)` and `(2 x 2) x (4 x 3)` cases can fit on a single node, which reduces MPI communication overhead.

CHAPTER

FIVE

CALIBRATION

Starting from v0.4.0, **PyAFV** provides a dedicated subpackage *pyafv.calibrate* for calibrating the physical parameters of the finite Voronoi (FV) model against a vertex-model-like deformable polygon (DP) model [4]. The calibration is performed by matching the steady states and detachment forces of cell doublets between the two models [3].

> **Important**
>
> *To make it clear, the calibration tools described here are intended for advanced users who are familiar with the underlying assumptions and know exactly what they are doing.*

In most use cases, the default value of the contact truncation threshold `delta` in *pyafv.PhysicalParams* is sufficient and should work well.

> **Note**
>
> In vertex and Voronoi models, the target shape index is defined as $p_0 = P_0/\sqrt{A_0}$. The DP model is expected to be valid only for $p_0 \leqslant 2\sqrt{\pi}$, corresponding to the shape index of a perfect circle. This limitation arises because no additional constraints (e.g., curvature terms) are included in the energy to stabilize non-circular shapes in the DP model.

5.1 How to calibrate against the DP model

PyAFV provides a convenience function *pyafv.calibrate.auto_calibrate()*, which performs the calibration procedure automatically.

auto_calibrate(*phys*, *ext_forces=None*, *dt=0.001*, *nsteps=50000*, *show=None*)

Auto-calibrate the parameters *phys* against the deformable polygon (DP) model.

In this calibration, we simulate an initially steady-state cell doublet under increasing external force dipoles using the DP model; the external force starts from 0 to max(*ext_forces*). We identify the detachment force as the first external force at which detachment occurs. We then search for the `delta` value in the finite Voronoi (FV) model to match this detachment force.

Parameters

- **phys** (`PhysicalParams`) – The initial physical parameters.
- **ext_forces** (*ndarray | None*) – An array of external forces to apply during calibration; defaults to None, which uses `np.linspace(0, 10, 101)[1:]`; should start from a small positive value.
- **dt** (*float*) – Time step for each simulation step.

- **nsteps** (*int*) – Number of simulation steps to run for each external force.
- **show** (*bool | None*) – Whether to print progress information; no need to set it if **tqdm** is installed.

Raises

`TypeError` – If *phys* is not an instance of *PhysicalParams*.

Returns

A *tuple* containing the detachment force and the calibrated *PhysicalParams*. If detachment does not occur within the given force range, return a *NaN* force.

Return type

tuple[*float*, PhysicalParams]

> **Warning**
>
> This function may take some time to run, depending on the parameters. (If **tqdm** is installed, a progress bar will be shown automatically.)
>
> Do not change defaults unless you understand the implications. If you only need a rough or faster calibration, you may change the external force range or interval. Adjusting *dt* and *nsteps* may also speed up simulations, but may affect accuracy; test the `DeformablePolygonSimulator` model separately to ensure accuracy is acceptable.

In brief, the calibration procedure is as follows:

1. Match the steady-state geometry of a cell doublet in the FV and DP models by determining the optimal cell radius ℓ_0; this can be done by `pyafv.PhysicalParams.get_steady_state()` or `pyafv.PhysicalParams.with_optimal_radius()`.
2. Apply progressively increasing pulling forces to the cell doublet in the DP model until detachment occurs, and record the corresponding detachment forces; this can be done by `pyafv.calibrate.DeformablePolygonSimulator` (see *section* below).
3. Identify the value of `delta` in the FV model that reproduces the same detachment forces observed in the DP model; this can be done by `pyafv.target_delta()`.

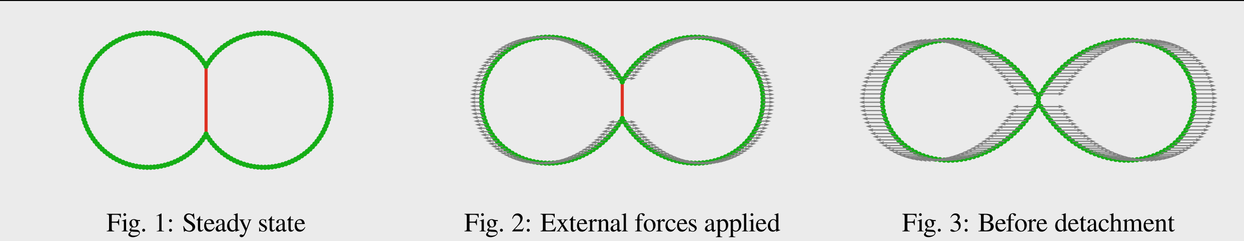


Fig. 1: Steady state   Fig. 2: External forces applied   Fig. 3: Before detachment

A detailed description of the calibration procedure and the corresponding results on tissue fracture timescales are provided in Ref. [3].

5.2 Usage of the DP simulator

In addition to the FV simulator, **PyAFV** includes a `pyafv.calibrate.DeformablePolygonSimulator` class for simulating cell doublets with the DP model, which can be used for calibration as well as standalone analyses.

class DeformablePolygonSimulator(*phys*, *num_vertices=100*)

Simulator for the deformable polygon (DP) model of cell doublets.

Parameters

- **phys** (PhysicalParams) – An instance of PhysicalParams containing the physical parameters, while *phys.r* and *phys.delta* are ignored.
- **num_vertices** (*int*) – Number of vertices M to use for each cell.

Raises

TypeError – If *phys* is not an instance of *PhysicalParams*.

> **Warning**
>
> If the target shape index (based on *phys.P0* and *phys.A0*) indicates a non-circular shape, a **UserWarning** is raised since the DP model is not valid in that regime.

detached	Indicates whether the doublet has detached.
simulate(ext_force, dt, nsteps[, resample_every])	Simulate the DP model for a number of time steps under an external force.
plot_2d([ax, show])	Render a 2D snapshot of the cell doublet in DP model.

Below, we present a minimal example illustrating how the DP simulator is used internally by *pyafv.calibrate.auto_calibrate()*:

```
import matplotlib.pyplot as plt
import pyafv as afv
import pyafv.calibrate as cal


phys = afv.PhysicalParams()

#===============  Auto calibrate ==================
f_detach, phys_cal = cal.auto_calibrate(phys, show=True)

print(f"Detachment force from DP model: {f_detach:.1f}")
print(f"{phys_cal=}")


#===========  Visualize DP simulation =============
"""
    The auto_calibrate process above is equivalent to
    applying a series of external forces until detachment
    and computing the delta value of finite Voronoi model
    that matches the detachment force of DP model.
"""

sim = cal.DeformablePolygonSimulator(phys)

# Initial shape
print(f"{sim.detached=}")
fig, ax = plt.subplots()
```

(continues on next page)

(continued from previous page)

```
sim.plot_2d(ax)
plt.show()
plt.close(fig)

#------------  Apply F = 2 -------------
sim.simulate(ext_force=2.0, dt=1e-3, nsteps=50_000)
print(f"{sim.detached=}")

fig, ax = plt.subplots()
sim.plot_2d(ax)
plt.show()
plt.close(fig)

#------------  Apply F = 4 -------------
sim.simulate(ext_force=4.0, dt=1e-3, nsteps=50_000)
print(f"{sim.detached=}")

fig, ax = plt.subplots()
sim.plot_2d(ax)
plt.show()
plt.close(fig)
```

The generated figures are shown above.

CHAPTER

SIX

CITING PYAFV

arXiv 2604.15481

If you use **PyAFV** in your research, please cite it. A paper associated with **PyAFV** has been posted to **arXiv**. Here is a ready-made BibTeX entry:

```
@article{wang2026divergence,
  title   = {{Divergence of detachment forces in the finite Voronoi model}},
  author  = {Wang, Wei and Camley, Brian A},
  journal = {arXiv preprint arXiv:2604.15481},
  year    = {2026},
  doi     = {10.48550/arXiv.2604.15481}
}
```

CHAPTER

SEVEN

CONTRIBUTING TO PYAFV

First off, **THANK YOU** for considering contributing to **PyAFV**! We welcome contributions from the community.

Before working on a feature or major change, please raise an issue and/or get in touch with the developers. They may have insights on how to implement the feature or useful advice to save you time.

Much of this guide is based on the pyqmc CONTRIBUTING.md, which itself draws from this excellent guide.

7.1 Code of Conduct

By participating in this project, you agree to abide by the PyAFV Code of Conduct.

7.2 How to contribute

7.2.1 Create a fork

Click the "Fork" button on the **PyAFV** *GitHub* page: https://github.com/wwang721/pyafv

Then clone **your fork** to your local machine and enter the repository directory

```
(.venv) $ cd pyafv
```

7.2.2 Set up your development environment

PyAFV uses **uv** for Python package management—a single tool to replace *pip* (⚡10-100x faster), *venv*, and even *conda*.

If you'd like to use your own Python, ensure the `which python` version meets the requirement so **uv** doesn't automatically download a different interpreter; otherwise, I recommend letting **uv** manage everything, including the Python interpreter.

After cloning, install **PyAFV** in editable mode and synchronize dependencies:

```
(.venv) $ uv sync
```

This installs the core package dependencies along with **pytest** required for development and testing.

Note

- You can install additional packages as needed using `uv add <package_name>`.
- In some environments (like HPC clusters), global Python path can contaminate the project environment. You may need to add the `PYTHONPATH=""` prefix to all `uv` commands to isolate the project.
- The current version uses **Cython** to translate `.pyx` files into `.cpp`, (and therefore requires a working C/C++ compiler), though a fallback backend (based on early pure-Python release) is also implemented.
- For *Windows* **MinGW GCC** users (rather than **MSVC**), add a `setup.cfg` file at the repository root

  ```
  # setup.cfg
  [build_ext]
  compiler=mingw32
  ```

  This is equivalent to pass the `--compiler=mingw32` flag when invoking build commands such as `python setup.py build_ext --inplace`. To avoid accidentally committing this *ad hoc* file, do not modify `.gitignore`; instead, add it to local `.git/info/exclude` in the repository, which functions like `.gitignore`.

7.2.3 Create a feature branch and start development

Always branch from `main`, not from another feature branch

```
(.venv) $ git checkout main
(.venv) $ git checkout -b your-feature-name
```

You may then begin editing the codebase and developing new features.

Note

If you modify any `*.pyx` Cython source files, you must reinstall the package to ensure the changes take effect: `uv sync --reinstall-package pyafv --inexact` (the `--inexact` flag prevents **uv** from removing any installed packages).

- If the compiled C/C++ extension is accidentally removed or corrupted (you will see a **RuntimeWarning** about falling back to the pure-Python implementation), you can also reinstall the package.
- For the legacy pure-Python implementation with no C/C++ compiled dependencies, see v0.1.0 (also on GitLab). Starting from **PyAFV** v0.3.4, the pure-Python backend can be selected by passing `backend="python"` when creating the *`pyafv.FiniteVoronoiSimulator`* instance.

7.2.4 Keeping your fork up to date

Add the upstream repository as a remote (do this once):

```
(.venv) $ git remote add upstream https://github.com/wwang721/pyafv.git
(.venv) $ git remote -v
```

To sync with upstream (do this regularly):

```
(.venv) $ git fetch upstream
(.venv) $ git checkout main
(.venv) $ git merge upstream/main
```

If needed, update your feature branch with the latest changes:

```
(.venv) $ git checkout your-feature-name
(.venv) $ git rebase main
```

Note

We use `rebase` to keep the commit history clean.

7.3 Coding standards

1. **Single responsibility:** Keep functions small and focused on one task. Each function should do one thing well.
2. **Avoid Python loops:** Use *numpy* vectorized operations to avoid Python's performance overhead. Operate on batches of data rather than looping. Performance-critical code may be accelerated using *Cython*.
3. **Minimize dependencies**: Avoid adding new libraries unless absolutely necessary. If required, discuss with maintainers first.
4. **Code style**: Follow PEP 8 style guidelines for Python code.

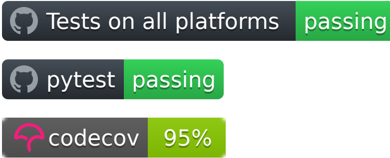
7.4 Documentation requirements

1. **Type annotations:** Use type hints for function arguments and return values.
2. **Array dimensions:** Add comments indicating dimensions for multidimensional arrays:

```
positions = np.zeros((100, 2))  # N x dimension
```

3. **Docstrings:** Each function should have a docstring following PEP 257 and written in either Google style (currently used) or Numpy style so that it can be parsed by *Sphinx* via `sphinx.ext.napoleon`. The docstring should explain:
   - Purpose of the function
   - All input parameters
   - Return values
   - Any exceptions raised

7.5 Writing tests



Tests are located in the `tests/` directory. Run the test suite with

```
(.venv) $ uv run pytest
```

For coverage reports:

```
(.venv) $ uv run pytest --cov
```

Current CI status of the test suite, run via **GitHub Actions** on Python 3.12 (with additional test jobs covering all supported platforms and Python versions), is shown in the badges above.

> **Note**
>
> - A comparison against the **MATLAB** implementation from Ref. [1] is included in current test suite.
> - Unlike v0.1.0, the current test suite is designed to raise errors if the Cython-compiled C/C++ backend is not available, even though a pure-Python fallback implementation is provided and tested.
> - **Pytest** and related plugins (**pytest-cov**, **pytest-benchmark**) are included in the *dev* dependency group and are installed by default when running `uv sync`.

7.5.1 Testing strategies (in order of preference)

1. **Exact solutions:** Compare numerical results to exact analytical solutions.
2. **Independent implementations:** Compare results from two independent numerical methods.
3. **Regression tests:** Ensure the function runs and produces consistent results with pre-computed references.
4. **Sanity checks:** Verify that results make physical sense (e.g., energies decrease after optimization).

7.5.2 Benchmarking

There is also a set of lightweight benchmarks in `tests` using **pytest-benchmark**, e.g., `test_bench_build.py` compares the runtimes of the Cython and pure-Python backends. To run them:

```
(.venv) $ uv run pytest --benchmark-only --benchmark-warmup on --benchmark-histogram
```

This will display the benchmark results and generate an SVG histogram file in the current directory (see an example *here*). You should write benchmarks for any new performance-critical code you add.

7.6 Featured examples

To run current example scripts and notebooks in `examples/`, install all optional dependencies (e.g., **tqdm**, **jupyter**) via `uv sync --extra examples` or `uv sync --all-extras` (add the `--inexact` flag to avoid removing installed packages).

Extra dependencies: `examples`

Package	Minimum version	Usage
tqdm	4.67.1	Progress bars during calculations
jupyter	1.1.0	Jupyter Notebook / JupyterLab
ipywidgets	8.1.5	Jupyter HTML widgets

Then you can simply run the scripts with

```
(.venv) $ uv run <script_name>.py
```

- For developers to launch Jupyter Notebook: after `uv` has synced all extra dependencies, start Jupyter with `uv run jupyter notebook`. Do not use your system-level Jupyter, as the Python kernel of the current `uv` environment is not registered there.

Note

Jupyter notebooks and media are stored via **Git LFS**. If you clone the repository without **Git LFS** installed, these files will appear as small text pointers. You can either install **Git LFS** to fetch them automatically or download the files manually (e.g., download the repository as a ZIP archive) from the **GitHub** web interface.

7.7 Submitting a pull request

Before submitting, ensure you have completed this checklist:

- All new functions are documented with docstrings and type annotations
- Tests are written for the new feature or bug fix
- All tests pass: `uv run pytest`
- Code follows the coding standards above
- Relevant documentation is updated (README, examples, etc.)
- The branch is up to date with `main`

7.7.1 Pull request process

1. Push your feature branch to your fork:

```
(.venv) $ git push origin your-feature-name
```

2. Go to the PyAFV repository and click "Pull Request".
3. In your pull request description:
   - Clearly describe the new feature or bug fix
   - Reference any related issues (e.g., "Fixes #123")
   - For bug fixes, provide a minimal example demonstrating the bug and how the fix resolves it
   - For new features, explain the use case and provide example usage
4. Be responsive to feedback from reviewers and be prepared to make changes.

7.8 Reporting issues

When reporting bugs or requesting features:

1. **Search existing issues** to avoid duplicates
2. **Use a clear title** that describes the problem
3. **Provide details:**
   - For bugs: steps to reproduce, expected vs. actual behavior, error messages, environment details

    - For features: use case, proposed implementation (if any)

4. **Include code examples** when relevant (minimal reproducible examples are best)

7.9 Code review process

All submissions require review before merging. Reviewers will check:

- Code quality and adherence to coding standards
- Test coverage and quality
- Documentation completeness
- Performance implications
- Compatibility with existing code

7.10 Questions?

If you have questions about contributing, feel free to:

- Open an issue on GitHub
- Start a discussion in GitHub Discussion
- Contact the maintainer via email: ww000721@gmail.com

Thank you for helping make **PyAFV** better!

CHAPTER

EIGHT

API REFERENCE

8.1 pyafv

PyAFV - A Python implementation of the **active finite Voronoi (AFV) model** in 2D.

Classes

PhysicalParams([r, A0, P0, KA, KP, Lambda, ...])	Physical parameters for the active finite Voronoi (AFV) model.
FiniteVoronoiSimulator(pts, phys[, backend])	Simulator for the active finite Voronoi (AFV) model.
ParallelFiniteVoronoiSimulator(pts, phys[, ...])	Python multiprocessing domain-decomposed simulator for the AFV model.

Functions

visualize_2d(pts, diag, r[, ax, selected])	Visualize a 2D snapshot using the diagnostic dictionary *diag* generated by pyafv.FiniteVoronoiSimulator.build().
visualize_2d_parallel(pts, diag, r[, ax, ...])	Visualize a 2D snapshot from pyafv.ParallelFiniteVoronoiSimulator.build().
target_delta(params, target_force)	Given the physical parameters and a target detachment force, compute the corresponding delta.

Subpackages

calibrate	Subpackage for calibrating the detachment forces against the deformable polygon (DP) model.

Experimental APIs (use with caution)

tile_pbc(pts, L[, r])	Periodic tiling of pts (N,2), with bookkeeping.
rebuild_connection_matrix(N, connect)	Build a symmetric sparse adjacency matrix from a cell-cell edge list.
select_daughter_cluster(N, connect)	Randomly pick one connected component ("daughter cluster") from the connectivity graph.

continues on next page

Table 4 – continued from previous page

get_cluster_sizes(N, connect)	Compute the sizes of all connected components in the connectivity graph.
DomainDecomposition(domain_id, grid_ix, ...)	Point-only data for one spatial domain plus its halo.
decompose_points(points[, grid_shape, ...])	Decompose points into owned grid domains plus halo/local points.

8.2 pyafv.PhysicalParams

class PhysicalParams(*r=1.0*, *A0=3.141592653589793*, *P0=4.8*, *KA=1.0*, *KP=1.0*, *Lambda=0.2*, *delta=None*)

Physical parameters for the active finite Voronoi (AFV) model.

Warning

- **Frozen dataclass** is used for `PhysicalParams` to ensure immutability of instances.
- Do not set `delta` unless you know what you are doing.

Parameters

- **r** (*float*) – Radius (maximal) of the Voronoi cells, sometimes denoted as ℓ.
- **A0** (*float*) – Preferred area of the Voronoi cells.
- **P0** (*float*) – Preferred perimeter of the Voronoi cells.
- **KA** (*float*) – Area elasticity constant.
- **KP** (*float*) – Perimeter elasticity constant.
- **Lambda** (*float*) – Tension difference between non-contacting edges and contacting edges.
- **delta** (*float | None*) – Contact truncation threshold to avoid singularities in computations; if None, set to 0.45*r.

Attributes

`r`	Radius (maximal) of the Voronoi cells, sometimes denoted as ℓ.
`A0`	Preferred area of the Voronoi cells.
`P0`	Preferred perimeter of the Voronoi cells.
`KA`	Area elasticity constant.
`KP`	Perimeter elasticity constant.
`Lambda`	Tension difference between non-contacting edges and contacting edges.
`delta`	Contact truncation threshold to avoid singularities in computations.

Methods

`get_steady_state`()	Search for the steady-state (ℓ, d) of a cell doublet for the given physical parameters (by minimizing total energy).
`with_optimal_radius`([digits, delta])	Returns a new instance of `PhysicalParams` with the maximum radius ℓ (or `r`) updated to the steady state value ℓ_0 of cell doublets.
`replace`(**changes)	Returns a new instance of `PhysicalParams` with specified fields replaced by new values.

8.2.1 pyafv.PhysicalParams.get_steady_state

PhysicalParams.**get_steady_state**()

Search for the steady-state (ℓ, d) of a cell doublet for the given physical parameters (by minimizing total energy).

Returns

Steady-state (optimal) (ℓ_0, d_0) values.

Return type

tuple[*float*, *float*]

Note

ℓ is the maximal cell radius, and $2d$ is the cell-center distance of a doublet (rather than d).

8.2.2 pyafv.PhysicalParams.with_optimal_radius

PhysicalParams.**with_optimal_radius**(*digits=None*, *delta=None*)

Returns a new instance of `PhysicalParams` with the maximum radius ℓ (or `r`) updated to the steady state value ℓ_0 of cell doublets. Other parameters (except `delta`) remain unchanged.

Basically a wrapper around `get_steady_state()` + creating a new instance.

Parameters

- **digits** (*int | None*) – If not None, round the optimal radius ℓ_0 to the specified number of decimal places. Only intended for randomized tests to avoid floating-point precision issues.
- **delta** (*float | None*) – If not None, set the contact truncation threshold `delta` to this value in the returned instance.

Returns

New instance with optimal radius.

Return type

PhysicalParams

Important

In the returned instance, the contact truncation threshold `delta` is set to 0.45*r by default.

8.2.3 pyafv.PhysicalParams.replace

`PhysicalParams.replace(**changes)`

Returns a new instance of `PhysicalParams` with specified fields replaced by new values.

Parameters
`**changes` (*float | None*) – Field names and their new values to be replaced.

Returns
New instance with the updated fields.

Return type
PhysicalParams

Hint

This is a convenience method wrapping `dataclasses.replace()`, e.g., to change `A0` and `delta` of an existing instance *phys*: `phys_new = phys.replace(A0=5.0, delta=0.3)`.

8.3 pyafv.FiniteVoronoiSimulator

class FiniteVoronoiSimulator(*pts*, *phys*, *backend=None*)

Simulator for the active finite Voronoi (AFV) model.

This class provides an interface to simulate the finite Voronoi model. It wraps around the two backend implementations, which may be either a Cython-accelerated version or a pure Python fallback.

Parameters

- **`pts`** (*ndarray*) – (N,2) array of initial cell center positions.
- **`phys`** (`PhysicalParams`) – Physical parameters used within this simulator.
- **`backend`** (*Literal['cython', 'python'] | None*) – Optional, specify "python" to force the use of the pure Python fallback implementation. Otherwise, the "cython" backend is used.

Raises

- **`ValueError`** – If *pts* does not have shape (N,2).
- **`TypeError`** – If *phys* is not an instance of `PhysicalParams`.

Warning

If the Cython backend cannot be imported (unless *backend* is set to "python"), a **RuntimeWarning** is raised and the pure Python implementation is used instead.

Attributes

preferred_areas	Preferred areas $\{A_{0,i}\}$ for all cells (read-only).

8.3.1 pyafv.FiniteVoronoiSimulator.preferred_areas

property FiniteVoronoiSimulator.**preferred_areas: ndarray**

Preferred areas $\{A_{0,i}\}$ for all cells (read-only).

Returns
A copy of the internal preferred area array.

Return type
ndarray

Methods

build([connect])	Build the finite Voronoi structure and compute forces, returning a dictionary of diagnostics.
plot_2d([ax, show])	Build the finite Voronoi structure and render a 2D snapshot.
update_params(phys)	Update physical parameters.
update_positions(pts[, A0])	Update cell center positions.
update_preferred_areas(A0)	Update the preferred areas for all cells.
_build_voronoi_with_extensions([joggle])	Build standard Voronoi structure for current points.
_per_cell_geometry(vor, vertices_all, ...)	Build the finite Voronoi per-cell geometry and energy contributions.

8.3.2 pyafv.FiniteVoronoiSimulator.build

FiniteVoronoiSimulator.**build**(*connect=True*)

Build the finite Voronoi structure and compute forces, returning a dictionary of diagnostics.

Do the following:

- Build Voronoi (+ extensions)
- Get cell connectivity
- Compute per-cell quantities and derivatives
- Assemble forces

Parameters
connect (*bool*) – Whether to compute cell connectivity information. Setting this to `False` saves some computation time (though very marginal) when connectivity is not needed.

Returns

A dictionary containing forces and geometric properties with keys:

- **forces**: (N,2) array of forces on cell centers
- **areas**: (N,) array of cell areas
- **perimeters**: (N,) array of cell perimeters
- **arclens**: (N,) array of non-contacting edge (arc) lengths per cell
- **coord_nums**: (N,) integer array of coordination numbers per cell
- **vertices**: (M,2) array of all Voronoi + extension vertices
- **edges_type**: List-of-lists of edge types per cell (1=straight, 0=circular arc)

- **regions**: List-of-lists of vertex indices per cell
- **connections**: (K,2) array of connected cell index pairs

Return type
dict[*str*, *object*]

8.3.3 pyafv.FiniteVoronoiSimulator.plot_2d

`FiniteVoronoiSimulator.`**`plot_2d`**`(`*ax=None*, *show=False*, ***kw*`)`

Build the finite Voronoi structure and render a 2D snapshot. The plotting style follows `scipy.spatial.voronoi_plot_2d()`.

This method is basically a wrapper of `_build_voronoi_with_extensions()` and `_per_cell_geometry()` functions + plot.

Parameters

- **`ax`** (`Axes | None`) – If provided, draw into the axes; otherwise get the current axes.
- **`show`** (`bool`) – Whether to call `plt.show()` at the end.
- **`show_points`** (`bool, optional`) – Add cell center points to the plot, default *True*.
- **`point_size`** (`float, optional`) – Specifies the marker size for the points, default 2.
- **`show_inner_vertices`** (`bool, optional`) – Add inner vertices to the plot, default *False*.
- **`show_outer_vertices`** (`bool, optional`) – Add outer vertices to the plot, default *False*.
- **`line_color_in`** (`str, optional`) – Specifies the color for contact edges (and inner vertices), default 'b'.
- **`line_color_out`** (`str, optional`) – Specifies the color for non-contact edges (and outer vertices), default 'C6'.
- **`line_width`** (`float, optional`) – Specifies the line width for cell boundaries, default 1.5.
- **`line_alpha`** (`float, optional`) – Specifies the line alpha for cell boundaries, default 1.0.
- **`show_voronoi`** (`bool, optional`) – Add the Voronoi edges to the plot, default *True*.

Returns
The matplotlib axes containing the plot.

Return type
Axes

8.3.4 pyafv.FiniteVoronoiSimulator.update_params

`FiniteVoronoiSimulator.`**`update_params`**`(`*phys*`)`

Update physical parameters.

Parameters
`phys` (`PhysicalParams`) – New `PhysicalParams` object.

Raises
`TypeError` – If *phys* is not an instance of `PhysicalParams`.

Return type
None

Warning

This also resets all preferred cell areas to the new value of *A0*.

8.3.5 pyafv.FiniteVoronoiSimulator.update_positions

`FiniteVoronoiSimulator.`**`update_positions`**(*pts*, *A0=None*)

Update cell center positions.

Note

If the number of cells changes, the preferred areas for all cells are reset to the default value—defined either at simulator instantiation or by `update_params()`—unless *A0* is explicitly specified.

Parameters

- **`pts`** (*`ndarray`*) – New cell center positions.
- **`A0`** (*`float | ndarray | None`*) – Optional, set new preferred area(s).

Raises

- **`ValueError`** – If *pts* does not have shape (N,2).
- **`ValueError`** – If *A0* is an array and does not have shape (N,).

Return type
None

8.3.6 pyafv.FiniteVoronoiSimulator.update_preferred_areas

`FiniteVoronoiSimulator.`**`update_preferred_areas`**(*A0*)

Update the preferred areas for all cells.

Parameters
`A0` (*`float | ndarray`*) – New preferred area(s) for all cells, either as a scalar or as an array of shape (N,).

Raises
`ValueError` – If *A0* does not match cell number.

Return type
None

8.3.7 pyafv.FiniteVoronoiSimulator._build_voronoi_with_extensions

`FiniteVoronoiSimulator.`**`_build_voronoi_with_extensions`**(*joggle=False*)

Build standard Voronoi structure for current points.

For N<=2, emulate regions. For N>=3, extend infinite ridges, add extension vertices, and update regions accordingly. Return the augmented structures.

Warning

This is an internal method. Use with caution.

Parameters
`joggle` (*bool*) – Whether to joggle input points (N>=3) slightly to avoid precision issues (e.g., collinearities, co-circularities).

Returns

A *tuple* containing:

- **vor**: SciPy Voronoi object for current points with extensions.
- **vertices_all**: (M,2) array of all Voronoi vertices including extensions.
- **ridge_vertices_all**: (R,2) array of vertex indices for each ridge, including extensions.
- **num_vertices**: Number of Voronoi vertices before adding extension.
- **vertexpair2ridge**: *dict* mapping vertex index pairs to ridge index.
- **vertex_points**: *dict* mapping vertex index to list of associated point indices.

Return type
tuple[*Voronoi*, *ndarray*, *ndarray*, *int*, *dict*[*tuple*[*int*,*int*], *int*], *dict*[*int*, *list*[*int*]]]

8.3.8 pyafv.FiniteVoronoiSimulator._per_cell_geometry

`FiniteVoronoiSimulator._per_cell_geometry`(*vor*, *vertices_all*, *ridge_vertices_all*, *num_vertices*, *vertexpair2ridge*)

Build the finite Voronoi per-cell geometry and energy contributions.

Iterate each cell to:

- sort polygon/arc vertices around each cell
- classify edges (1 = straight Voronoi edge; 0 = circular arc)
- compute area/perimeter for each cell
- accumulate derivatives w.r.t. vertices (dA_poly/dh, dP_poly/dh)
- register "outer" vertices created at arc intersections and track their point pairs

Warning

This is an internal method. Use with caution.

Parameters

- `vor` (*Voronoi*) – SciPy Voronoi object for current points with extensions.
- `vertices_all` (*ndarray*) – (M,2) array of all Voronoi vertices including extensions.
- `ridge_vertices_all` (*ndarray*) – (R,2) array of vertex indices for each ridge.
- `num_vertices` (*int*) – Number of Voronoi vertices before adding extension.
- `vertexpair2ridge` (*dict[tuple[int, int], int]*) – *dict* mapping vertex index pairs to ridge index.

Returns

A diagnostics dictionary containing:

- **vertex_in_id**: *set* of inner vertex ids.
- **vertex_out_id**: *set* of outer vertex ids.
- **vertices_out**: (L,2) array of outer vertex coordinates.
- **vertex_out_points**: (L,2) array of point index pairs associated with each outer vertex.
- **vertex_out_da_dtheta**: Array of dA/dtheta for all outer vertices.
- **vertex_out_dl_dtheta**: Array of dL/dtheta for all outer vertices.
- **dA_poly_dh**: Array of dA_polygon/dh for each vertex.
- **dP_poly_dh**: Array of dP_polygon/dh for each vertex.
- **area_list**: Array of polygon areas for each cell.
- **perimeter_list**: Array of polygon perimeters for each cell.
- **arclen_list**: Array of non-contacting arc lengths for each cell.
- **point_edges_type**: List of lists of edge types per cell.
- **point_vertices_f_idx**: List of lists of vertex ids per cell.
- **num_vertices_ext**: Number of vertices including infinite extension vertices.
- **coord_nums**: (N,) integer array of coordination numbers per cell.

Return type

dict[*str*, *object*]

8.4 pyafv.ParallelFiniteVoronoiSimulator

class ParallelFiniteVoronoiSimulator(*pts*, *phys*, *grid_shape=(2, 2)*, *n_workers=None*, ***, *halo_width=None*, *backend=None*, *domain_bounds=None*, *decomposition_method='dense'*)

Python multiprocessing domain-decomposed simulator for the AFV model.

This class decomposes the full point set into rectangular domains with halo regions, then composes `FiniteVoronoiSimulator` on each local subdomain. The returned diagnostics are merged back into global cell indexing for owned cells. When `n_workers > 1`, local subdomain builds use Python worker processes.

Parameters

- **pts** (*ndarray*) – (N,2) array of initial cell center positions.
- **phys** (`PhysicalParams`) – Physical parameters used within this simulator.
- **grid_shape** (*tuple[int, int]*) – Number of domains in the x and y directions.
- **n_workers** (*int | None*) – Number of worker processes. If *None*, use `os.cpu_count()`.
- **halo_width** (*float | None*) – Width of the halo region added to each domain. If *None*, use `4.01 * phys.r`.
- **backend** (*Literal['cython', 'python'] | None*) – Optional, specify "python" to force the use of the pure Python fallback implementation inside each local finite Voronoi simulator. Otherwise, the "cython" backend is used.

- **domain_bounds** (*tuple[tuple[float, float], tuple[float, float]] | None*) – Optional domain bounds as `((xmin, xmax), (ymin, ymax))`. If *None*, bounds are inferred from the current point positions.
- **decomposition_method** (*Literal['dense', 'binned', 'sorted_x']*) – Method used to collect halo points. `"dense"` is usually faster for moderate systems. `"binned"` can be faster for many domains. `"sorted_x"` uses less temporary memory.

Raises

- **ValueError** – If *pts* does not have shape (N,2).
- **ValueError** – If *grid_shape*, *halo_width*, *domain_bounds*, or *decomposition_method* is invalid.
- **TypeError** – If *phys* is not an instance of *PhysicalParams*.

> **Note**
>
> For repeated calls with `n_workers > 1`, put the loop inside the context manager so worker processes are created once and reused across build steps:

```
with sim:       # sim is an instance of ParallelFiniteVoronoiSimulator
    for step in range(num_steps):
        diag = sim.build()
```

Attributes

preferred_areas	Return a copy of the preferred area array.

8.4.1 pyafv.ParallelFiniteVoronoiSimulator.preferred_areas

property ParallelFiniteVoronoiSimulator.**preferred_areas**: **ndarray**

Return a copy of the preferred area array.

Returns
(N,) array of preferred cell areas.

Return type
ndarray

Methods

build([connect, plot_mode])	Build local finite Voronoi structures and merge global diagnostics.
update_params(phys)	Update physical parameters.
update_positions(pts[, A0])	Update cell center positions.
update_preferred_areas(A0)	Update the preferred areas for all cells.

8.4.2 pyafv.ParallelFiniteVoronoiSimulator.build

`ParallelFiniteVoronoiSimulator.`**`build`**(*connect=False*, *plot_mode=False*)

Build local finite Voronoi structures and merge global diagnostics.

Do the following:

- Decompose cell centers into owned domains plus halo points
- Build a finite Voronoi diagram in each local subdomain
- Extract diagnostics for owned cells
- Merge forces and geometric quantities back into global indexing
- Optionally collect global cell connectivity and per-domain plot data

Parameters

- **`connect`** (*`bool`*) – Whether to compute cell connectivity information. Setting this to `False` saves some computation time when connectivity is not needed. Note that the default is `False`, unlike *`FiniteVoronoiSimulator.build()`*, where the default is `True`.
- **`plot_mode`** (*`bool`*) – Whether to include per-domain plotting diagnostics. If `True`, use *`visualize_2d_parallel()`* for visualization.

Returns

A dictionary containing merged forces and geometric properties with keys:

- **forces**: (N,2) array of forces on cell centers
- **areas**: (N,) array of cell areas
- **perimeters**: (N,) array of cell perimeters
- **arclens**: (N,) array of non-contacting edge (arc) lengths per cell
- **coord_nums**: (N,) integer array of coordination numbers per cell
- **connections**: (K,2) array of connected global cell index pairs
- **pids**: (D,) integer array of process ids used for each domain
- **plot_mode**: Whether per-domain plot diagnostics are included

If *plot_mode* is `True`, the dictionary also contains:

- **owned_global_ids**: List of global cell ids owned by each domain
- **diag_plot**: List of local plotting dictionaries, each with `vertices`, `edges_type`, and `regions` for owned cells

Return type

dict[*str*, *object*]

8.4.3 pyafv.ParallelFiniteVoronoiSimulator.update_params

`ParallelFiniteVoronoiSimulator.`**`update_params`**(*phys*)

Update physical parameters.

Parameters

`phys` (`PhysicalParams`) – New *`PhysicalParams`* object.

Raises
`TypeError` – If *phys* is not an instance of `PhysicalParams`.

Return type
None

Warning

This also resets all preferred cell areas to the new value of *A0*. If *halo_width* was not explicitly specified at instantiation, it also updates the halo width to `4.01 * phys.r`.

8.4.4 pyafv.ParallelFiniteVoronoiSimulator.update_positions

`ParallelFiniteVoronoiSimulator.`**`update_positions`**`(`*pts*, *A0=None*`)`

Update cell center positions.

Note

If the number of cells changes, the preferred areas for all cells are reset to the default value—defined either at simulator instantiation or by `update_params()`—unless *A0* is explicitly specified.

Parameters

- **`pts`** (`ndarray`) – New cell center positions.
- **`A0`** (`float | ndarray | None`) – Optional, set new preferred area(s).

Raises

- `ValueError` – If *pts* does not have shape (N,2).
- `ValueError` – If *A0* is an array and does not have shape (N,).

Return type
None

8.4.5 pyafv.ParallelFiniteVoronoiSimulator.update_preferred_areas

`ParallelFiniteVoronoiSimulator.`**`update_preferred_areas`**`(`*A0*`)`

Update the preferred areas for all cells.

Parameters
`A0` (`float | ndarray`) – New preferred area(s) for all cells, either as a scalar or as an array of shape (N,).

Raises
`ValueError` – If *A0* does not match cell number.

Return type
None

8.5 pyafv.visualize_2d

visualize_2d(*pts*, *diag*, *r*, *ax=None*, ***, *selected=None*, ***kw*)

Visualize a 2D snapshot using the diagnostic dictionary *diag* generated by `pyafv.FiniteVoronoiSimulator.build()`. This is basically a wrapper around the vectorized custom plotting functions from the example notebooks and generally preferred over the original `pyafv.FiniteVoronoiSimulator.plot_2d()` method. The plotting style follows `scipy.spatial.voronoi_plot_2d()`.

Note

If you are visualizing a *diag* from `ParallelFiniteVoronoiSimulator.build()`, you should use `visualize_2d_parallel()` instead.

Parameters

- **pts** (*ndarray*) – An (N, 2) array of point coordinates.
- **diag** (*dict[str, object]*) – A diagnostic *dict* containing Voronoi diagram information.
- **r** (*float*) – Maximum radius (or denoted as ℓ) used for drawing arcs.
- **ax** (*Axes | None*) – If provided, draw into the axes; otherwise create a new one.
- **selected** (*array-like | None, optional*) – Cells to draw. This can be either a one-dimensional array of integer indices or a boolean mask with length equal to `len(pts)`. If *None*, draw all cells.
- **cell_colors** (*color or list, optional*) – A single color or a sequence of colors for filling cells, default 'C2'. Use *None* for no fill. If *selected* is provided, full-length per-cell color sequences are sliced to the selected cells.
- **fill_alpha** (*float, optional*) – Specifies the alpha for cell fills, default 0.1.
- **fill_zorder** (*float, optional*) – Specifies the z-order for cell fills, default 0.
- **show_points** (*bool, optional*) – Add cell center points to the plot, default *False*.
- **point_size** (*float, optional*) – Specifies the marker area for the points, default 4. If *selected* is provided, full-length per-cell sequences are sliced to the selected cells.
- **point_colors** (*color or list, optional*) – A single color or a sequence of colors for the points, default 'C0'. If *selected* is provided, full-length per-cell color sequences are sliced to the selected cells.
- **point_zorder** (*float, optional*) – Specifies the z-order for the points, default 3.
- **straight_colors** (*color | None, optional*) – Color for straight contact edges, default 'C0'. Use *None* to skip drawing straight edges.
- **straight_lw** (*float, optional*) – Line width for straight edges, default 1.0.
- **straight_alpha** (*float, optional*) – Alpha for straight edges, default 1.0.
- **straight_capstyle** (*str, optional*) – Cap style for straight edges, default 'butt'.
- **straight_zorder** (*float, optional*) – Z-order for straight edges, default 2.
- **arc_colors** (*color | None, optional*) – Color for arc non-contact edges, default 'C2'. Use *None* to skip drawing arc edges.
- **arc_lw** (*float, optional*) – Line width for arc edges, default 1.0.

- **arc_alpha** (*float, optional*) – Alpha for arc edges, default 1.0.
- **arc_capstyle** (*str, optional*) – Cap style for arc edges, default 'butt'.
- **arc_zorder** (*float, optional*) – Z-order for arc edges, default 1.
- **auto_adjust_bounds** (*bool, optional*) – Whether to automatically adjust the plot bounds to fit the diagram, default *True*.

Returns
The matplotlib figure object representing the entire canvas.

Return type
Figure

8.6 pyafv.visualize_2d_parallel

visualize_2d_parallel(*pts*, *diag*, *r*, *ax=None*, ***, *selected=None*, ***kw*)

Visualize a 2D snapshot from `pyafv.ParallelFiniteVoronoiSimulator.build()`. Note that "_parallel" here means that *diag* is generated by the parallel simulator, not that the visualization itself is parallelized with Python multiprocessing (though we have vectorized the drawing of individual domains).

The diagnostic dictionary must be built with `plot_mode=True`. Each domain is drawn using `visualize_2d()`, and the global axes bounds are adjusted once at the end.

Parameters

- **pts** (*ndarray*) – An (N, 2) array of point coordinates.
- **diag** (*dict[str, object]*) – A diagnostic *dict* containing Voronoi diagram information.
- **r** (*float*) – Maximum radius (or denoted as ℓ) used for drawing arcs.
- **ax** (*Axes | None*) – If provided, draw into the axes; otherwise create a new one.
- **selected** (*array-like | None, optional*) – Global cells to draw. This can be either a one-dimensional array of integer indices or a boolean mask with length equal to `len(pts)`. If *None*, draw all owned cells from every domain.
- ****kw** – Additional keyword arguments passed to `visualize_2d()`.

Returns
The matplotlib figure object representing the entire canvas.

Return type
Figure

8.7 pyafv.target_delta

target_delta(*params*, *target_force*)

Given the physical parameters and a target detachment force, compute the corresponding delta.

Parameters

- **params** (`PhysicalParams`) – Physical parameters of the AFV model.
- **target_force** (*float*) – Target detachment force.

Raises

- **TypeError** – If *params* is not an instance of `PhysicalParams`.

- **ValueError** – If the target force is not within the achievable range.

Returns
Corresponding value of the truncation threshold δ.

Return type
float

> **Note**
>
> We search for the cell-cell separation at which the intercellular force equals the target force, scanning distances from $10^{-6}\ell$ to $(2-10^{-6})\ell$ in steps of $10^{-6}\ell$, and select the **largest distance** at which the match occurs.

8.8 pyafv.calibrate

Subpackage for calibrating the detachment forces against the deformable polygon (DP) model.

Core functions

auto_calibrate(phys[, ext_forces, dt, ...])	Auto-calibrate the parameters *phys* against the deformable polygon (DP) model.

Classes

DeformablePolygonSimulator(phys[, num_vertices])	Simulator for the deformable polygon (DP) model of cell doublets.

Tool functions

polygon_centroid(pts)	Centroid (center of mass) of a simple polygon with uniform density.
polygon_area_perimeter(pts)	Compute the area and perimeter for a Counter-ClockWise (CCW) polygon.
resample_polyline(pts[, M])	Resample an open polyline to M points with uniform arclength spacing.

8.8.1 pyafv.calibrate.auto_calibrate

auto_calibrate(*phys*, *ext_forces=None*, *dt=0.001*, *nsteps=50000*, *show=None*)

Auto-calibrate the parameters *phys* against the deformable polygon (DP) model.

In this calibration, we simulate an initially steady-state cell doublet under increasing external force dipoles using the DP model; the external force starts from 0 to max(*ext_forces*). We identify the detachment force as the first external force at which detachment occurs. We then search for the `delta` value in the finite Voronoi (FV) model to match this detachment force.

Parameters

- **phys** (`PhysicalParams`) – The initial physical parameters.

- **ext_forces** (*ndarray | None*) – An array of external forces to apply during calibration; defaults to None, which uses `np.linspace(0, 10, 101)[1:]`; should start from a small positive value.
- **dt** (*float*) – Time step for each simulation step.
- **nsteps** (*int*) – Number of simulation steps to run for each external force.
- **show** (*bool | None*) – Whether to print progress information; no need to set it if **tqdm** is installed.

Raises

TypeError – If *phys* is not an instance of *PhysicalParams*.

Returns

A *tuple* containing the detachment force and the calibrated *PhysicalParams*. If detachment does not occur within the given force range, return a *NaN* force.

Return type

tuple[*float*, PhysicalParams]

Warning

This function may take some time to run, depending on the parameters. (If **tqdm** is installed, a progress bar will be shown automatically.)

Do not change defaults unless you understand the implications. If you only need a rough or faster calibration, you may change the external force range or interval. Adjusting *dt* and *nsteps* may also speed up simulations, but may affect accuracy; test the `DeformablePolygonSimulator` model separately to ensure accuracy is acceptable.

8.8.2 pyafv.calibrate.DeformablePolygonSimulator

class DeformablePolygonSimulator(*phys*, *num_vertices=100*)

Simulator for the deformable polygon (DP) model of cell doublets.

Parameters

- **phys** (`PhysicalParams`) – An instance of PhysicalParams containing the physical parameters, while *phys.r* and *phys.delta* are ignored.
- **num_vertices** (*int*) – Number of vertices M to use for each cell.

Raises

TypeError – If *phys* is not an instance of *PhysicalParams*.

Warning

If the target shape index (based on *phys.P0* and *phys.A0*) indicates a non-circular shape, a **UserWarning** is raised since the DP model is not valid in that regime.

Attributes

pts1: ndarray

(N,2) array of vertices in cell 1.

pts2: ndarray

(N,2) array of vertices in cell 2.

contact_length: float

Current contact length.

detach_criterion: float

Contact length at which detachment occurs; defaults to $2\pi\ell_0/M$.

detached: bool

Indicates whether the doublet has detached.

Methods

simulate(*ext_force*, *dt*, *nsteps*, *resample_every=1000*)

Simulate the DP model for a number of time steps under an external force.

This is basically a wrapper around `_step_update()` and `resample_polyline()` with some bookkeeping.

Parameters

- **ext_force** (*float*) – The external force applied to the cell doublet.
- **dt** (*float*) – Time step size.
- **nsteps** (*int*) – Number of time steps to simulate.
- **resample_every** (*int*) – How often (in steps) to resample the polygon vertices for uniform spacing.

Return type

None

plot_2d(*ax=None*, *show=False*, ***kw*)

Render a 2D snapshot of the cell doublet in DP model.

Parameters

- **ax** (*Axes | None*) – If provided, draw into the axes; otherwise get the current axes.
- **show** (*bool*) – Whether to call `plt.show()` at the end.
- **line_color_in** (*str, optional*) – Specifies the color for the contact edge.
- **line_color_out** (*str, optional*) – Specifies the color for the vertices and non-contact edges.
- **line_width_in** (*float, optional*) – Specifies the line width for the contact edge.
- **line_width_out** (*float, optional*) – Specifies the line width for the non-contact edges.
- **point_size** (*float, optional*) – Specifies the marker size for the vertices.

Returns

The matplotlib axes containing the plot.

Return type

Axes

_step_update(*ext_force*, *dt*)

Single simulation step under external force.

Parameters

- **ext_force** (*float*) – The external force applied to the cell doublet.
- **dt** (*float*) – Time step size.

Return type
None

> **Warning**
>
> This is an internal method. Use with caution. We have implicitly assumed that the vertex mobility is $\mu = 1$ so that $\Delta x = F\Delta t$.

8.8.3 pyafv.calibrate tools

polygon_centroid(*pts*)

Centroid (center of mass) of a simple polygon with uniform density.

Parameters
pts (*ndarray*) – (N,2) array of vertices in order (first need not repeat at end).

Returns
(2,) centroid array. (If degenerate area, returns vertex mean.)

Return type
ndarray

polygon_area_perimeter(*pts*)

Compute the area and perimeter for a Counter-ClockWise (CCW) polygon.

Parameters
pts (*ndarray*) – (N,2) array of vertices in CCW order (first need not repeat at end).

Returns
A *tuple* of (area, perimeter).

Return type
tuple[*float*, *float*]

resample_polyline(*pts*, *M=None*)

Resample an open polyline to M points with uniform arclength spacing. Keeps endpoints exactly. If M is None, uses original number of points.

Parameters

- **pts** (*ndarray*) – (N,2) array of polyline vertices.
- **M** (*int | None*) – Number of output points; if None, uses N.

Returns
(M,2) array of resampled polyline vertices.

Return type
ndarray

8.9 Experimental APIs

Warning

These functions are still under development and may be removed or changed without deprecation. Use with caution.

Periodic boundary condition utilities

tile_pbc(*pts*, *L*, *r=None*)

Periodic tiling of pts (N,2), with bookkeeping.

Parameters

- **pts** (*ndarray*) – (N,2) original positions.
- **L** (*float*) – Box size for periodic boundary conditions.
- **r** (*float | None*) – Maximum radius (or denoted as ℓ) used to determine the tiling range (at least 4ℓ, which ensures both correct cell shapes and forces, whereas 2ℓ only ensures correct cell shapes); if None, defaults to L.

Returns

An (M,2) array containing the tiled positions and an (M,) array containing indices mapping each tiled point back to its original index in 0..N-1.

Return type

tuple[*ndarray*, *ndarray*]

Connectivity utilities

rebuild_connection_matrix(*N*, *connect*)

Build a symmetric sparse adjacency matrix from a cell-cell edge list.

Parameters

- **N** (*int*) – Total number of cells.
- **connect** (*ndarray*) – (E,2) edge list, where each row is a pair of cell indices.

Returns

An (N,N) boolean adjacency matrix, symmetrized so that both `(i,j)` and `(j,i)` are set for every input edge.

Return type

csr_matrix

select_daughter_cluster(*N*, *connect*)

Randomly pick one connected component (“daughter cluster”) from the connectivity graph.

Parameters

- **N** (*int*) – Total number of cells.
- **connect** (*ndarray*) – (E,2) edge list, where each row is a pair of cell indices.

Returns

A *tuple* containing: an (N_sub,) `numpy.ndarray` of global indices for cells in the chosen cluster, the cluster size `N_sub`, and the (E_sub,2) `numpy.ndarray` of edges re-indexed to local `0..N_sub-1`. If the graph has only one connected component, returns `(None, N, connect)` unchanged.

Return type

tuple[*ndarray* | *None*, *int*, *ndarray*]

get_cluster_sizes(*N*, *connect*)

Compute the sizes of all connected components in the connectivity graph.

Parameters

- **N** (*int*) – Total number of cells.
- **connect** (*ndarray*) – (E,2) edge list, where each row is a pair of cell indices.

Returns

A *tuple* containing: an (n_components,) `numpy.ndarray` of component sizes, the number of connected components, and an (N,) `numpy.ndarray` of component labels for each cell.

Return type

tuple[*ndarray*, *int*, *ndarray*]

Domain decomposition utilities

class DomainDecomposition(*domain_id*, *grid_ix*, *grid_iy*, *x_range*, *y_range*, *halo_x_range*, *halo_y_range*, *local_global_ids*, *owned_local_ids*, *local_pts*)

Point-only data for one spatial domain plus its halo.

This object stores the index bookkeeping needed to relate a local domain calculation back to the original global point array. It contains no finite Voronoi or AFV-specific data.

Parameters

- **domain_id** (*int*) – Integer id of this spatial domain.
- **grid_ix** (*int*) – Domain index in the x direction.
- **grid_iy** (*int*) – Domain index in the y direction.
- **x_range** (*tuple[float, float]*) – Owned-domain x interval.
- **y_range** (*tuple[float, float]*) – Owned-domain y interval.
- **halo_x_range** (*tuple[float, float]*) – Local-domain x interval after expanding by the halo width.
- **halo_y_range** (*tuple[float, float]*) – Local-domain y interval after expanding by the halo width.
- **local_global_ids** (*ndarray*) – Global point ids included in this domain's local box, including halo points.
- **owned_local_ids** (*ndarray*) – Local indices of points uniquely owned by this domain.
- **local_pts** (*ndarray*) – Local point coordinates, equivalent to `points[local_global_ids]`.

decompose_points(*points*, *grid_shape=(2, 2)*, *halo_width=0.0*, *, *domain_bounds=None*, *method='dense'*)

Decompose points into owned grid domains plus halo/local points.

Points on internal owned-domain boundaries are assigned to exactly one domain by using right-sided binning and clipping at the outermost boundary. Halo/local domains include points on all halo box edges.

> **Tip**
>
> This function is independent of the finite Voronoi model. It only computes global/local point-index bookkeeping.

Parameters

- **points** (*ndarray*) – (N,2) array of point coordinates.
- **grid_shape** (*tuple[int, int]*) – Number of owned domains in the x and y directions.
- **halo_width** (*float*) – Width added to each side of every owned domain to collect local halo points.
- **domain_bounds** (*tuple[tuple[float, float], tuple[float, float]] | None*) – Optional domain bounds as `((xmin, xmax), (ymin, ymax))`. If *None*, bounds are inferred from *points*.
- **method** (*Literal['dense', 'binned', 'sorted_x']*) – Method used to collect halo points. `"dense"` builds a dense domain-by-point mask and is usually faster for moderate systems. `"binned"` reuses the owned-domain bins to reduce the number of candidate points checked for each halo and is usually faster for many domains. `"sorted_x"` uses less temporary memory.

Returns

One list of `DomainDecomposition` objects in row-major order, where each element stores the local and global point information for a single domain.

Return type

list[DomainDecomposition]

Raises

- **ValueError** – If *points* does not have shape (N,2).
- **ValueError** – If *points* contains non-finite values.
- **ValueError** – If *grid_shape*, *halo_width*, *domain_bounds*, or *method* is invalid.

> **Caution**
>
> Future versions may introduce changes to features and APIs.

References

Indices

- genindex
- modindex

BIBLIOGRAPHY

[1] J. Huang, H. Levine, and D. Bi. Bridging the gap between collective motility and epithelial–mesenchymal transitions through the active finite voronoi model. *Soft Matter*, 19(48):9389–9398, 2023. doi:10.1039/D3SM00327B.

[2] E. Teomy, D. A. Kessler, and H. Levine. Confluent and nonconfluent phases in a model of cell tissue. *Phys. Rev. E*, 98:042418, Oct 2018. doi:10.1103/PhysRevE.98.042418.

[3] W. Wang and B. A. Camley. Divergence of detachment forces in the finite Voronoi model. *arXiv preprint arXiv:2604.15481*, 2026. doi:10.48550/arXiv.2604.15481.

[4] J.-Q. Lv, P.-C. Chen, Y.-P. Chen, H.-Y. Liu, S.-D. Wang, J. Bai, C.-L. Lv, Y. Li, Y. Shao, X.-Q. Feng, and B. Li. Active hole formation in epithelioid tissues. *Nature Physics*, 20(8):1313–1323, 2024. doi:10.1038/s41567-024-02504-1.

PYTHON MODULE INDEX

c

pyafv.calibrate, 63

p

pyafv, 49

INDEX

Symbols

_build_voronoi_with_extensions() (*FiniteVoronoiSimulator method*), 55
_per_cell_geometry() (*FiniteVoronoiSimulator method*), 56
_step_update() (*DeformablePolygonSimulator method*), 65

A

auto_calibrate() (*in module pyafv.calibrate*), 63

B

build() (*FiniteVoronoiSimulator method*), 53
build() (*ParallelFiniteVoronoiSimulator method*), 59

C

contact_length (*DeformablePolygonSimulator attribute*), 65

D

decompose_points() (*in module pyafv*), 68
DeformablePolygonSimulator (*class in pyafv.calibrate*), 64
detach_criterion (*DeformablePolygonSimulator attribute*), 65
detached (*DeformablePolygonSimulator attribute*), 65
DomainDecomposition (*class in pyafv*), 68

F

FiniteVoronoiSimulator (*class in pyafv*), 52

G

get_cluster_sizes() (*in module pyafv*), 68
get_steady_state() (*PhysicalParams method*), 51

M

module
    pyafv, 49
    pyafv.calibrate, 63

P

ParallelFiniteVoronoiSimulator (*class in pyafv*), 57
PhysicalParams (*class in pyafv*), 50
plot_2d() (*DeformablePolygonSimulator method*), 65
plot_2d() (*FiniteVoronoiSimulator method*), 54
polygon_area_perimeter() (*in module pyafv.calibrate*), 66
polygon_centroid() (*in module pyafv.calibrate*), 66
preferred_areas (*FiniteVoronoiSimulator property*), 53
preferred_areas (*ParallelFiniteVoronoiSimulator property*), 58
pts1 (*DeformablePolygonSimulator attribute*), 64
pts2 (*DeformablePolygonSimulator attribute*), 64
pyafv
    module, 49
pyafv.calibrate
    module, 63

R

rebuild_connection_matrix() (*in module pyafv*), 67
replace() (*PhysicalParams method*), 52
resample_polyline() (*in module pyafv.calibrate*), 66

S

select_daughter_cluster() (*in module pyafv*), 67
simulate() (*DeformablePolygonSimulator method*), 65

T

target_delta() (*in module pyafv*), 62
tile_pbc() (*in module pyafv*), 67

U

update_params() (*FiniteVoronoiSimulator method*), 54
update_params() (*ParallelFiniteVoronoiSimulator method*), 59
update_positions() (*FiniteVoronoiSimulator method*), 55
update_positions() (*ParallelFiniteVoronoiSimulator method*), 60
update_preferred_areas() (*FiniteVoronoiSimulator method*), 55

update_preferred_areas() (*ParallelFiniteVoronoiSimulator method*), 60

V

visualize_2d() (*in module pyafv*), 61
visualize_2d_parallel() (*in module pyafv*), 62

W

with_optimal_radius() (*PhysicalParams method*), 51