TUTORIAL ON VERTEX MODEL FOR TISSUE MECHANICS

Rastko Sknepnek

School of Life Sciences
School of Science and Engineering

University of Dundee

VMTutorial

Biotechnology and Biological Sciences Research Council

Engineering and Physical Sciences Research Council

Layout

PART I: A brief overview of basic biology of epithelia

PART II: Vertex model
- Model overview and key features
- Adding dynamics
- Forces on a vertex

PART III: Implementation
- Aims
- Package design
- Half-edge data structure
- Add physics

PART IV: Running simulations
- Building initial configurations
- Setting and running simulation
- Data visualisation and analysis

PART I: A brief overview of basic biology of epithelia

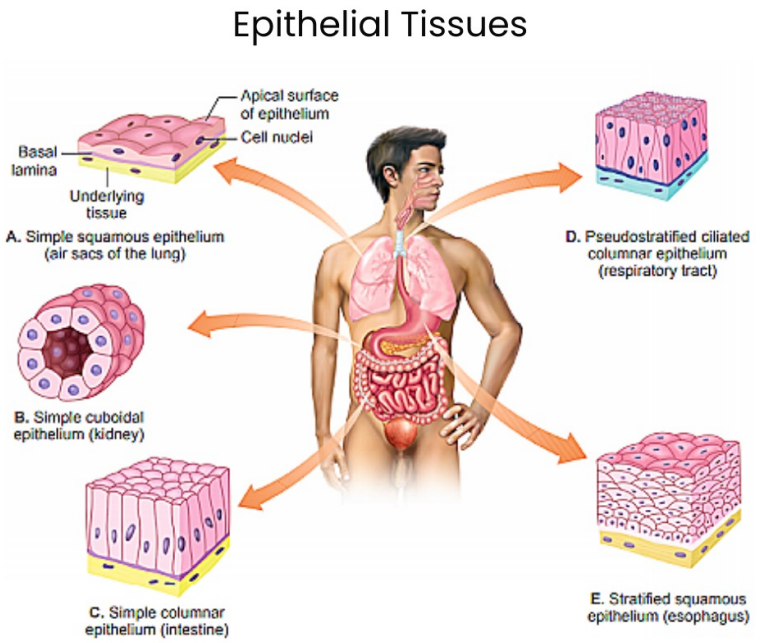
Epithelia are sheets of cells covering exterior surfaces of the body, lining internal cavities and passageways, and forming certain glands.

Functions:

- Protection (e.g., skin)
- Absorption (e.g., intestinal lining)
- Secretion (e.g., glandular epithelium)
- Sensation (e.g., taste buds)

Characteristics:

- Closely packed cells with minimal extracellular material
- Cells are polarized with distinct apical (top) and basal (bottom) surfaces
- Attachment to a basement membrane
- Avascular but innervated

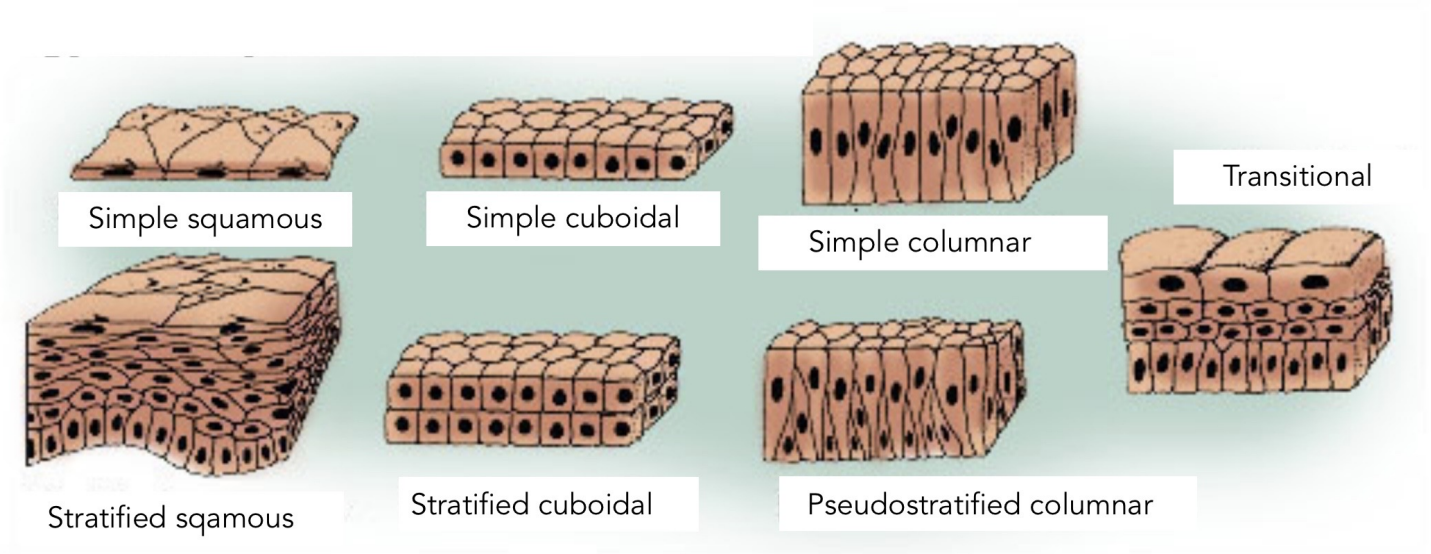


Classification based on layers:

- Simple Epithelium (single cell layer)
- Stratified Epithelium (multiple layers)

Classification based on cell shape:

- Squamous (flat and thin)
- Cuboidal (cube-like)
- Columnar (taller than wide)



Examples:

- Simple Squamous: Alveoli in lungs
- Stratified Squamous: Skin
- Simple Cuboidal: Kidney tubules
- Simple Columnar: Intestinal lining

Role in Health:

- Barrier against pathogens
- Regulates permeability
- Important in sensation

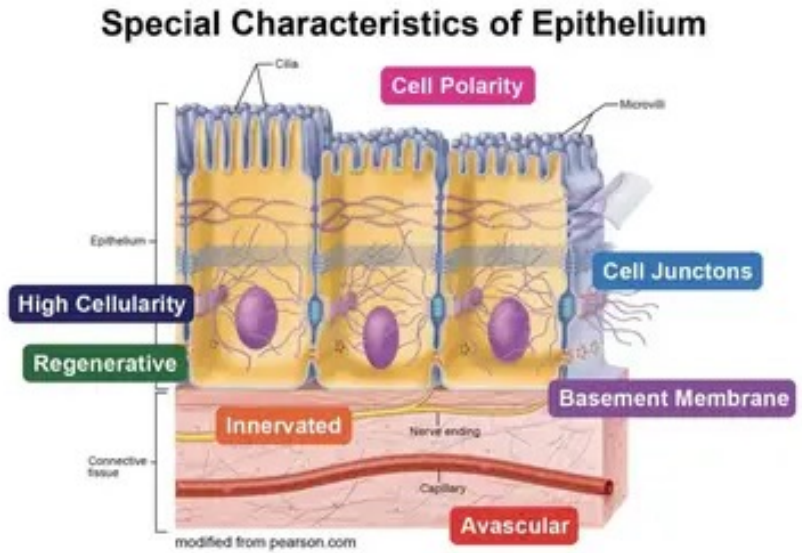
Role in Disease:

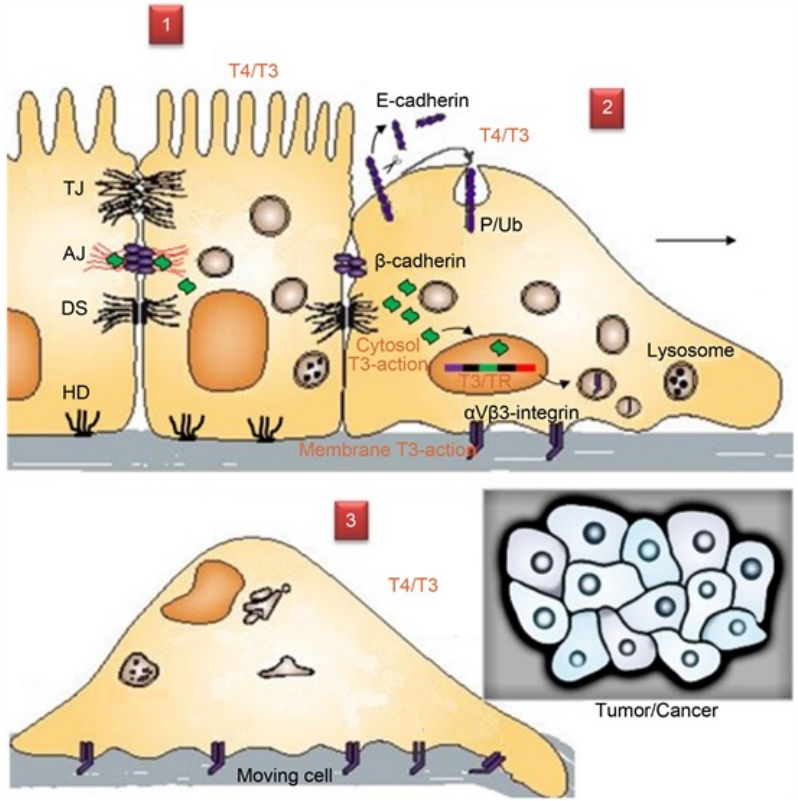
- Epithelial damage can lead to infections and diseases (e.g., ulcers)
- Cancer: Many cancers originate in epithelial cells (carcinomas)
- Healing: Epithelial cells play a key role in wound healing

Epithelial cells

Basic Structure:

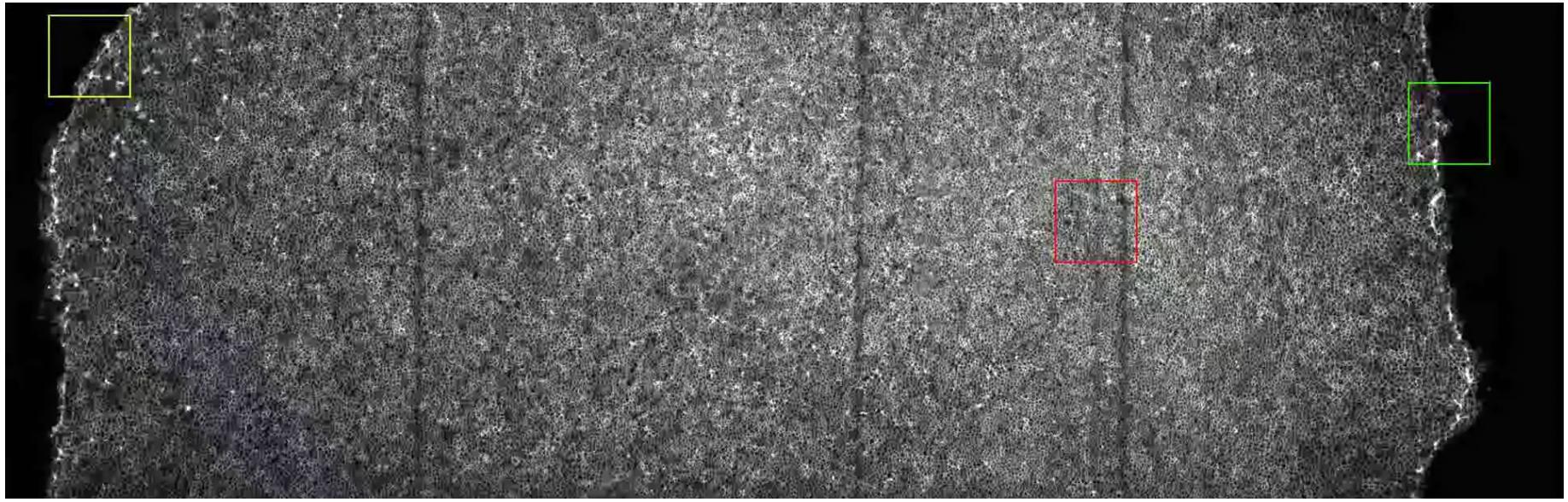
- **Polarity:** Epithelial cells have distinct apical (top), lateral, and basal (bottom) surfaces, each with different structures and functions.
- **Cell Junctions:** Tight junctions, adherens junctions, desmosomes, and gap junctions connect cells, ensuring structural integrity and facilitating communication.
- **Basement Membrane:** A specialized structure of extracellular matrix that anchors the epithelial tissue to underlying connective tissue.



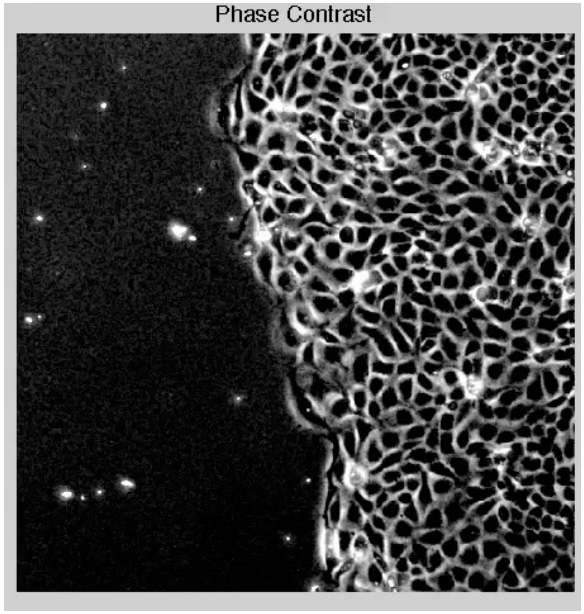


TJ: tight junction; AJ: adherens junction; DS: desmosome; HD: hemidesmosome

Gastrulation in chick embryo



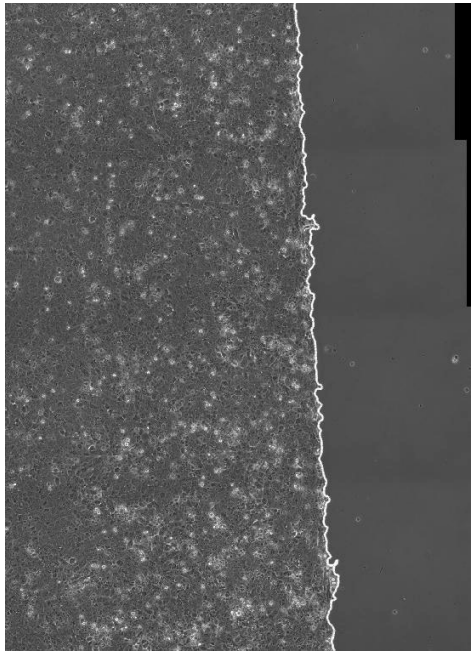
MDCK cells on a substrate



X. Trepat, et al., 2009

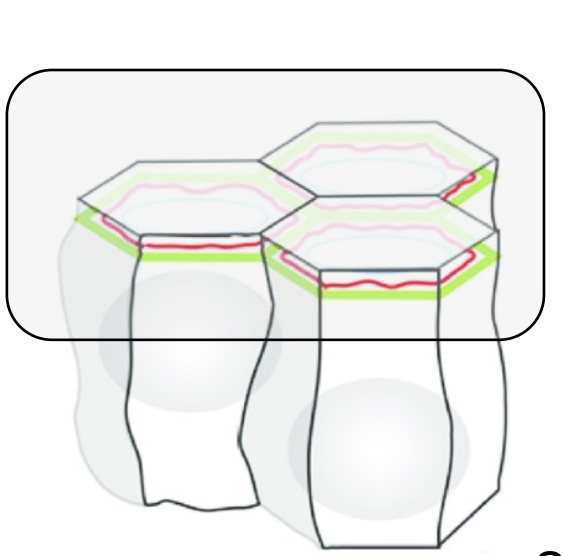
U2OS cells on a substrate

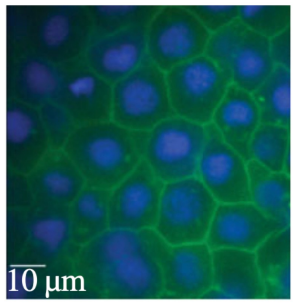
E. Rozbicki, et al., (2015).

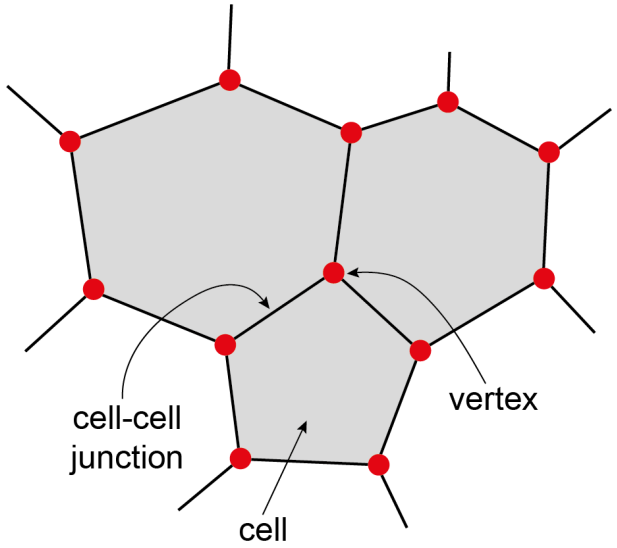


Näthke lab

PART II: Vertex model







Schötz et al. (2013)

Each cell described by:

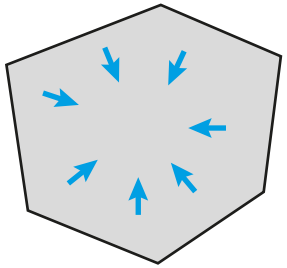
- Cell area A
- Cell perimeter P
- Native area A_0
- Area stiffness K
- Perimeter stiffness Γ

Each junction described by:

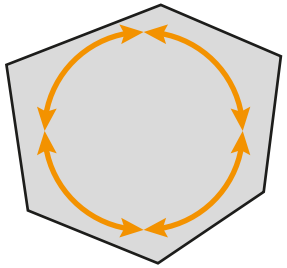
- Length l_{ij}
- Line tension Λ_{ij}

Minimum of energy: $$E = \sum_c \frac{K_c}{2}\left(A_C - A_0\right)^2 + \sum_c \frac{\Gamma_c}{2} P_c^2 + \sum_{\langle ij \rangle} \Lambda_{ij} \ell_{ij}$$

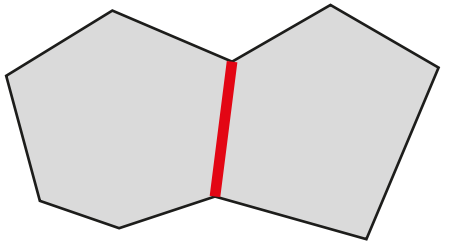
Honda&Eguchi, 1980
Farhadifar, et al. 2007
Fletcher, et al. 2014



area contraction/expansion



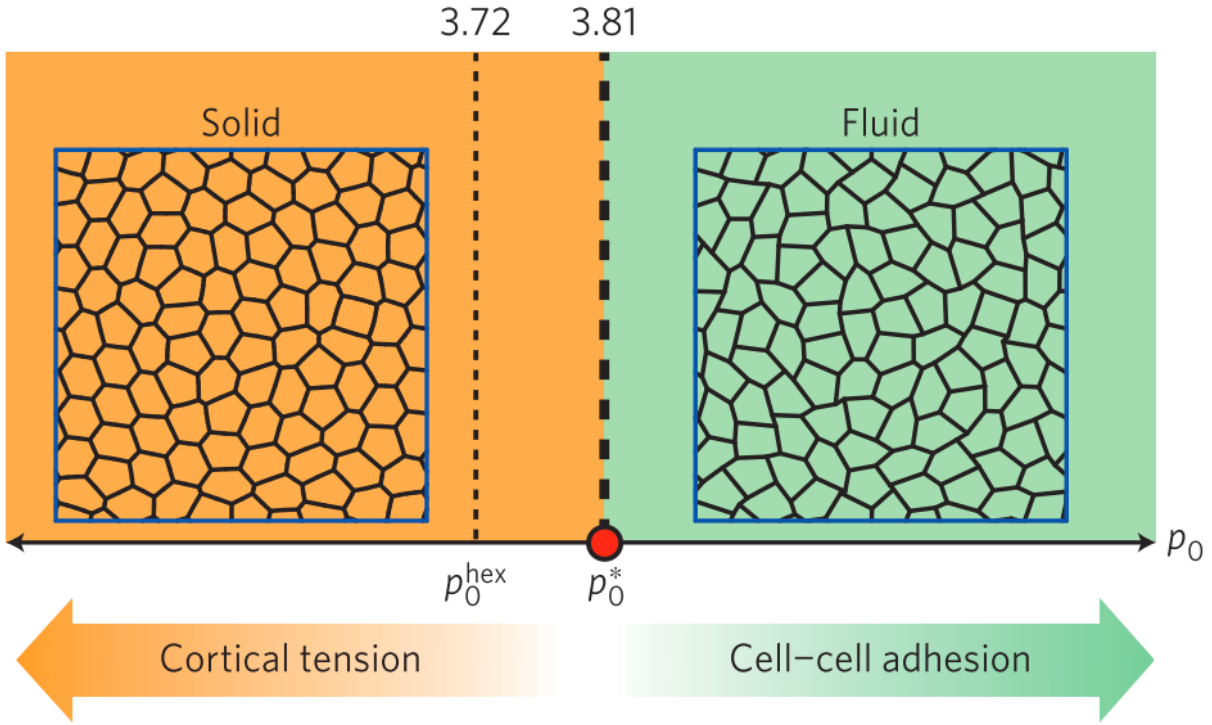
area contraction



line tension

Solid-fluid transition

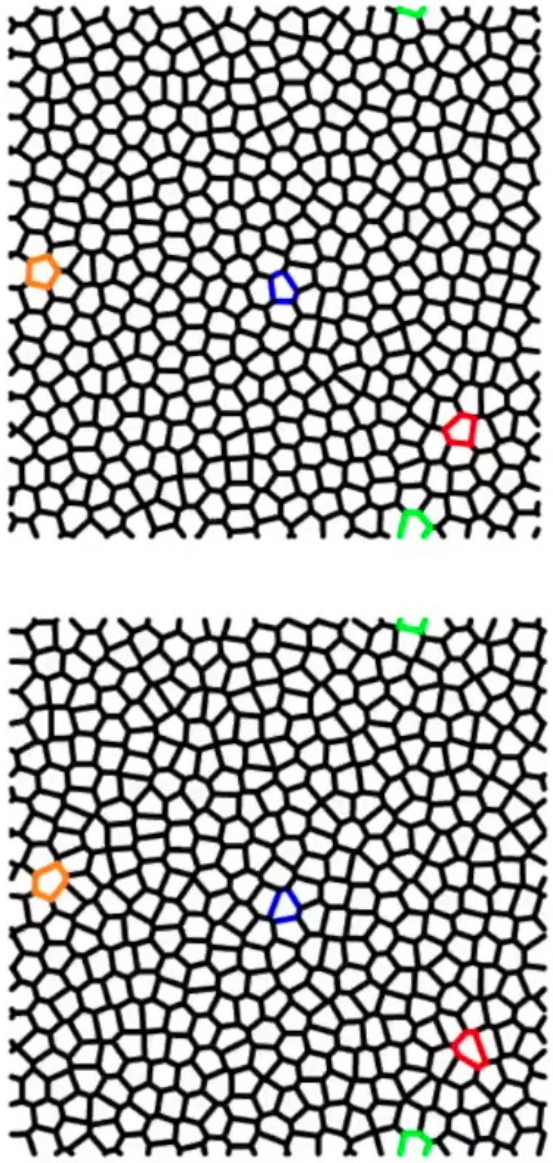
Rigidity transition as a function of geometry



Bi et al., 2015

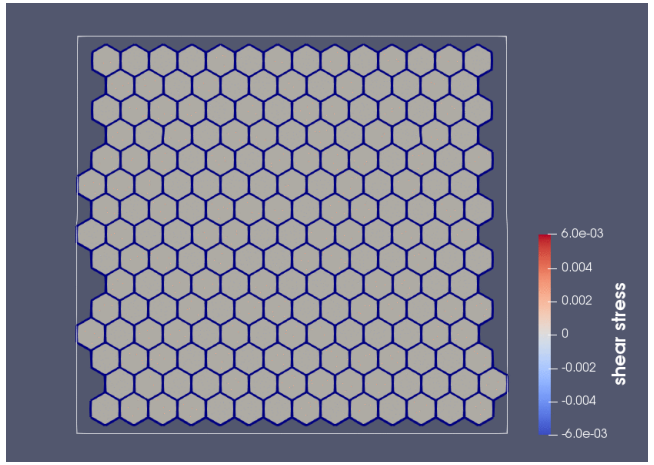
$$E = \sum_c \left[\frac{K}{2}(A_c - A_0)^2 + \frac{\Gamma}{2}(P_c - P_0)^2 \right] \qquad p_0 = \frac{P_0}{\sqrt{A_0}}$$

Overdamped dynamics with self propulsion: $\gamma \dot{\mathbf{r}}_i = f_0 \mathbf{n}_c - \nabla_{\mathbf{r}_i} E$



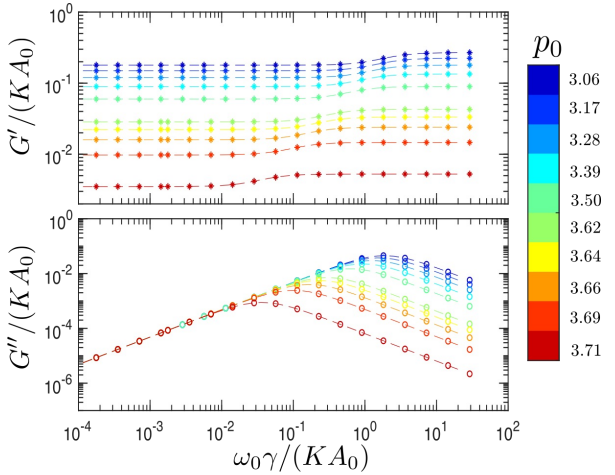
Bi, et al. 2016

Linear rheology

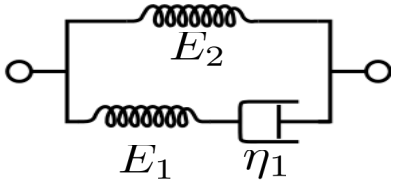


solid regime

$p_0 < p_0^{cr} = 3.72$



Standard Linear Solid (SLS) model



$\gamma\dot{\mathbf{r}}_i = -\nabla_{\mathbf{r}_i} E$ +affine deform.

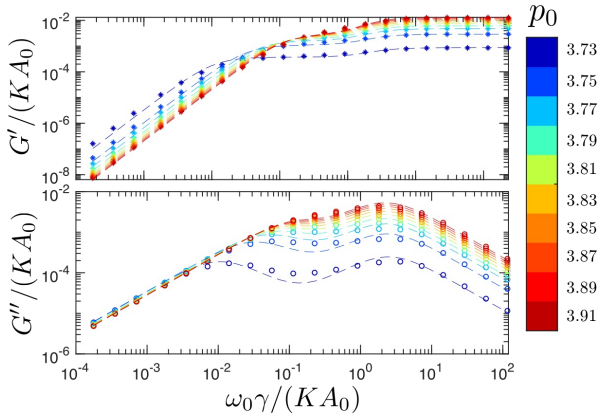
$$G^* = \frac{\tilde{\tau}(\omega_0)}{\tilde{\gamma}(\omega_0)}$$

$$G' = \mathrm{Re}(G^*)$$

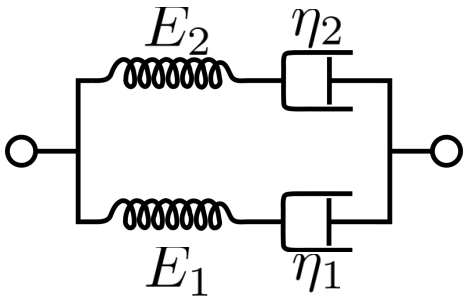
$$G'' = \mathrm{Im}(G^*)$$

fluid regime

$p_0 > p_0^{cr} = 3.72$



Burgers model

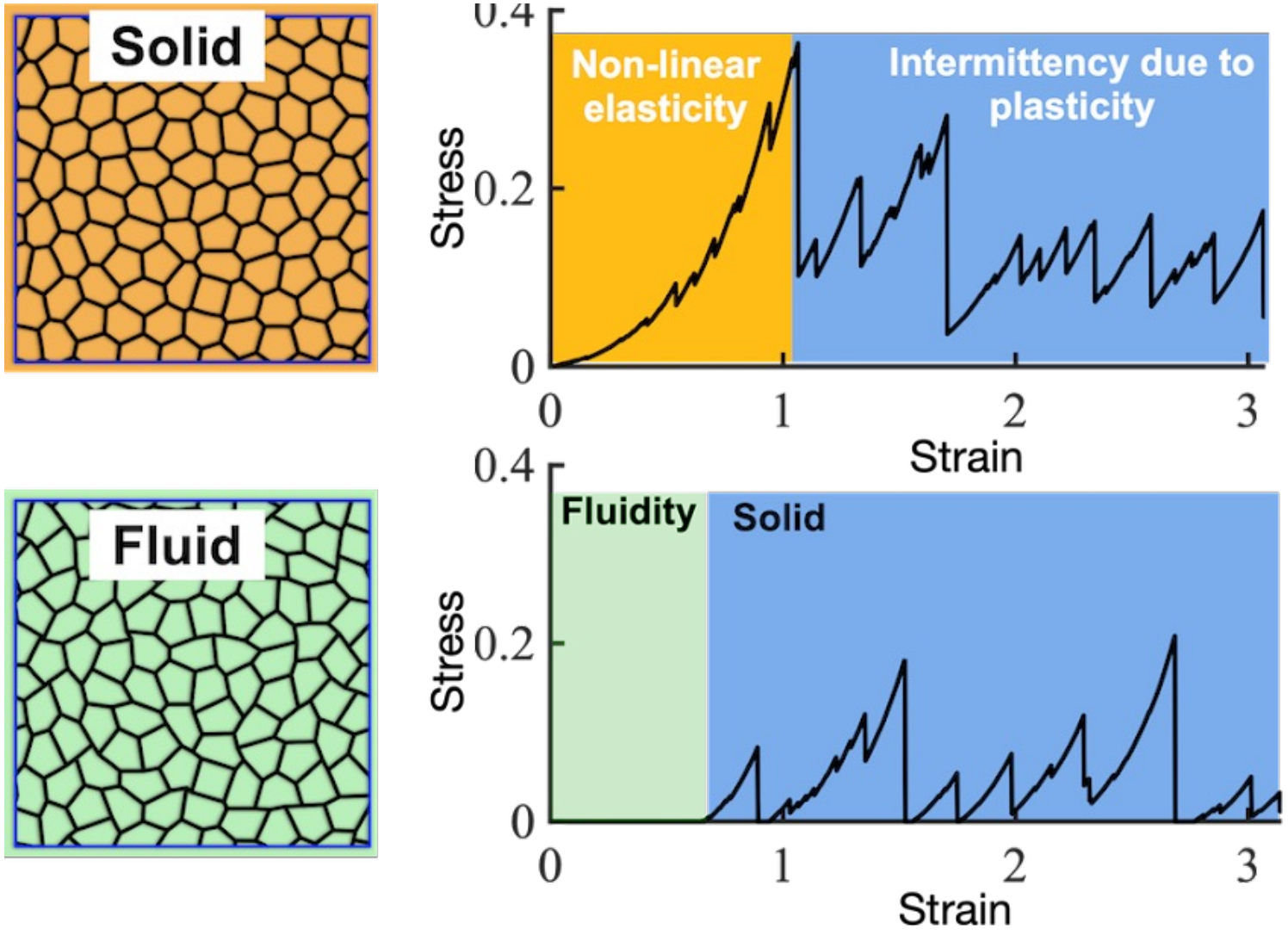


shear stress

$$p_0 = \frac{P_0}{\sqrt{A_0}}$$

Tong, et al. PLoS CB (2022)
Tong, et al. Phys Rev Res (2023)

Non-linear rheology

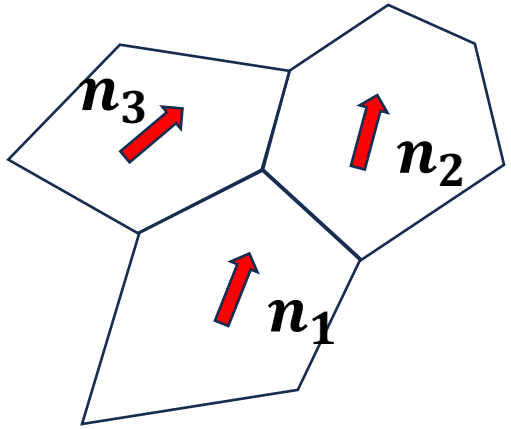


(image source: https://sites.google.com/view/dapengbi)

- fluid-like tissue acquires rigidity above a threshold value of the applied strain
- nonlinear elasticity becomes increasingly dominant closer to the critical point, where the mechanical response is completely nonlinear

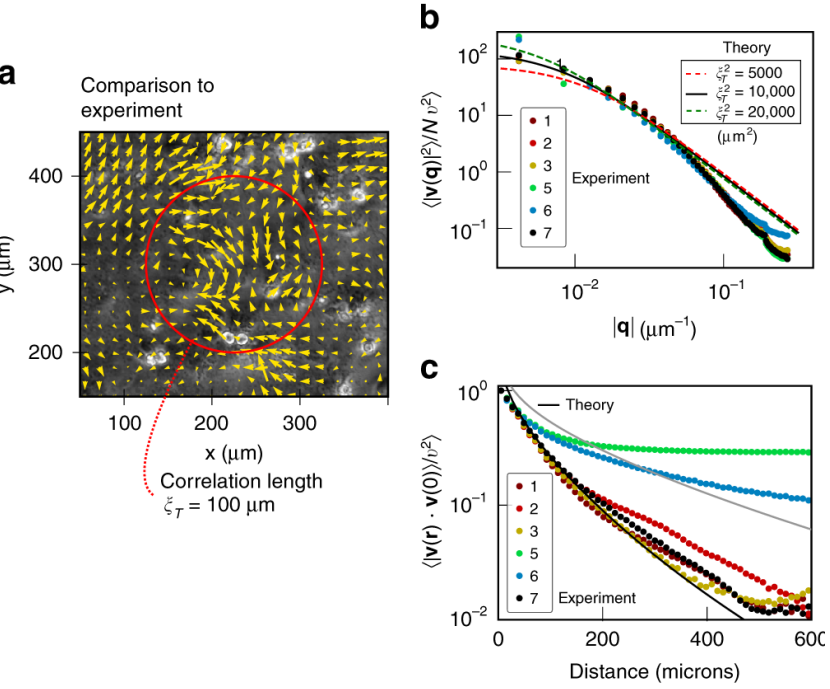
A. Hernandez, et al. PRE (2022)
J. Huang, et al. PRL (2022)
SM Fielding, et al. arXiv (2022)
MJ Hertaeg, et al. arXiv (20220

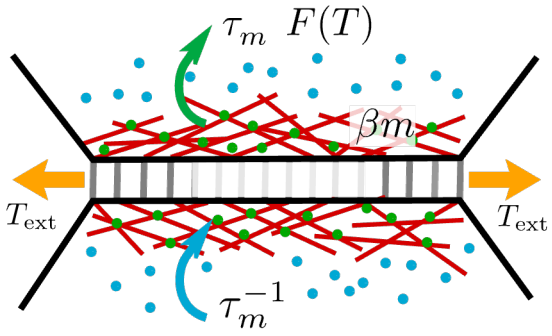
Examples of adding activity

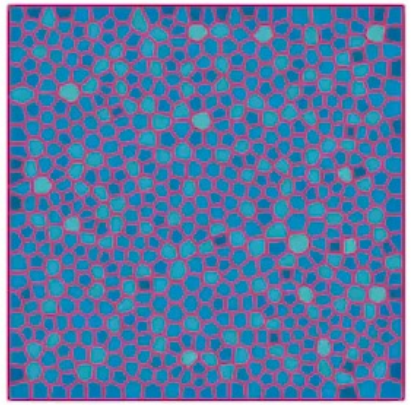


$$\mathbf{f}_i^{\text{active}} = f_0 \mathbf{n}_i$$

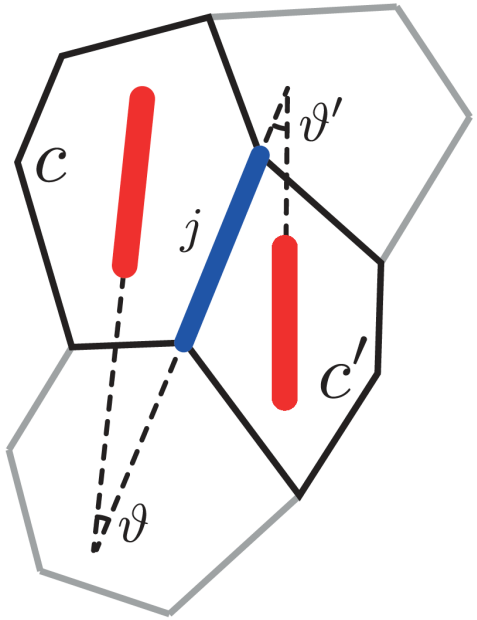
self-propulsion along vector $\boldsymbol{n}$.

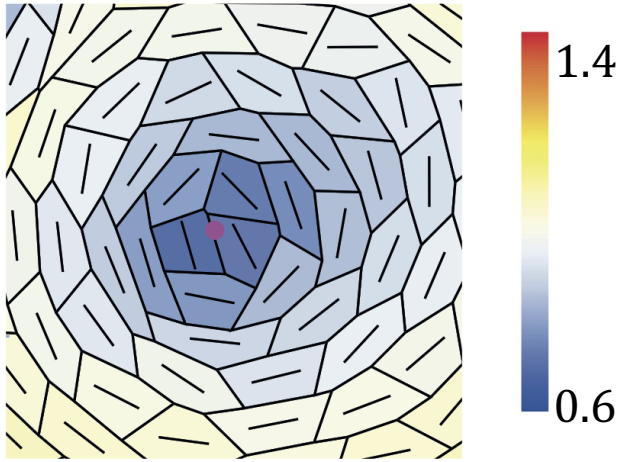






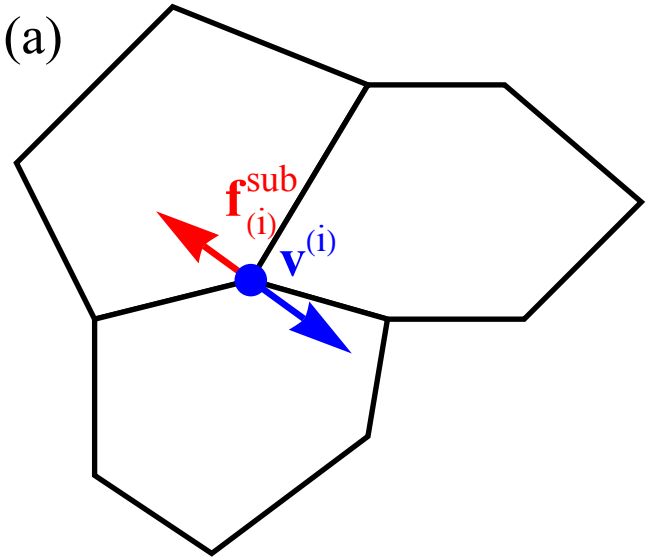
Barton, et al. PLoS CB, 2017
Matoz-Fernandez, et al., Soft Matter 2017
Petroli, et al. PRL, 2019
Henkes, et al. Nat. Comm. 2020
Saraswathibhatla, et al. Ext. Mech, 2021
Sknepnek, et al., eLife 2023





Lin, et al. PRL 2023
Rozman, et al. PRL 2023

Vertex model dynamics



Overdamped dynamics of the vertex i:

$$\gamma \dot{\mathbf{r}}_i = -\nabla_{\mathbf{r}_i} E + \mathbf{f}_i^{\text{active}}$$

friction with substrate

active force

$$E = \frac{\kappa}{2} \sum_C (A_C - A_0)^2 + \frac{\Gamma}{2} \sum_C (P_C - P_0)^2$$

$$\nabla_{\mathbf{r}_i} E = \sum_{[C]} \left\{ \frac{\kappa}{2} (A_C - A_0) \left(\mathbf{r}^{(C)}_{i+1,i} - \mathbf{r}^{(C)}_{i-1,i} \right) \times \mathbf{e}_z + \Gamma (P_C - P_0) \left[\frac{\mathbf{r}_i - \mathbf{r}^{(C)}_{i-1}}{\left| \mathbf{r}_i - \mathbf{r}^{(C)}_{i-1} \right|} - \frac{\mathbf{r}^{(C)}_{i+1} - \mathbf{r}_i}{\left| \mathbf{r}^{(C)}_{i+1} - \mathbf{r}_i \right|} \right] \right\}$$

Symbol $[C]$ means that the sum is only over cells that contain vertex i.

$\mathbf{r}^{(C)}_{i\pm1}$ refers to the previous/next vertex in the cell C

We focus only on the mechanics part!

PART III: Implementation

Scientific computing workflow:

1. Initial conditions setup (trivial to very complex)
2. Simulation (typically the most time-consuming part)
3. Data analysis and visualisation (can be computation-heavy)
4. Dissemination of the results

Aims:

- Build a coherent software package that integrates all steps in the simulation workflow.
- Have a user-friendly interface that is flexible to allow rapid simulation setup and execution.
- Separate physics from low-level implementation details.
- Be aware of the performance issues but not at the expense of clarity and maintainability.

Before we start…

!!! BE VERY CAREFUL WHEN MAKING EARLY DESIGN DECISIONS !!!

- How are different parts of the workflow going to communicate with each other?
- How is the data going to be stored and curated?
- How will the data provenance going to be achieved?

Things to consider:

- How “expensive” is the data (i.e. can it be easily regenerated).
- How scalable you need to be (singe-core/multi-core parallel/large-scale parallel).
- How does the new tool fit into the existing software ecosystem.

“If you think it's simple, then you have misunderstood the problem.”
— **Bjarne Stroustrup**

Design choices for this tutorial

- Initialisation, simulation, and analysis will be **fully decupled**, i.e. they will communicate with each other via a shared file format (we'll use JSON)
  - **Initialisation:** quite flexible, no performance concerns → simple self-contained Python scripts
  - **Simulation:** needs to be flexible yet efficient → C++ backend with Python frontend
  - **Data analysis and visualisation:** not too heavy → a Python package with a consistent interface.
- Moderate amount of data → no specific need for fancy data formats, i.e. use JSON and VTK (for visualisation)
- Limit to several thousand cells → we can do everything on a single CPU core.
- Easy installation and deployment (a simple pip install from the project home directory)
- No GUI → command line interface, but with Jupyter support
- Linux/Mac OS only but works in WSL2 on Windows 11.
- Visualisation will be left to third-party packages.

> ”I like to think of Python as your average American. It tries not to offend anybody and goes out of its way to look nice and be helpful.”
> **-- Guido Van Rossum**

Directory structure

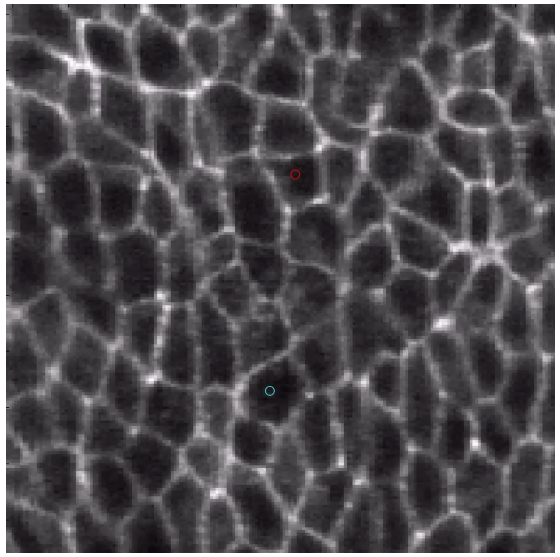
VMTutorial – base directory
- VMToolkit – the core of the package
  - VM – Vertex Model simulation
  - VMAnalysis – Python package for basic analysis
- examples – Several detailed examples
- config_builder – Set of tools for building initial configurations

NOTE! Proper directory structure is important for proper installation and function of the package.

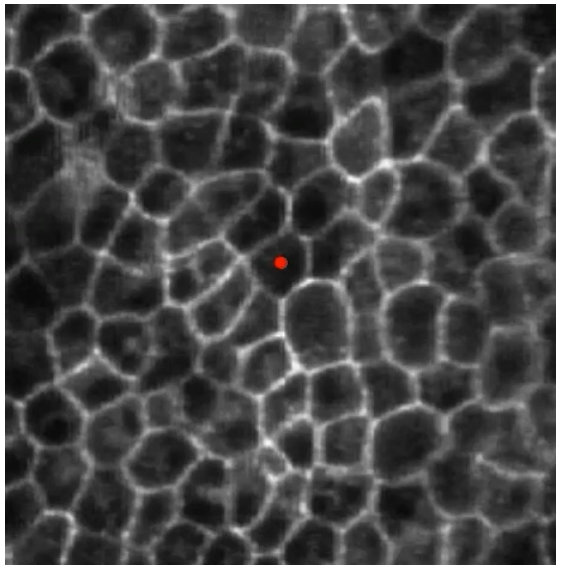
Vertex Model implementation

Required features:

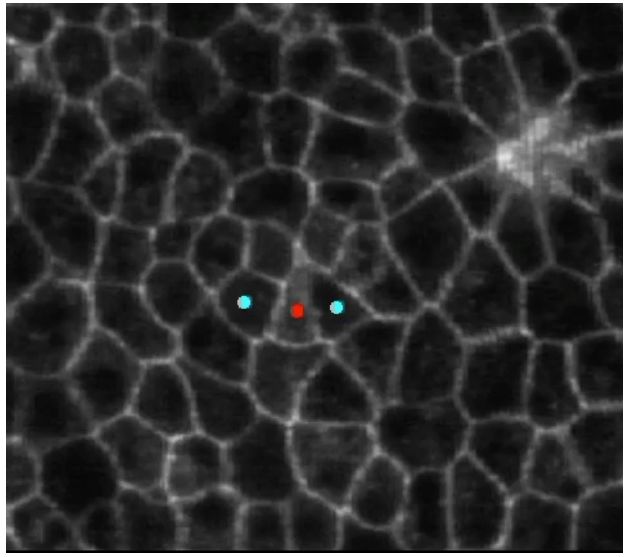
Vertex dynamics – vertices need to be able to move in response to forces acting on them
Topology changed – cells need to be able to change neighbours, divide, and ingress



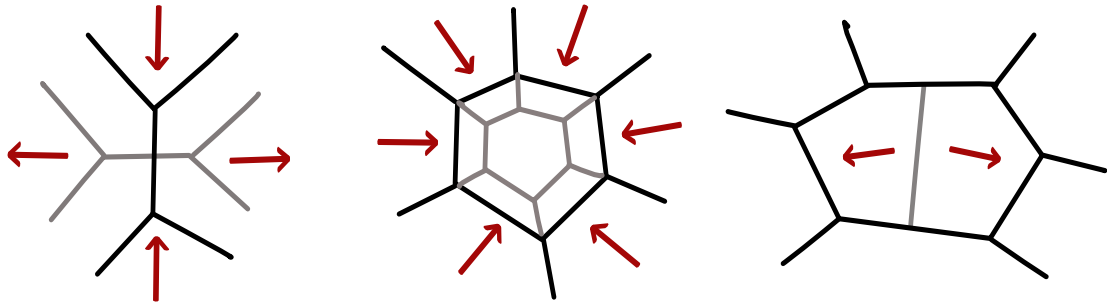
intercalation
(neighbour exchange)



ingression/extrusion



cell division



Key implementation choices

- C++ backend for efficiency, with Python frontend for clarity and ease of use
- Modular design – key simulation steps are implemented as separate, loosely coupled modules.
- Find balance between speed and flexibility (avoid hard-coding specific assumptions)
- Decouple physics from the low-level implementation details (e.g. force on a vertex does not need to know how the vertex is stored in memory).

Rely on libraries as much as possible (when practical).

Components

1. **Mesh** – the data structure representation of the tissue configuration
2. **System** – Handles input, setting cell properties, etc.
3. **Forces** – Handles force computation on each vertex
4. **Integrators** – Handles integration of equations of motion
5. **Dump** – Handles output of simulation results in different formats.

Algorithms + Data Structures = Programs

(N. Wirth 1976)

Data structures

We need a way to represent a two-dimensional polygonal tiling. We refer to this the Mesh.

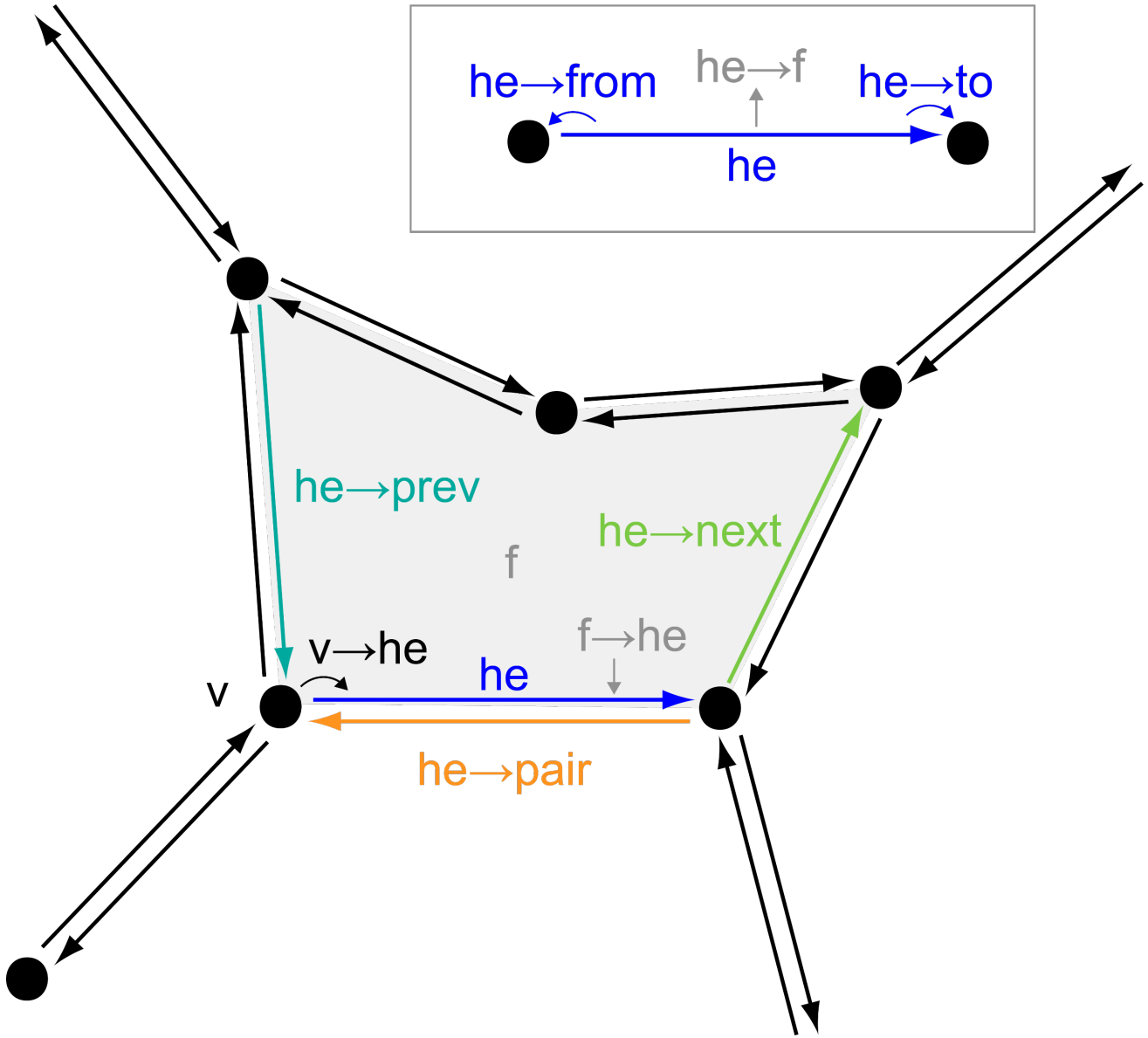
There are many excellent libraries for handling meshes, e.g.:

- OpenMesh
- CGAL

However, none of them do what we need and are also far complicated for our needs.

We will implement our own light-weight mesh representation based on the half-edge data structure.

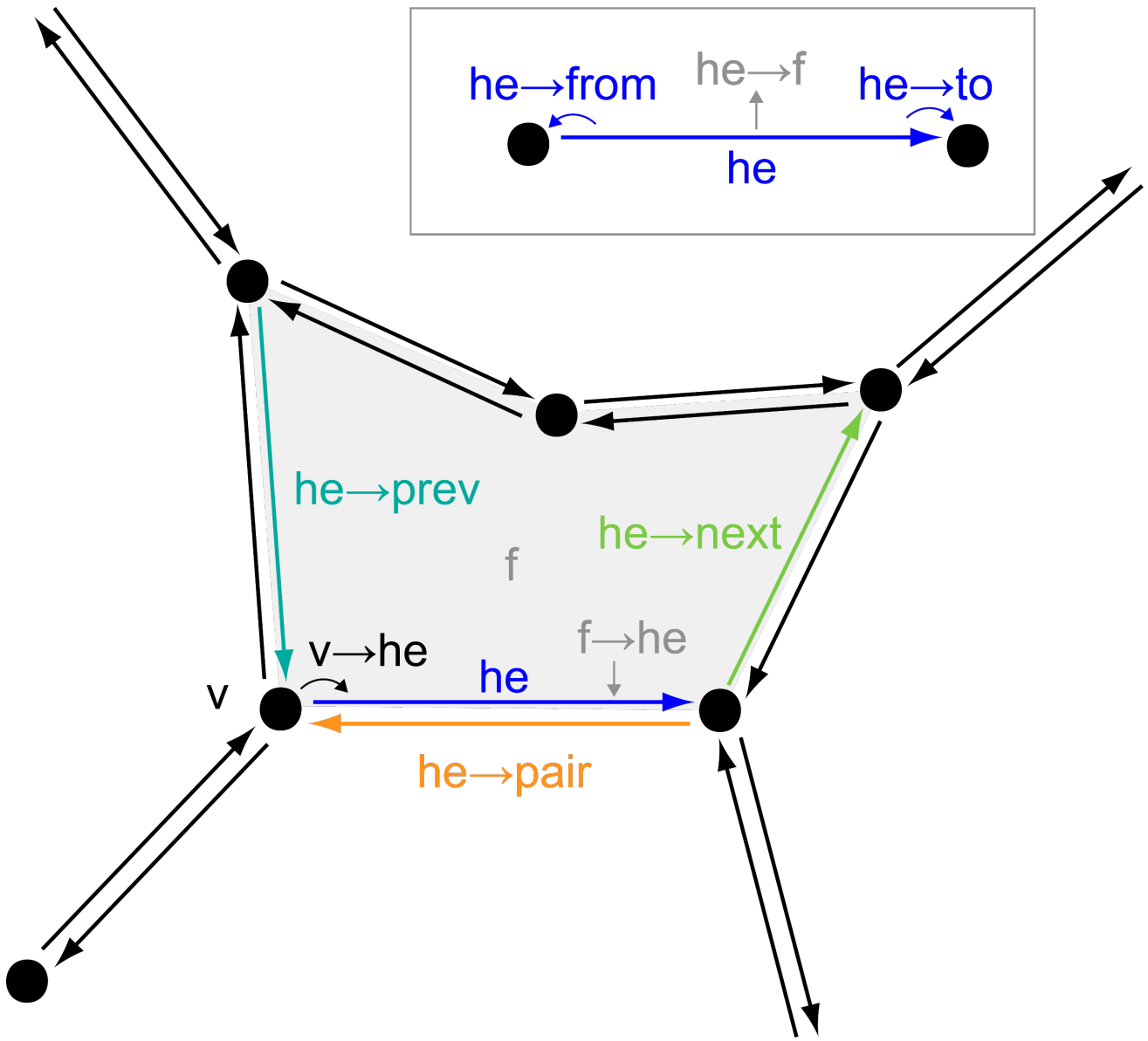
Basic half-edge data structure




```
struct Vertex {
   HalfEdge* he;
   // …
}

struct Face {
    HalfEdge* he;
    // …
}

struct HalfEdge {
   Vertex* from;
   Vertex* to;
   Face* f;
   HalfEdge* prev;
   HalfEdge* next;
   HalfEdge* pair;
   // …
}
```

Muller, Preparata (1978) *Theoretical Computer Science*. **7,** 217 (1978).

Traversing the mesh



Loop over vertex neighbours

```
HalfEdge* he = v→he;
HalfEdge* first = v→he;
do {
    vn = he→to;
    // do something with vn
    he = he→pair→next;
} while (he != first);
```

Loop over face vertices

```
HalfEdge* he = f→he;
HalfEdge* first = f→he;
do {
    v1 = he→from;
    v2 = he→to;
    // do something with v1 and v2
    he = he→next;
} while (he != first);
```

Half-edge data structure

PROS

Easy navigation

Easy insertion/removal of elements

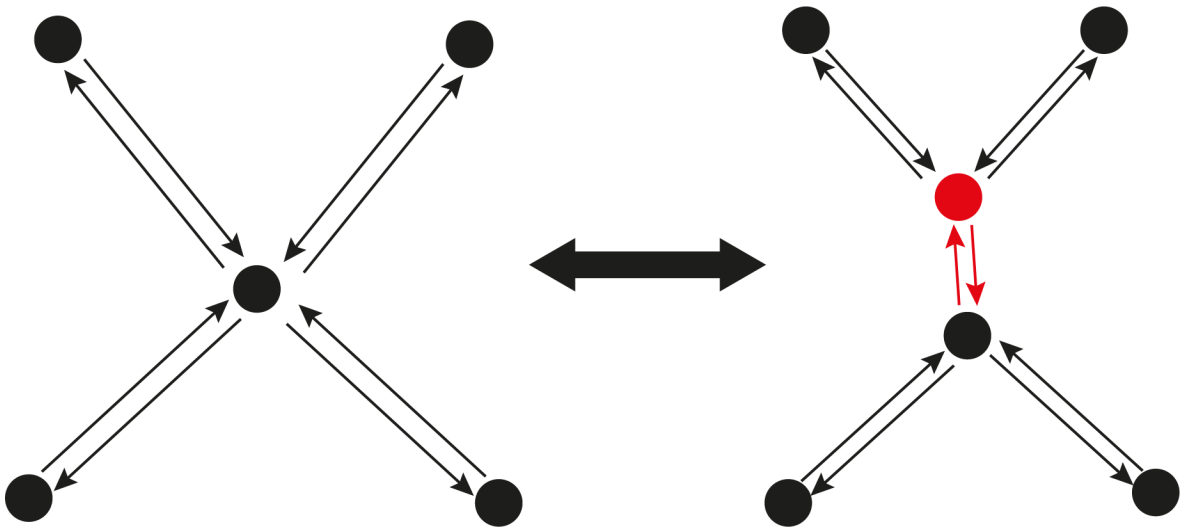
Robust against faulty meshes (needs to be self-consistent)

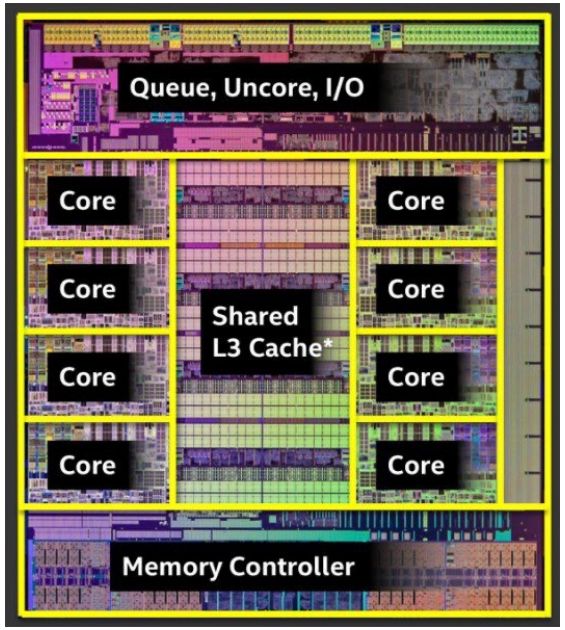
CONS

Not easy to implement

Hard to parallelise in its native form

Not cache-friendly (in its native form)

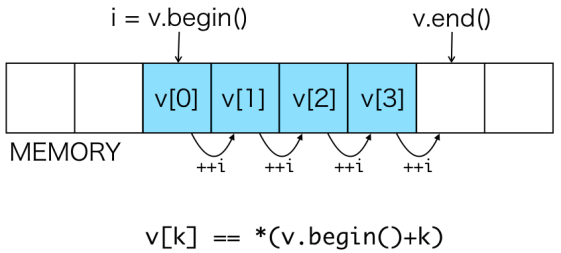




Our implementation

Use the C++ standard library

- STL containers – leave memory management to the STL
- iterators – related to pointers but a bit more intelligent




```
using HEHandle = vector<HalfEdge<Property>>::iterator;
using VertexHandle = vector<Vertex<Property>>::iterator;
using EdgeHandle = vector<Edge<Property>>::iterator;
using FaceHandle = vector<Face<Property>>::iterator;

template <typename Property>
class HalfEdge
{
public:
  typename Property::HEProperty &data() { return _property; }
  VertexHandle<Property> from() { return _mesh.get_mesh_vertex(_from); }
  VertexHandle<Property> to() { return _mesh.get_mesh_vertex(_to); }
  EdgeHandle<Property> edge() { return _mesh.get_mesh_edge(_edge); }
  FaceHandle<Property> face() { return _mesh.get_mesh_face(_face); }
  HEHandle<Property> pair()   { return _mesh.get_mesh_he(_pair); }
  HEHandle<Property> next()   { return _mesh.get_mesh_he(_next); }
  HEHandle<Property> prev()   { return _mesh.get_mesh_he(_prev); }
private:
  int _idx, _from, _to, _edge, _face, _pair, _next, _prev;
  typename Property::HEProperty _property;
  Mesh<Property> &_mesh;
}
```

aves programmers from having to reinvent the wheel."
— **Bjarne Stroustrup**

```
class Vertex {
public:
  // …
  typename Property::VertexProperty &data() { return _property; }
  HEHandle<Property> he() { return _mesh.get_mesh_he(_he); }
  Vec r;
  // ..
private:
  typename Property::VertexProperty _property;
  int _he;
  Mesh<Property> &_mesh;
  // …
};
```

```
class Face {
public:
  // …
  typename Property::FaceProperty &data() { return _property; }
  HEHandle<Property> he() { return _mesh.get_mesh_he(_he); }
  // …
private:
  typename Property::FaceProperty _property;
  int _he;
  Mesh<Property> &_mesh;
  // …
};
```

```
template <typename Property>
class Mesh
{
  // …
  HEHandle<Property> get_mesh_he(int);
  VertexHandle<Property> get_mesh_vertex(int);
  EdgeHandle<Property> get_mesh_edge(int);
  FaceHandle<Property> get_mesh_face(int);
  // …
private:
  vector<HalfEdge<Property>> _halfedges;
  vector<Vertex<Property>> _vertices;
  vector<Edge<Property>> _edges;
  vector<Face<Property>> _faces;
  // …
};
```

Why vectors? Vectors have fast random access and memory coalescence (helps with cache).

Warning! Iterators are invalidated each time a vector grows. (Consider list, but not memory local.)

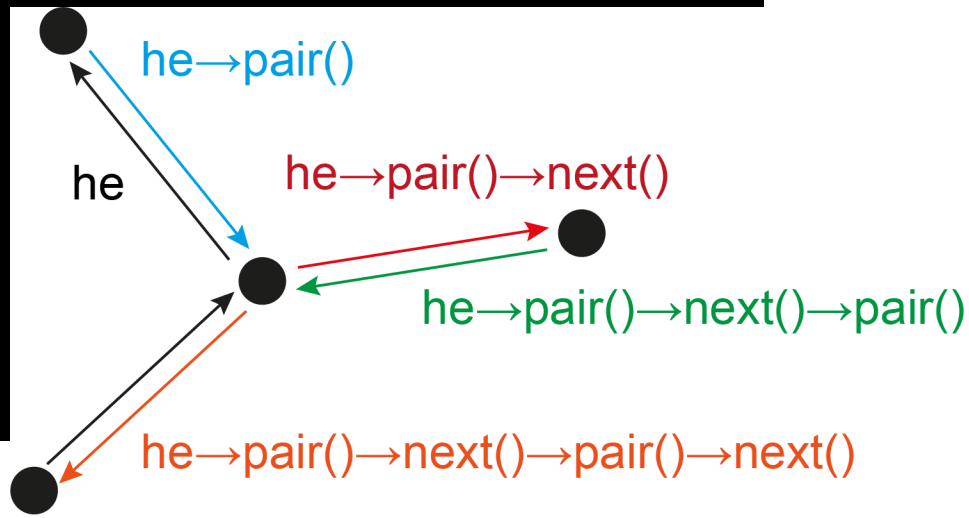
What is that “Property” thing and why is it templated?

Mesh really does not need to know about parameters such as A_0, P_0, κ, etc., so we burry all details into the Property class and pass it as a template argument.

```
struct Property : public BaseProperty
{
  struct HEProperty : public BaseProperty::HEProperty {
    double tension = 0.0;
    // …
  };
  struct VertexProperty : public BaseProperty::VertexProperty {
    Vec vel;
    // …
  };
  struct FaceProperty : public BaseProperty::FaceProperty {
    double A0;
  };
};
```

Finding vertex coordination number

```
template<typename Property>
int Mesh<Property>::coordination(const VertexHandle<Property>& vh)
{
  HEHandle<Property> he = vh->he();
  HEHandle<Property> first = he;
  int i = 0;
  do
  {
    i++;
    he = he->pair()->next();
  } while (he != first);
  return i;
}
```

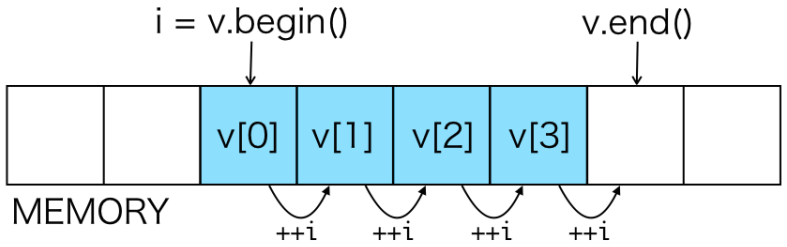


Clumsy and bloated! Instead…

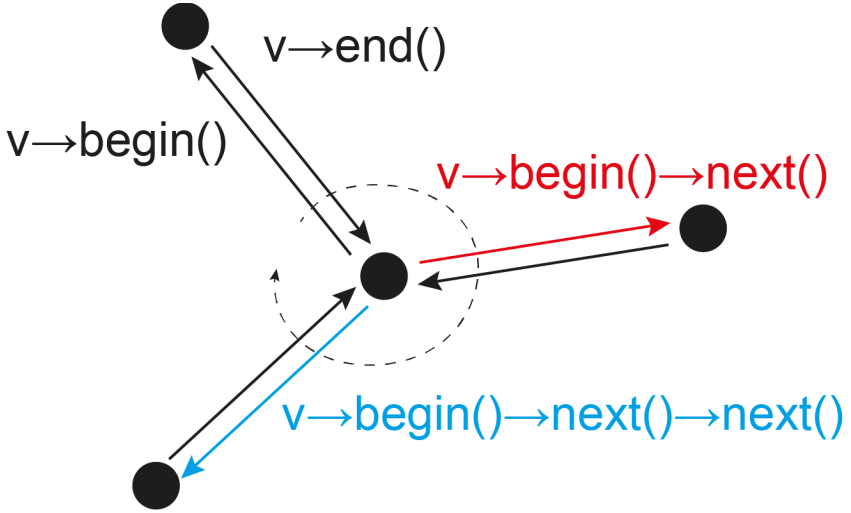
```
template <typename Property>
int Mesh<Property>::coordination(const Vertex<Property> &v)
{
  int i = 0;
  for (auto he : v.circulator())
    i++;
  return i;
}
```

Welcome to the world of circulators!

A close cousin of iterators…



v[k] == *(v.begin()+k)




```
template <typename Property>
class VertexCirculator
{
  // …
  VertexCirculator() : _start{}, _current{},_isEnd{true} {}
  VertexCirculator(HEHandle<Property> he) : _start{he},
_current{he}, _isEnd{false} { }
  VertexCirculator &operator++() {
    _current = _current->pair()->next();
    if (_current == _start) isEnd = true;
    return *this;
  }
  bool operator==(const VertexCirculator &other) const {
      return (_isEnd && other._isEnd) || (_current ==
other._current);
   }
  FaceCirculator begin() { return *this; }
  FaceCirculator end() { return FaceCirculator(); }
 // …
 private:
   HEHandle<Property> _start;
   HEHandle<Property> _current;
   bool _isEnd;
}
```

Force calculation (i.e. back to physics…)

Force on a vertex:

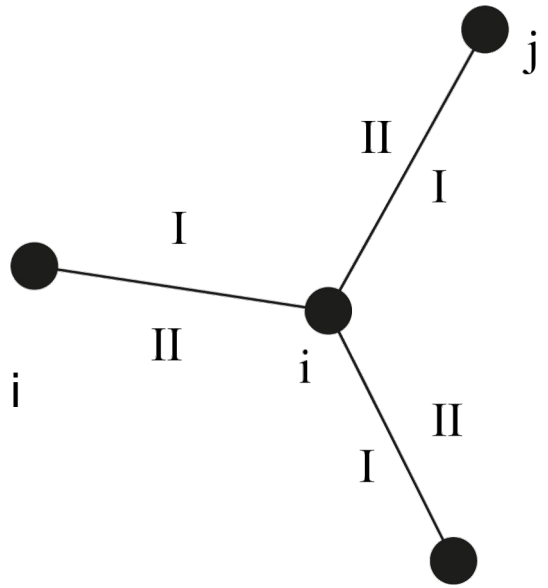
$$\nabla_{\mathbf{r}_i} E = \sum_{[C]} \left\{ \frac{\kappa}{2} \left(A_C - A_0\right) \left(\mathbf{r}^{(C)}_{i+1,i} - \mathbf{r}^{(C)}_{i-1,i}\right) \times \mathbf{e}_z + \Gamma \left(P_C - P_0\right) \left[\frac{\mathbf{r}_i - \mathbf{r}^{(C)}_{i-1}}{\left|\mathbf{r}_i - \mathbf{r}^{(C)}_{i-1}\right|} - \frac{\mathbf{r}^{(C)}_{i+1} - \mathbf{r}_i}{\left|\mathbf{r}^{(C)}_{i+1} - \mathbf{r}_i\right|} \right] \right\}$$

Symbol $[C]$ means that the sum is only over cells that contain vertex i.

$\mathbf{r}^{(C)}_{i\pm1}$ refers to the previous/next vertex in the cell C

It can be rewritten as

$$\mathbf{F}_i = -\sum_j \left[\frac{\kappa}{2} \left(A_I - A_{II}\right) \mathbf{e}_z \times \mathbf{r}_{j,i} - \Gamma \left(p_I + p_{II}\right) \hat{\mathbf{r}}_{j,i} \right]$$



j sum runs in counterclockwise order over all vertices that are connected i

$$p_{I,II} = \left(P_{I,II} - P_0\right)$$

```
void compute(Vertex<Property> &v) {
    v.data().force = Vec(0.0,0.0);
    for (auto he : v.circulator())
        v.data().force += this->compute(v, he);
}
```

We can, however, split force into different contributions and turn them on/off as needed.

$$\mathbf{F}_i = -\sum_j \left[\frac{\kappa}{2} \left(A_I - A_{II} \right) \mathbf{e}_z \times \mathbf{r}_{j,i} - \Gamma \left(p_I + p_{II} \right) \hat{\mathbf{r}}_{j,i} \right]$$

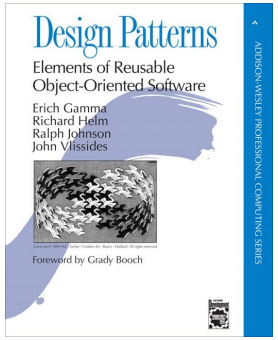
ForceArea ForcePrerimeter

Computing total force is just a sum over all components:

```
Vec compute(Vertex<Property> &v, const HalfEdge<Property> &he) {
   Vec ftot(0,0);
   for (auto& f : this->factory_map)
     ftot += f.second->compute(v, he);
   return ftot;
}
```

Simple, but we need the factory pattern…

```
class ClassFactory {
  typedef unique_ptr<BaseType> ptrBaseType;
  typedef map<string, ptrBaseType> factory_type;
public:
  // …
  template <typename DerivedType, typename... Args>
  void add(const string &key, const Args &...args) {
     if (factory_map.find(key) != factory_map.end())
         throw invalid_argument(key + " is already in the class factory.");
     factory_map[key] = make_unique<DerivedType>(args...);
  }

}
```



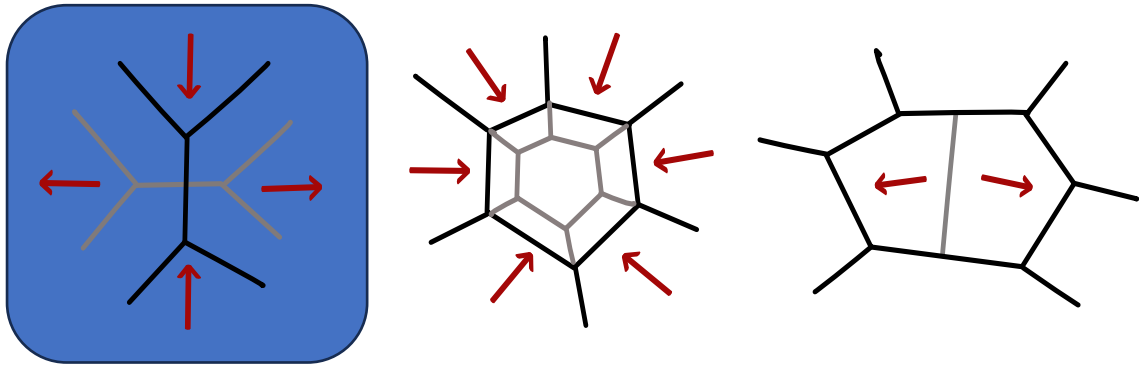
```
class ForceCompute : public ClassFactory<Force>
{
public:
  // …
  void add_force(const string& fname) {
      if (name == "area")
          this->add<ForceArea,System&>(name, _sys);
      else if (name == "perimeter")
          this->add<ForcePerimeter,System&>(name, _sys);
      else if (name == "self-propulsion")
           this->add<ForceSelfPropulsion,System&>(name, _sys);
      else
           throw runtime_error("Unknown force type : " + name + ".");
   }

private:
   System& _sys;
};
```

Adding a new force type is simple – just make a new force class and register it here.

```
class ForceArea : public Force {
public:
  //…
  Vec compute(const Vertex<Property>&, const HalfEdge<Property>&) override;
  // …
}
```

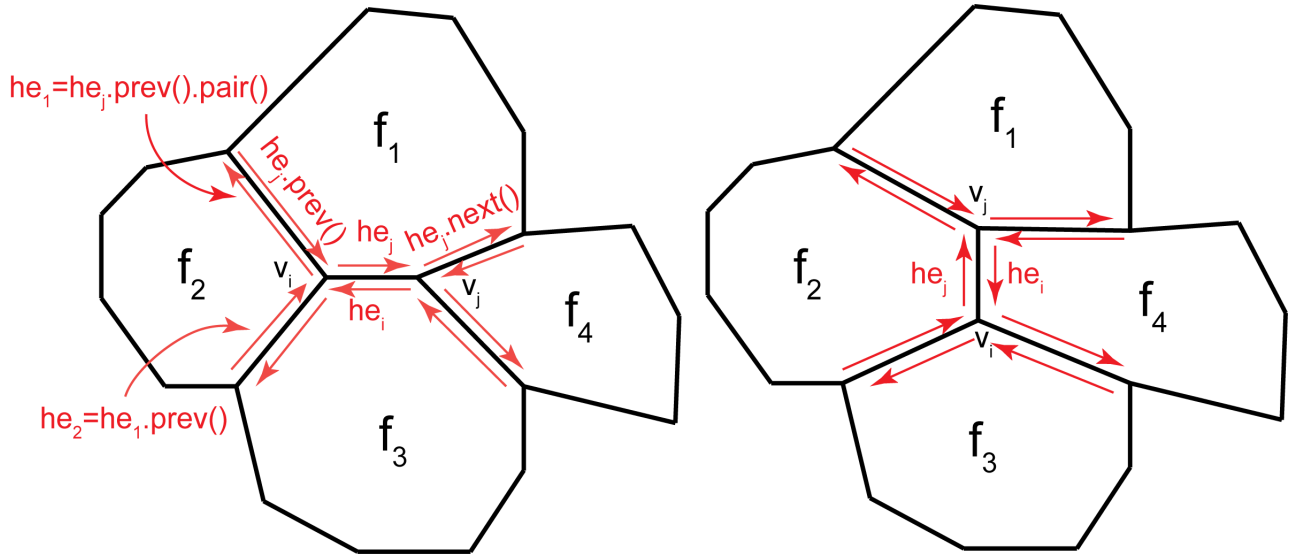
```
Vec ForceArea::compute(const Vertex<Property> &v, const HalfEdge<Property> &he) {
   Vec l = he.to()->r - v.r;
   const Face<Property> &f = *(he.face());
   const Face<Property> &fp = *(he.pair()->face());
   double A1 = _sys.mesh().area(f);
   double A2 = _sys.mesh().area(fp);
   double A0_1 = f.data().A0;
   double A0_2 = fp.data().A0;

   double kappa_1 = (f.outer) ? 0.0 : _kappa[f.data().face_type];
   double kappa_2 = (fp.outer) ? 0.0 : _kappa[f.data().face_type];

   Vec farea_vec = 0.5 * (kappa_1 * (A1 - A0_1) - kappa_2 * (A2 - A0_2)) * l.ez_cross_v();

   return farea_vec;
}
```

Topological changes



Focus only on the T1 transition.

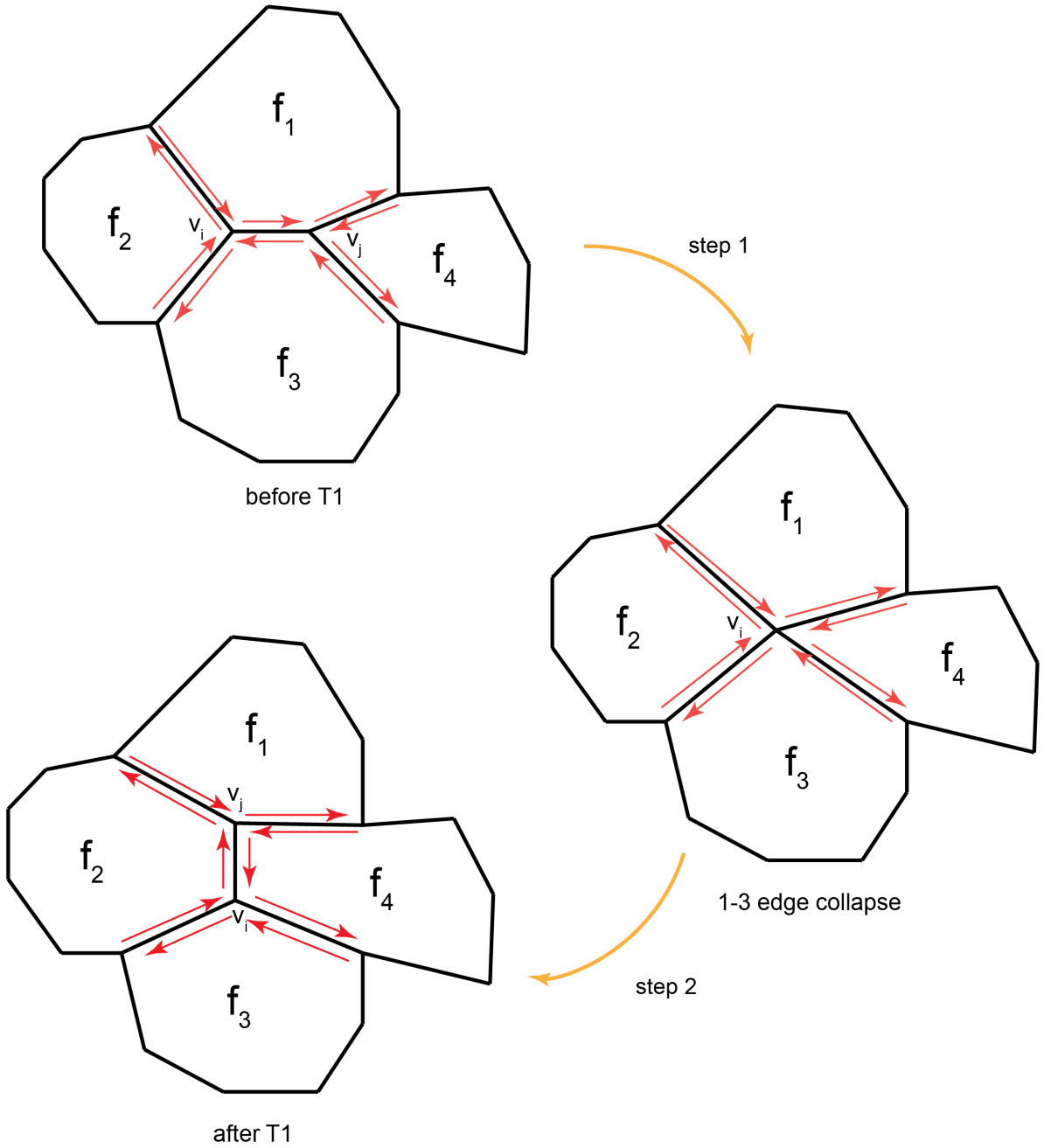
If an edge is shorter than l_{min} rotate it by $\pi/2$ and make a new edge of length $(1+\epsilon)l_{min}$.



before T1

after T1

An alternative approach (not implemented here, but available elsewhere)



```
bool Mesh<Property>::T1(Edge<Property> &e, double edge_len) {
  if (e.boundary)return false;
  HEHandle<Property> he = e.he();
  HEHandle<Property> hep = he->pair();
  VertexHandle<Property> v1 = he->from();
  VertexHandle<Property> v2 = he->to();
  Vec l = 0.5 * (v2->r - v1->r);
  Vec rc = v1->r + l;
  Vec rot_l = Vec(-l.y, l.x).unit();
  v1->r = rc - 0.5 * edge_len * rot_l;
  v2->r = rc + 0.5 * edge_len * rot_l;
  v1->he() = he;
  v2->he() = hep;
  HEHandle<Property> he1 = he->prev();
  HEHandle<Property> he2 = he->next();
  HEHandle<Property> he3 = hep->prev();
  HEHandle<Property> he4 = hep->next();
  he1->next() = he2;
  he2->prev() = he1;
  he3->next() = he4;
  he4->prev() = he3;
  he->next() = he1->pair();
  he->prev() = he4->pair();
  hep->next() = he3->pair();
  hep->prev() = he2->pair();
  he1->pair()->prev() = he;
  he2->pair()->next() = hep;
  he3->pair()->prev() = hep;
  he4->pair()->next() = he;
  he1->to() = v2;
  he1->pair()->from() = v2;
  he3->to() = v1;
  he3->pair()->from() = v1;
  he->face()->he() = he2;
  hep->face()->he() = he4;
  he->face() = he1->pair()->face();
  hep->face() = he2->pair()->face();
  he->face()->nsides = this->face_sides(*(he->face()));
  hep->face()->nsides = this->face_sides(*(hep->face()));
  return true;
}
```

Reading the initial configuration

Initial configuration is stored as in a JSON file:

- Box information
- List of vertices
  - Position
  - Type
  - Boundary
- List of faces
  - List of vertices (counterclockwise)
  - Type


```
"mesh": {
      "box": {
         "lx": 61,
         "ly": 53.693575034635195,
         "periodic": true
      },
  "faces": [
         {
            "A0": 0.8660254037844375,
             "outer": false,
            "type": "passive",
            "vertices": [2271, 4812, 4813, 3589, 2272, 2270]
         },…
  "vertices": [
         {
            "boundary": false,
            "constraint": "none",
            "id": 0,
            "r": [ 15.0, -10.96965511460289],
           "type": "regular"
         }, …
```


```
// Populate vertices
for (int i = 0; i < j["mesh"]["vertices"].size(); i++) {
   int id = j["mesh"]["vertices"][i]["id"];
   double x = j["mesh"]["vertices"][i]["r"][0];
   double y = j["mesh"]["vertices"][i]["r"][1];
   bool boundary = j["mesh"]["vertices"][i]["boundary"];
   _mesh.add_vertex(Vertex<Property>(id, Vec(x, y, _mesh.box()),
boundary, _mesh));
   Vertex<Property> &v = _mesh.vertices().back();
   this->add_vert_type(j["mesh"]["vertices"][i]["type"]);
   v.data().vert_type = _vert_types[j["mesh"]["vertices"][i]["type"]];
   v.data().type_name = get_vert_type_name(v.data().vert_type);
   v.data().constraint = j["mesh"]["vertices"][i]["constraint"];
}
```

Exposing it to Python

pybind11

(https://github.com/pybind/pybind11)

```
PYBIND11_MODULE(vm, m)
{
   VMTutorial::export_Vec(m);
   VMTutorial::export_Box(m);
   VMTutorial::export_VertexProperty(m);
   //…
}
```

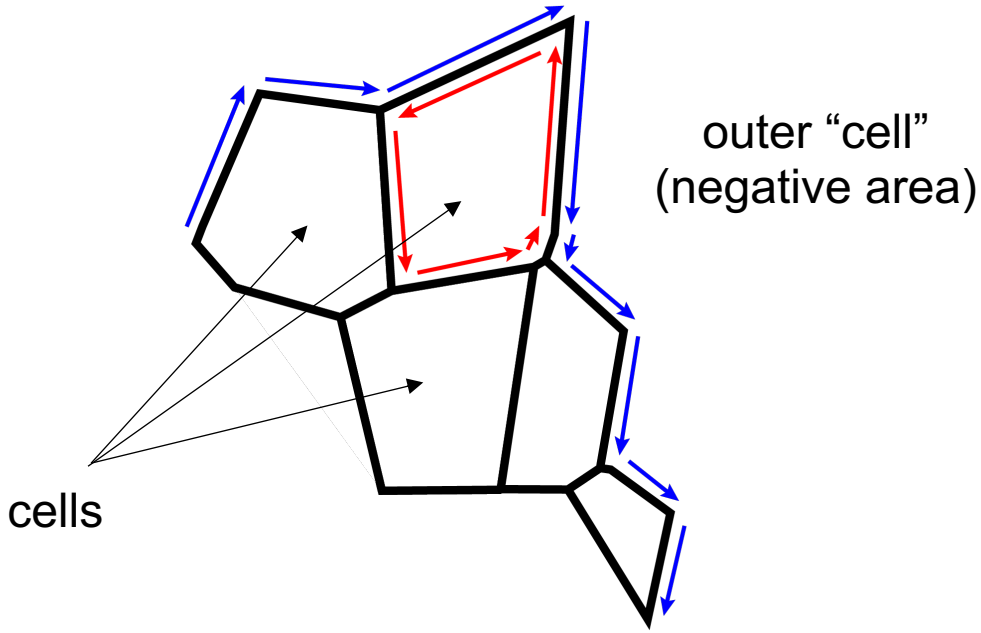
```
void export_Mesh(py::module& m) {
   py::class_<Mesh<Property>>(m, "Tissue")
   .def(py::init<>())
   .def("num_vert", &Mesh<Property>::num_vert)
   .def("set_cell_type", [](Mesh<Property>& m, int i, int type) { m.get_face(i).data().face_type = type; })
   .def("set_cell_A0", [](Mesh<Property>& m, int i, double A0) { m.get_face(i).data().A0 = A0; })
   .def("set_cell_P0", [](Mesh<Property>& m, int i, double P0) { m.get_face(i).data().P0 = P0; })
   .def("get_vertex", &Mesh<Property>::get_vertex , py::return_value_policy::reference)
   .def("get_junction", &Mesh<Property>::get_halfedge, py::return_value_policy::reference)
   .def("get_cell", &Mesh<Property>::get_face, py::return_value_policy::reference)
   .def("cells", &Mesh<Property>::faces, py::return_value_policy::reference)
   .def("get_cell_centre", [](Mesh<Property>& m, int i) -> Vec { return
m.get_face_centre(*(m.get_mesh_face(i))); })
   .def("cell_area", [](Mesh<Property> &m, int i) -> double { return m.area(*(m.get_mesh_face(i))); })
   .def("cell_perim", [](Mesh<Property> &m, int i) -> double { return m.perim(*(m.get_mesh_face(i))); });
}
```

PART IV: Preparing and running simulations

Building initial configurations

- It is not performance-critical, so we do it in Python
- To setup a mesh, one needs to know positions of all vertices and which vertices belong to which cell. Cell vertices are always ordered counterclockwise (except for outer edge).
- Mesh can be regular (hexagonal cells) or random (irregular cell shapes)
- Mesh can be open (free or constrained boundaries) or periodic.

Dealing with open boundaries



Example build tools located in VMTutorail/config_builder

```
from random_lattice import *
from make_mesh import *
import os

seed = int(os.urandom(4).hex(), base=16)
t = RandomLattice(400, 20, 20, 0.6204032394013997, seed =
seed)
t.build_periodic_lattice()
m = MakeMesh(t)
m.make_initial_configuration()
m.json_out('random_conf.json')
```

Controlling and running the simulation

Simulation is set up, controlled and run via a python script. It can be done either as a standalone script or in a Jupyter notebook.

Basic script:

```
from VMToolkit.VM import *
tissue  = Tissue()                                                  # initialise mesh
sim_sys = System(tissue)                                            # base object for the system
forces = Force(sim_sys)                                             # handles all types of forces
integrators = Integrate(sim_sys, forces, args.seed)                 # handles all integrators
topology = Topology(sim_sys, forces)                                # handles all topology changes
dumps = Dump(sim_sys, forces)                                       # handles all data output
simulation = Simulation(sim_sys, integrators, forces, topology)  # simulation object
sim_sys.read_input('some_inp_conf.json')
forces.add('area')          # add area force form term E = 0.5*kappa*(A-A0)^2
forces.add('perimeter')     # add perimeter force term from E = 0.5*gamma*P^2 + lambda*P (maybe -?)
forces.set_params('area', 'passive', {'kappa' : 1.0})
forces.set_params('perimeter', 'passive',  {'gamma': 0.25, 'lambda': 1.8})
topology.set_params({'min_edge_len': 0.05, 'new_edge_len': 0.055})
integrators.add('brownian')
integrators.set_dt(0.1)
for i in range(1000):
    if i % 10 == 0:
        dumps.dump_cells(f'cells_{i:08d}.vtp', draw_periodic=True)
        dumps.dump_mesh(f'mesh_{i:08d}.json')
    simulation.run(10)
```

Analysis of results

- Basic analysis can be done within the same execution script.
- For more involved analysis, it is advisable to save the data (several data formats are readily available, but adding new ones is simple), and process it separately.
- Typically, data analysis is done in Python, but more complicated analysis might require C/C++ code (beyond the scope of this tutorial).
- Visualisation is done by 3rd party software (Paraview and PyVista in this case).

What is missing…

- Support for three-dimensional vertex model.
- Proper documentation (generated from source comments, e.g. by Doxygen).
- Proper metadata handling (automatic generation of parameter files that could be used for full data provenance).
- A proper web-based GUI (e.g. ported in JavaScipt with WebAssmebly backend).

Summary

- Developed a basic two-dimensional vertex model code
- Emphasis on maintainable consistent design
- Use modern software design concepts (let the compiler do the work)
- Use libraries whenever possible (do not reinvent the wheel)
- Leave memory management to libraries (preferably the Standard Template Library)
- Modular approach (makes the code easy to extend and reduce a chance of bugs)

Code available at: https://github.com/sknepneklab/VMTutorial