Polynomial Inverse Mapping:

an acceleration strategy for modern Particle-in-Cell implementations

Yuri Chiucconi

Scuola Internazionale Superiore di Studi Avanzati, Trieste

August 31, 2021

Overview

PIC is a CFD technique (Evans & Harlow, 1957 [1]).

Modern algorithm (René Gasmöller et al., 2018 [2]):

- adaptive mesh refinement,
- parallel computations.

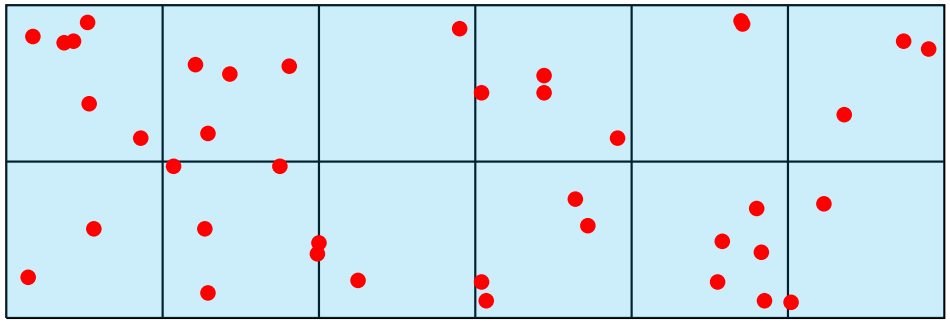
Our goal: speed up one of the main bottlenecks, i.e. $\forall K \in \mathcal{T}_h$ compute the inverse of

$$\Phi_K : \hat{K} \longmapsto K.$$

PIC concepts

PIC couples two viewpoint:

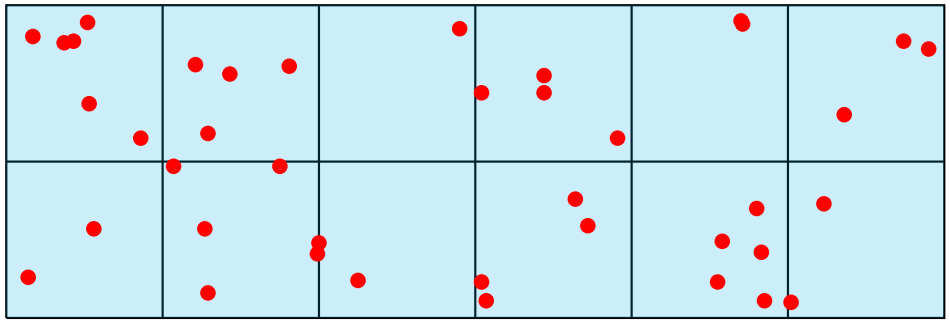
- discrete, using *particle*;
- continuous, mesh-based (*in-cell*).



PIC concepts

PIC couples two viewpoint:

- discrete, using *particle*;
- continuous, mesh-based (*in-cell*).

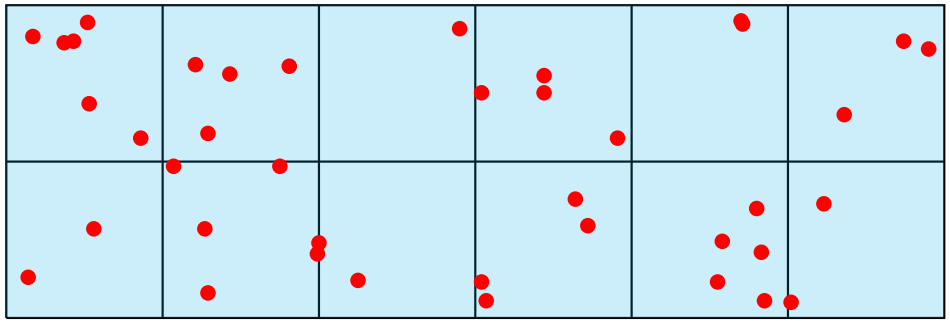


Transfer some continuous quantities into a set of discrete particles.

PIC concepts

PIC couples two viewpoint:

- discrete, using *particle*;
- continuous, mesh-based (*in-cell*).



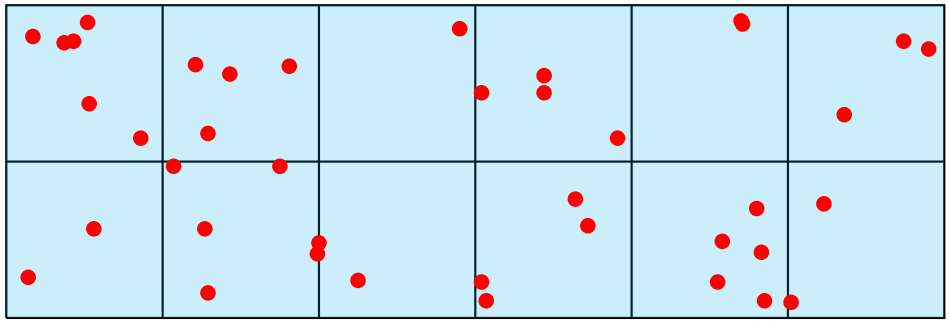
Transfer some continuous quantities into a set of discrete particles.

Get a set of (non-coupled) ODEs, i.e. the eq. of motion of each particle.

PIC concepts

PIC couples two viewpoint:

- discrete, using *particle*;
- continuous, mesh-based (*in-cell*).



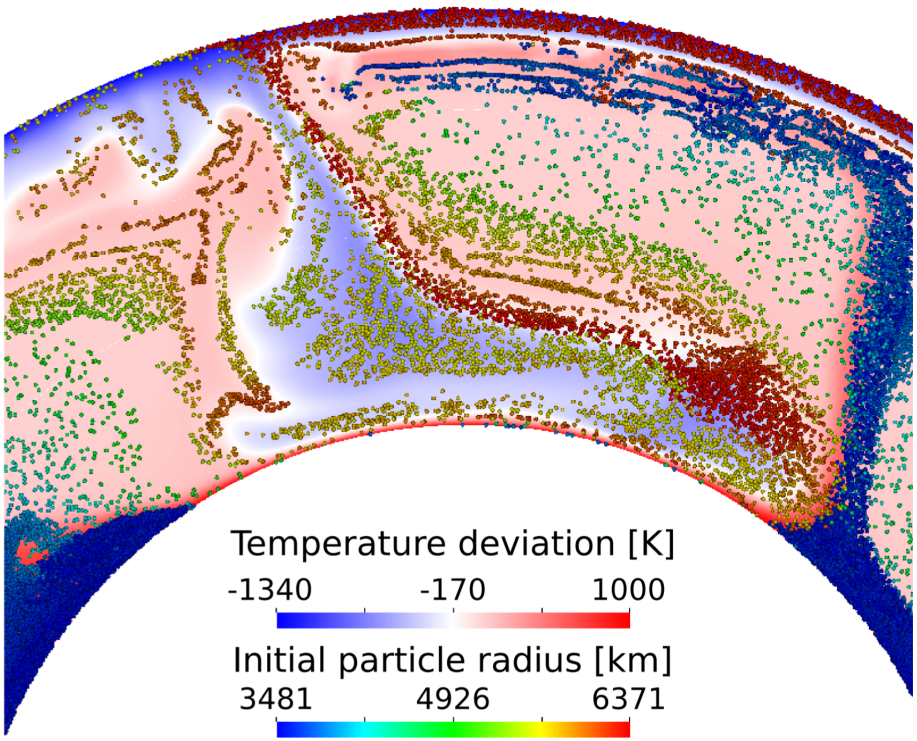
Transfer some continuous quantities into a set of discrete particles.

Get a set of (non-coupled) ODEs, i.e. the eq. of motion of each particle.

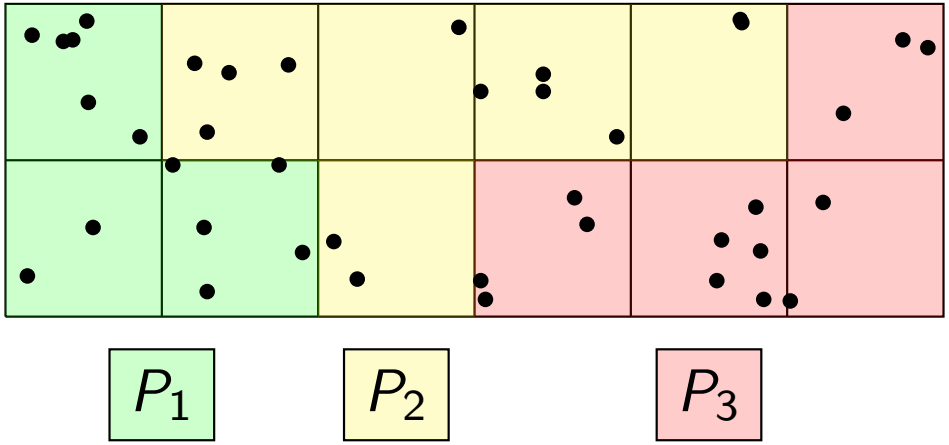
Advect particles along with the flow.

Why using particles

- Visualization of flows;
- tracking interfaces and origins;
- tracking history of material.



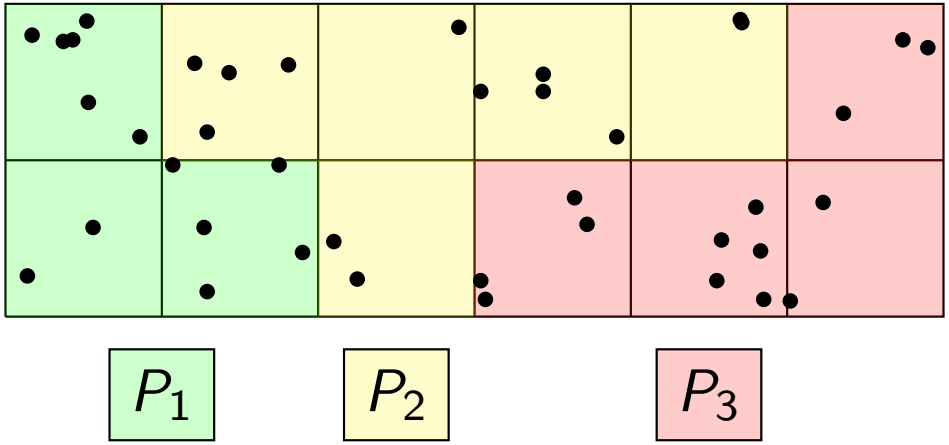
Data structures



For t fixed:

- each particle is inside one cell;
- the grid is fully distributed.

Data structures



For t fixed:

- each particle is inside one cell;
- the grid is fully distributed.

For t moving forward:

- particles move and change cell;
- cells are redistributed (load balancing).

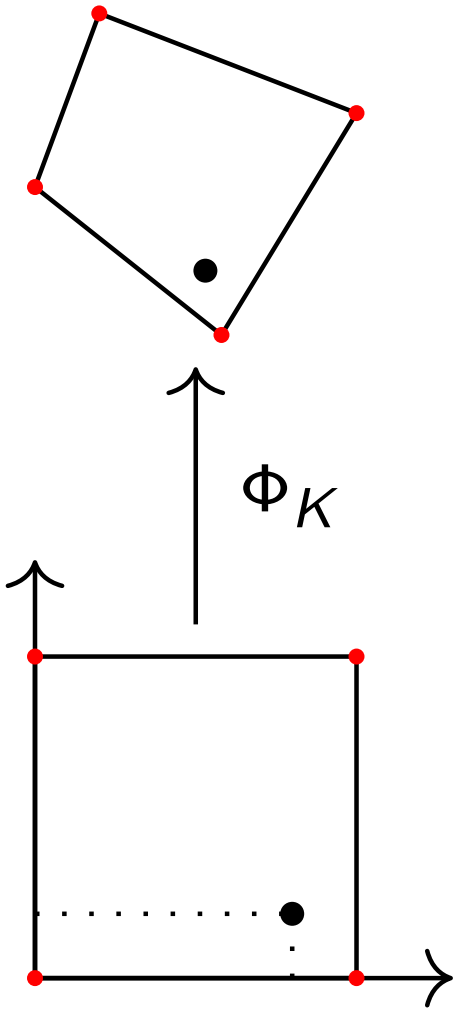
Data structures

C++ Particle object

```
template <int dim>
struct Particle
{
    ParticleId      p_id;
    CellId          cell_id;

    Point<dim>      location;
    Point<dim>      ref_location;

    double*         properties;
};
```



$$\text{ref_location} = \Phi_K^{-1}(\text{location}).$$

Is p inside K?

Suppose we know $p \in K_1 \cup \cdots \cup K_N$.

How do we find the only j s.t. $p \in K_j$?

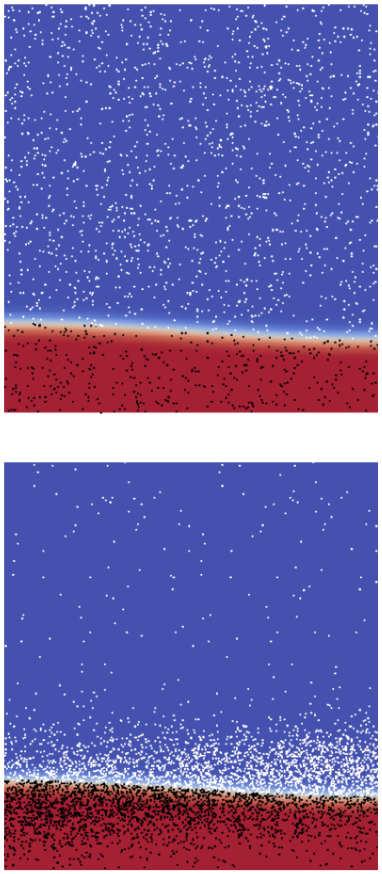
Is p inside K?

Suppose we know $p \in K_1 \cup \cdots \cup K_N$.

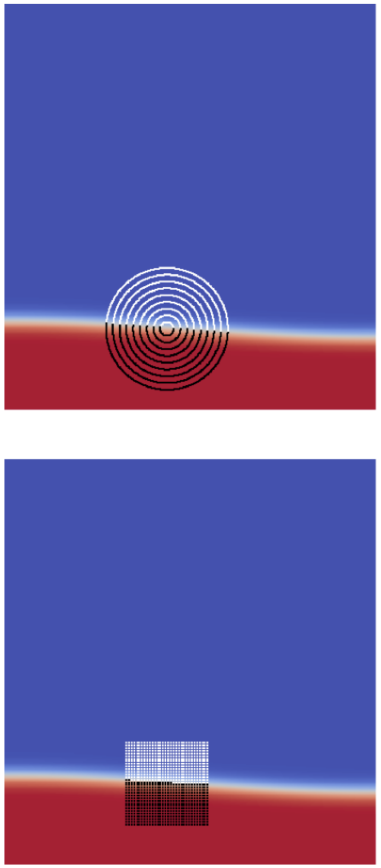
How do we find the only j s.t. $p \in K_j$?

$$p \in K_j \iff \Phi_{K_j}^{-1}(p) \in \hat{K}.$$

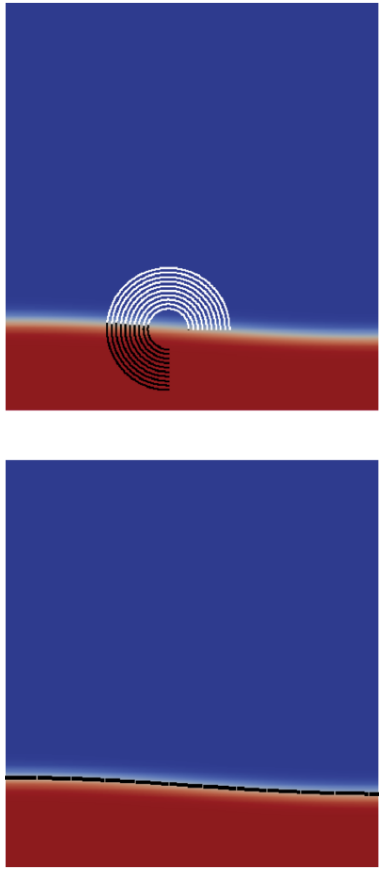
Particles generation



Statistical distribution



Regular distribution



User-defined pattern

Particles advection

$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{u}(\boldsymbol{x}, t).$$

Particles advection

$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{u}_h(\boldsymbol{x}, t).$$

- FE approximation;

Particles advection

$$\frac{d\boldsymbol{x}(t_n)}{dt} = \boldsymbol{u}_h(\boldsymbol{x}, t_n).$$

- FE approximation;
- discrete time step;

Particles advection

$$\frac{d\boldsymbol{x}(t_n)}{dt} = \boldsymbol{u}_h(\boldsymbol{x}, t_n).$$

- FE approximation;
- discrete time step;
- ODE integration scheme:
  - Forward Euler,
  - Runge-Kutta 2/4.

Particles advection

$$\frac{d\boldsymbol{x}(t_n)}{dt} = \boldsymbol{u}_h(\boldsymbol{x}, t_n).$$

- FE approximation;
- discrete time step;
- ODE integration scheme:
  - Forward Euler,
  - Runge-Kutta 2/4.

Evaluation of $\boldsymbol{u}_h^n$ at arbitrary positions is required.

Particles advection

After the advection, we need to find the current cell for each particle.

Particles advection

After the advection, we need to find the current cell for each particle.

There are four possibilities:

- in the same cell,
- in a locally owned cell,
- in a ghost cell,
- outside the subdomain known by the processor.

Mesh refinement

Two steps:

1. mesh locally refined or coarsened;
2. load balancing.

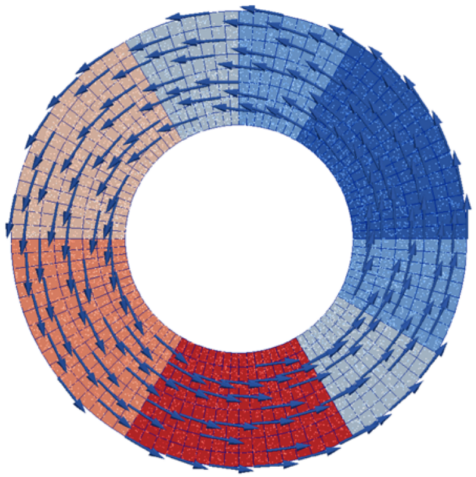
Mesh refinement

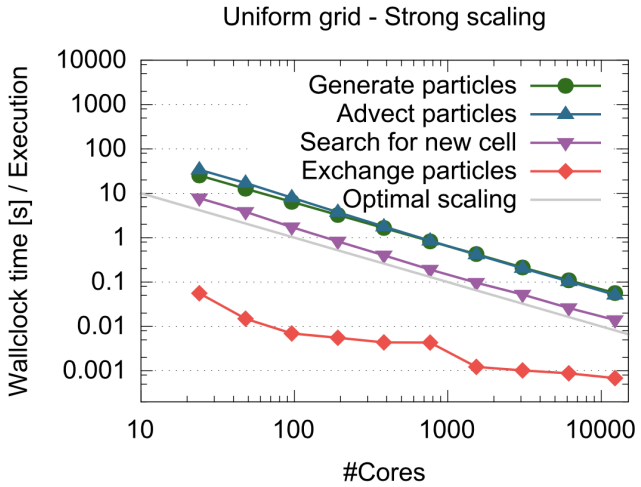
Two steps:

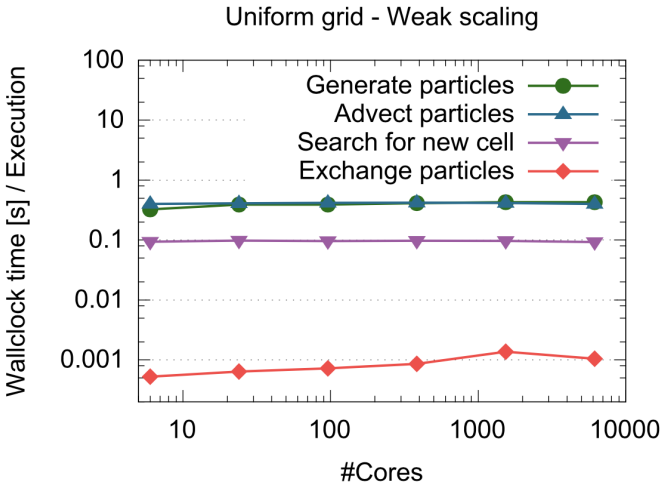
1. mesh locally refined or coarsened;
2. load balancing.

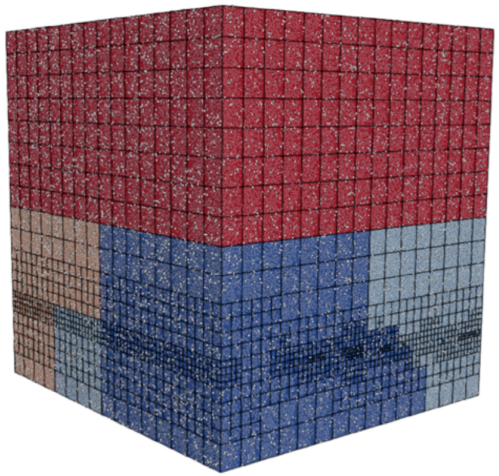
At the end of Step 1 we need to find current cell for each particle.

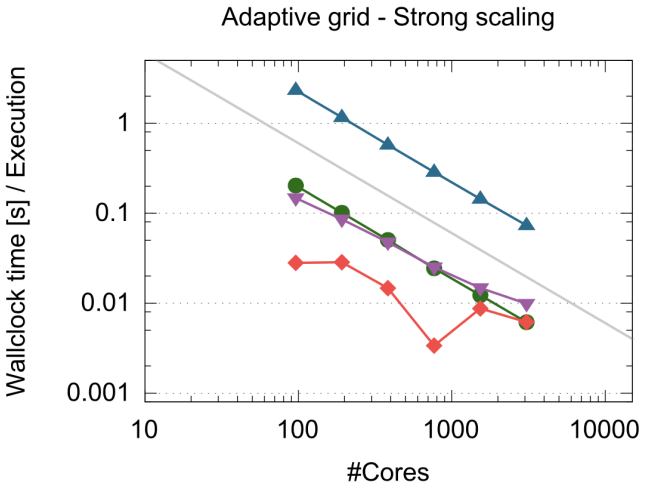
Scalability

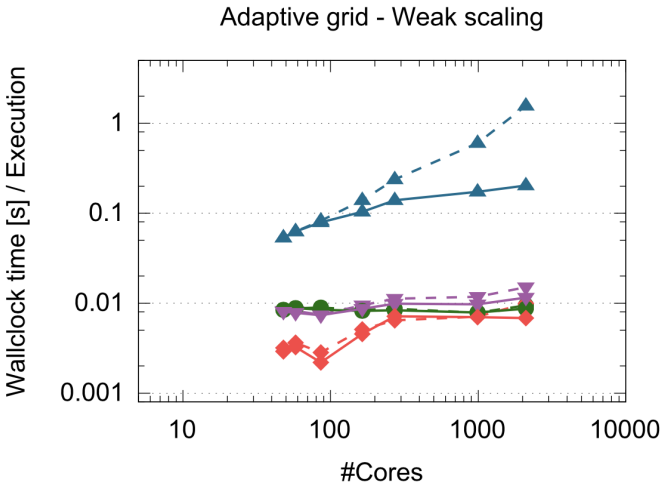












The inverse map problem

How do we compute $\Phi_K^{-1}(\boldsymbol{x})$?

- K hypercube $\longrightarrow$ Φ_K affine $\longrightarrow$ $\Phi_K^{-1}(\boldsymbol{x})$ trivial.
- K not hypercube $\longrightarrow$ Φ_K higher order polynomial $\longrightarrow$ $\Phi_K^{-1}(\boldsymbol{x})$?

The inverse map problem

How do we compute $\Phi_K^{-1}(\boldsymbol{x})$?

- K hypercube $\longrightarrow$ Φ_K affine $\longrightarrow$ $\Phi_K^{-1}(\boldsymbol{x})$ trivial.
- K not hypercube $\longrightarrow$ Φ_K higher order polynomial $\longrightarrow$ $\Phi_K^{-1}(\boldsymbol{x})$?

Newton-Raphson algorithm

- start ($i = 0$) from an *initial guess* $\boldsymbol{\alpha}^{(0)} \in \hat{K}$.
- **while** ($|\Phi_K(\boldsymbol{\alpha}^{(i)}) - \boldsymbol{x}| > \epsilon$):
  - $\boldsymbol{\alpha}^{(i+1)} = \boldsymbol{\alpha}^{(i)} - \mathbf{J}_{\Phi_K}^{-1}(\boldsymbol{\alpha}^{(i)}) \cdot (\Phi_K(\boldsymbol{\alpha}^{(i)}) - \boldsymbol{x})$,
  - i += 1;

Polynomial Inverse Mapping

We would like to:

- define Φ_K^{-1} not as an iteration but as a direct (polynomial) map

$$\Psi_K : K \longmapsto \hat{K},$$

  where possibly $\deg(\Phi_K) \neq \deg(\Psi_K)$;

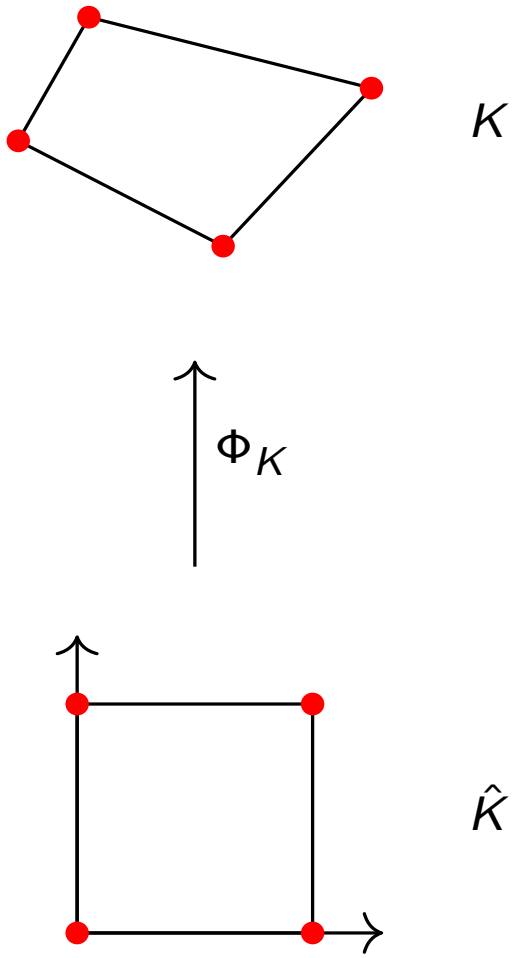
- exploit the off-line VS on-line paradigm.

The off-line part

Given $m \in \mathbb{N}$ (the *inverse map degree*) and given $\mathcal{T}_h$,

$\forall K \in \mathcal{T}_h$:

- find the bounding box $\mathcal{BB}_K$;
- define the support points of order m on $\mathcal{BB}_K$;
- find their pre-images using Newton (and store them).

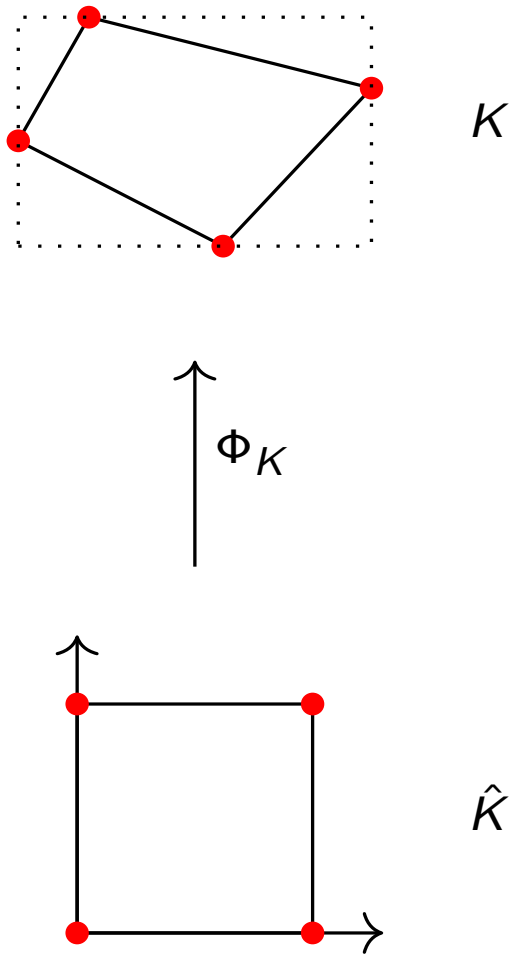


The off-line part

Given $m \in \mathbb{N}$ (the *inverse map degree*) and given $\mathcal{T}_h$,

$\forall K \in \mathcal{T}_h$:

- find the bounding box $\mathcal{BB}_K$;
- define the support points of order m on $\mathcal{BB}_K$;
- find their pre-images using Newton (and store them).

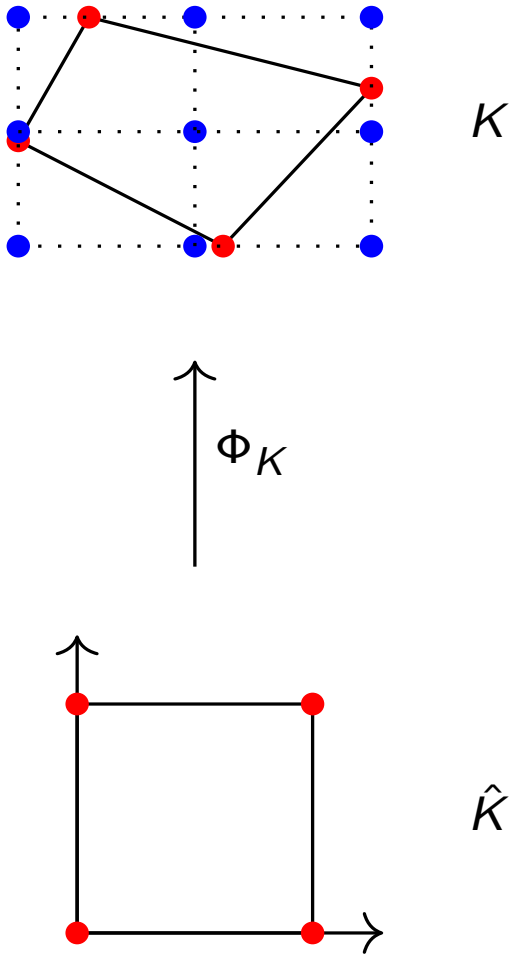


The off-line part

Given $m \in \mathbb{N}$ (the *inverse map degree*) and given $\mathcal{T}_h$,

$\forall K \in \mathcal{T}_h$:

- find the bounding box $\mathcal{BB}_K$;
- define the support points of order m on $\mathcal{BB}_K$;
- find their pre-images using Newton (and store them).

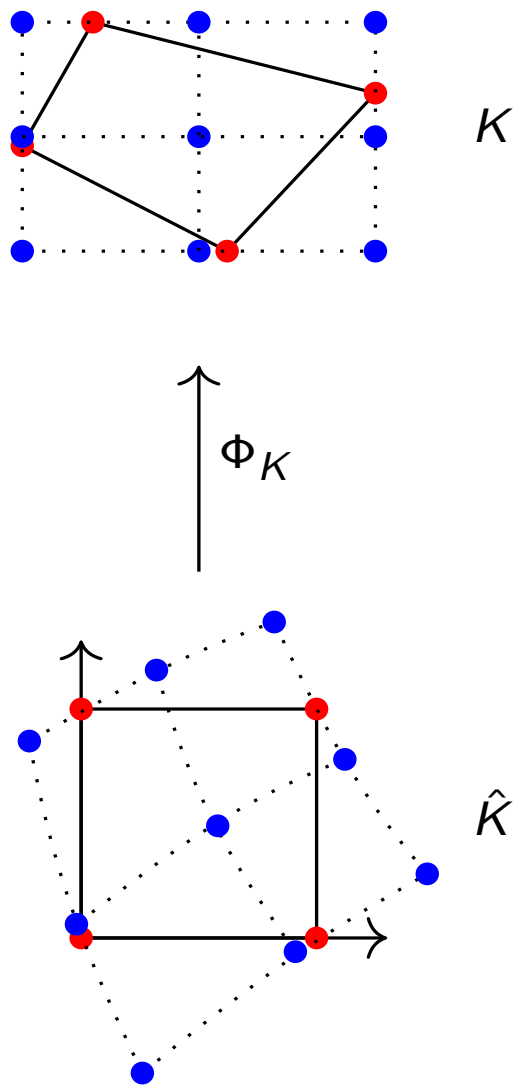


The off-line part

Given $m \in \mathbb{N}$ (the *inverse map degree*) and given $\mathcal{T}_h$,

$\forall K \in \mathcal{T}_h$:

- find the bounding box $\mathcal{BB}_K$;
- define the support points of order m on $\mathcal{BB}_K$;
- find their pre-images using Newton (and store them).

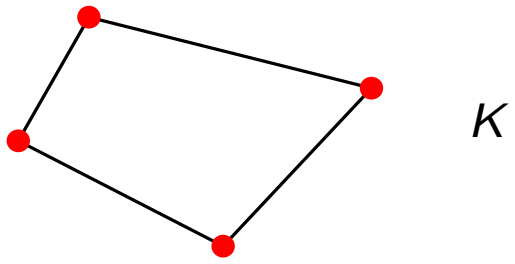


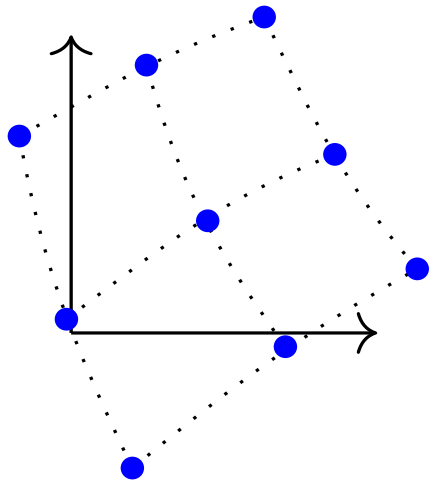
The off-line part

Given $m \in \mathbb{N}$ (the *inverse map degree*) and given $\mathcal{T}_h$,

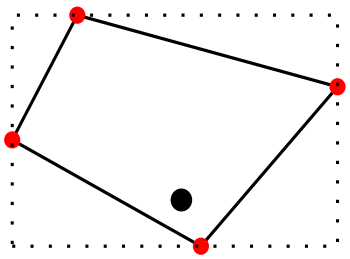
$\forall K \in \mathcal{T}_h$:

- find the bounding box $\mathcal{BB}_K$;
- define the support points of order m on $\mathcal{BB}_K$;
- find their pre-images using Newton (and store them).



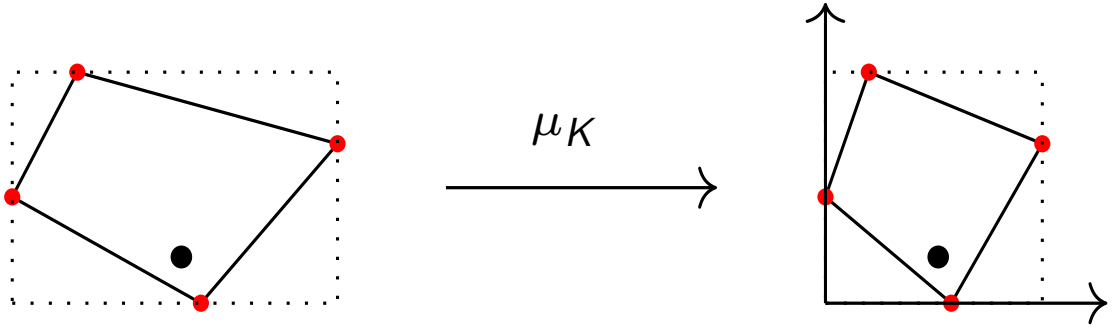


The on-line part



Given $p \in \mathcal{BB}_K$:

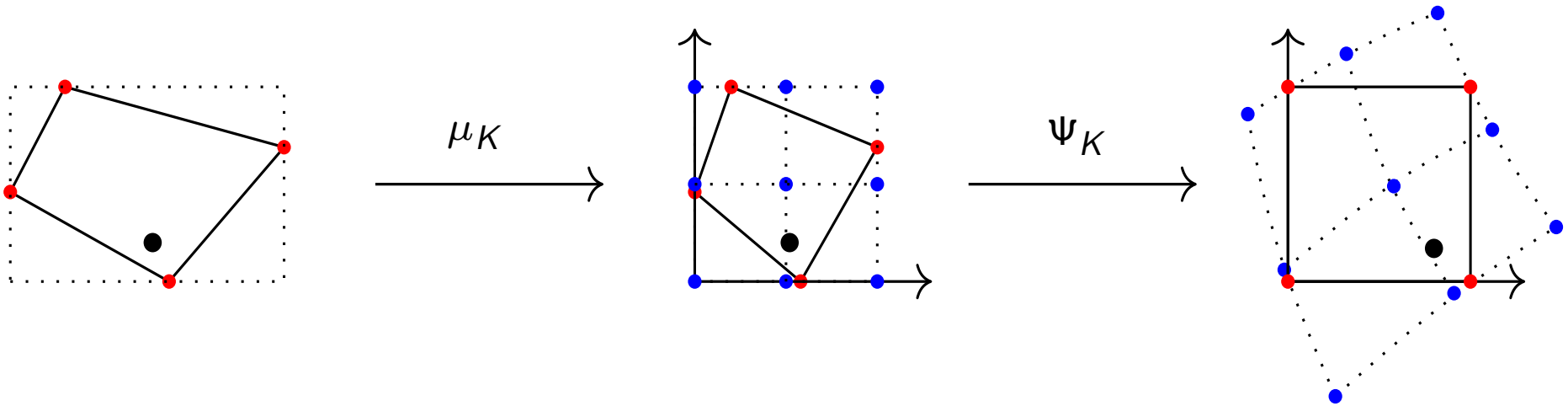
The on-line part



Given $p \in \mathcal{BB}_K$:

- compute $q = \mu_K(p)$, where $\mu_K : \mathcal{BB}_K \mapsto \hat{K}$;

The on-line part

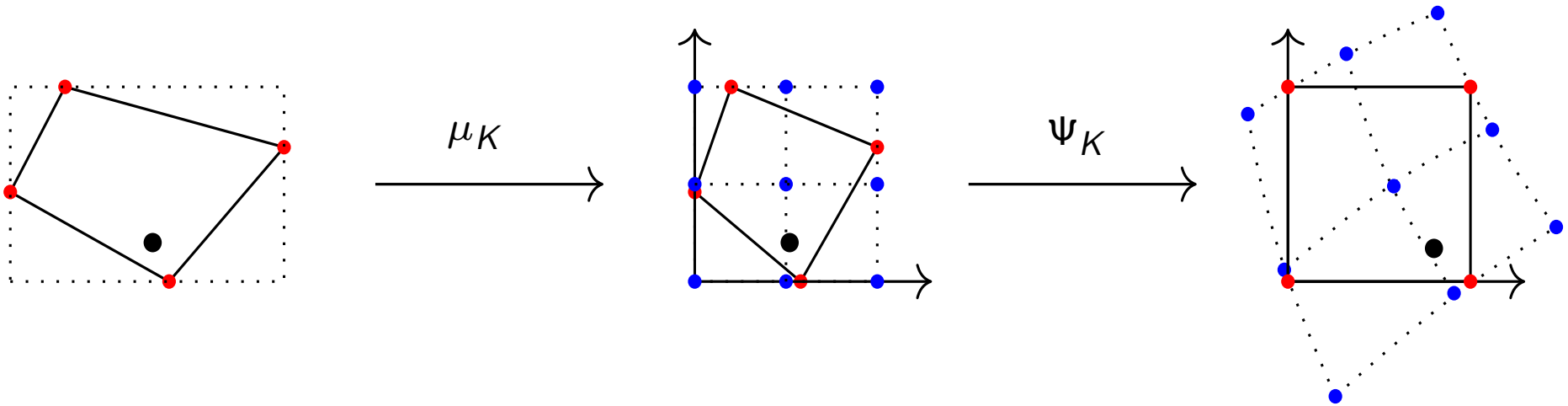


Given $p \in \mathcal{BB}_K$:

- compute $q = \mu_K(p)$, where $\mu_K : \mathcal{BB}_K \mapsto \hat{K}$;
- compute $\Psi_K(q)$, exploiting the off-line part.

Ψ_K is the polynomial function that maps blue dots to blue dots.

The on-line part



Given $p \in \mathcal{BB}_K$:

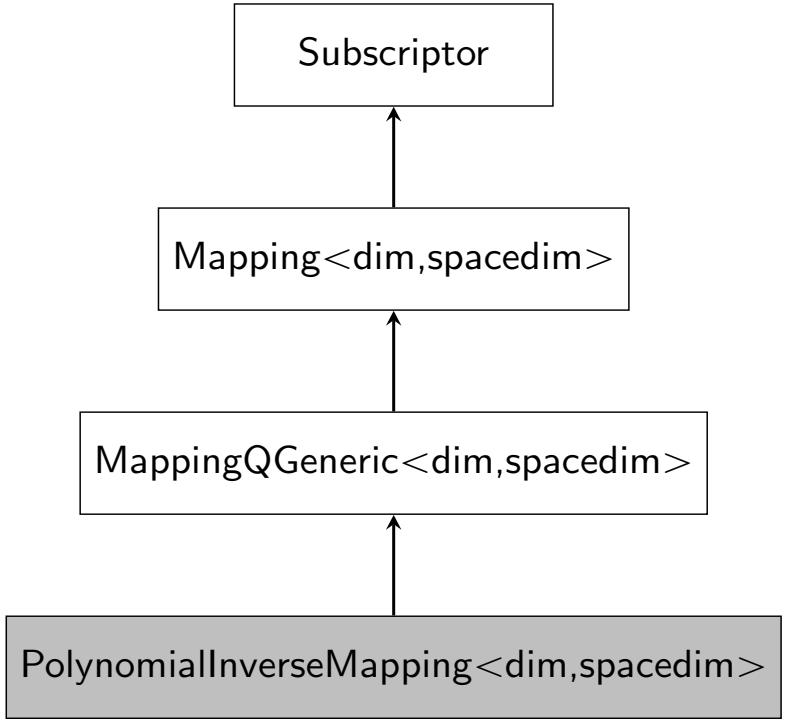
- compute $q = \mu_K(p)$, where $\mu_K : \mathcal{BB}_K \mapsto \hat{K}$;
- compute $\Psi_K(q)$, exploiting the off-line part.

Ψ_K is the polynomial function that maps blue dots to blue dots.

$\Longrightarrow$ if $\Psi_K(q) \in \hat{K}$ then $p \in K$, i.e. $\Phi_K^{-1} \doteq \Psi_K \circ \mu_K$.

PIM class: derivation in deal.ii

"[$\cdots$] This flexibility is conveniently mapped onto implementations in object-oriented programming languages by defining interface classes for each step. Concrete algorithms can then be implemented in the form of derived classes."



PIM class: public member functions

- Constructor

```
PolynomialInverseMapping(int forward_degree, int inverse_degree);
```

- Off-line part

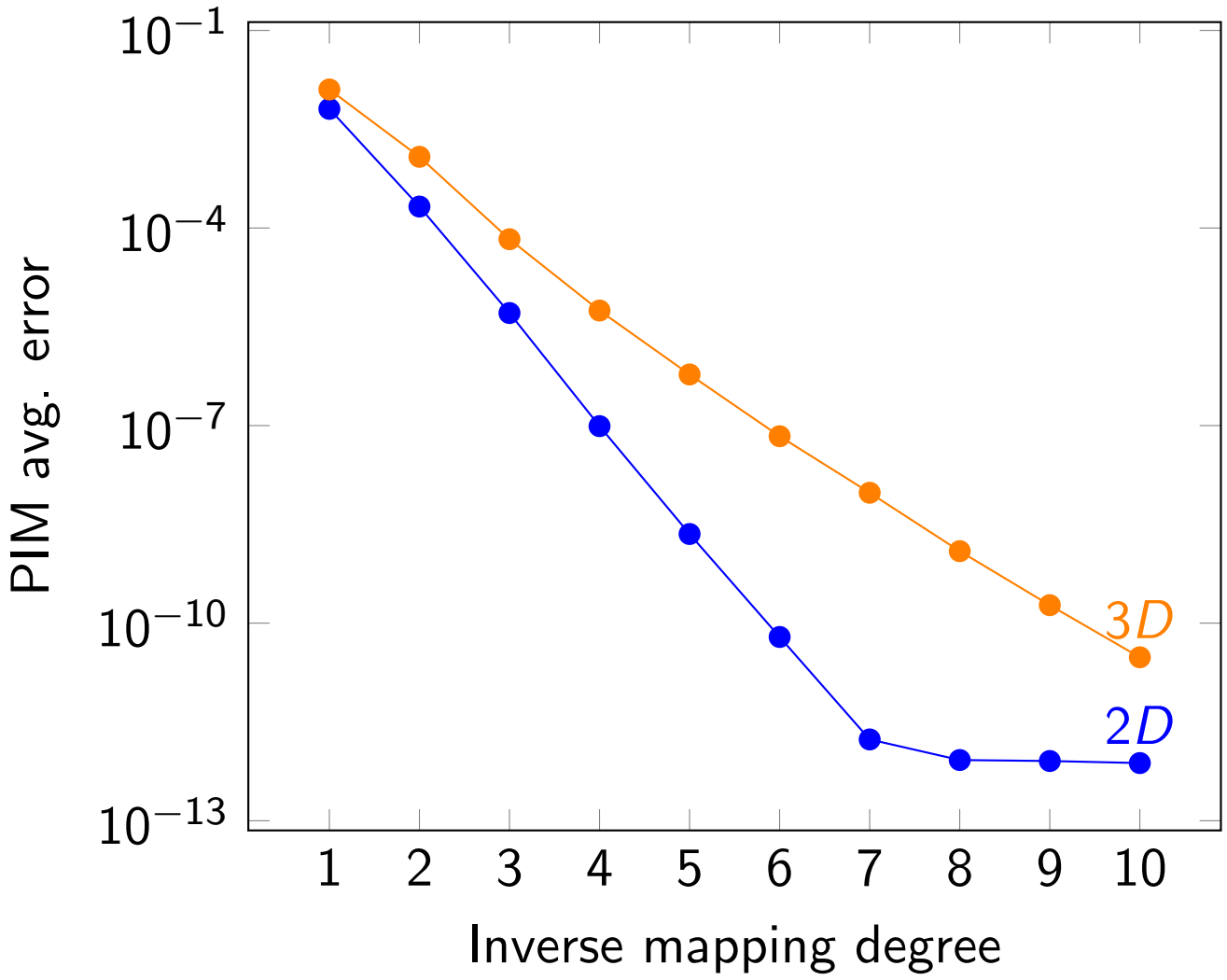
```
void initialize(const Triangulation<dim> &tria);
```

- On-line part

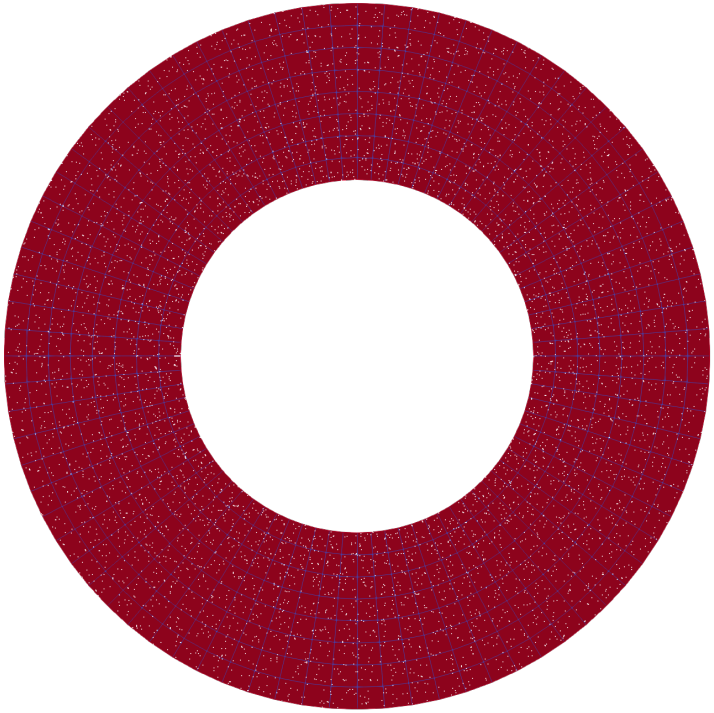
```
virtual
Point<dim> transform_real_to_unit_cell(
            const typename Triangulation<dim>::cell_iterator &cell,
            const Point<dim> &p
            ) const override;
```

PIM class: results (1)

Newton avg. error $\sim 10^{-13}$.



PIM class: results (2)

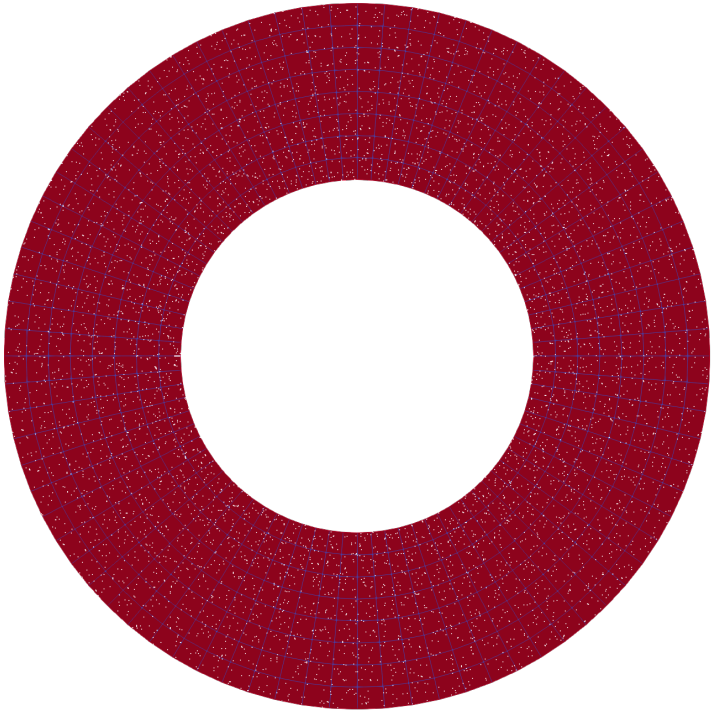


- Forward degree = 2,
- Inverse degree = 3.

Total CPU time elapsed since start : 0.509s.

(1) Newton: 0.0618s (12%),
(2) OFF-line: 0.134s (26%),
(3) IN-line: 0.122s (24%).

PIM class: results (3)



- Forward degree = 3,
- Inverse degree = 2.

Total CPU time elapsed since start : 0.518s.

(1) Newton: 0.0879s (17%),
(2) OFF-line: 0.133s (26%),
(3) IN-line: 0.0862s (17%).

Recap and conclusions

- PIC is a flexible and scalable method for modern CFD simulations.
- Polynomial Inverse Mapping algorithm (PIM) is an attempt to speed up one of the main bottleneck of PIC.
- Unfortunately, PIM is not working good as expected.

Recap and conclusions

- PIC is a flexible and scalable method for modern CFD simulations.
- Polynomial Inverse Mapping algorithm (PIM) is an attempt to speed up one of the main bottleneck of PIC.
- Unfortunately, PIM is not working good as expected.

FUTURE WORKS

Recap and conclusions

- PIC is a flexible and scalable method for modern CFD simulations.
- Polynomial Inverse Mapping algorithm (PIM) is an attempt to speed up one of the main bottleneck of PIC.
- Unfortunately, PIM is not working good as expected.

FUTURE WORKS

- Check for slow parts of the code and try to improve them.
- Try to exploit the last result we showed.

References

1. M. W. Evans, F. H. Harlow, and E. Bromberg. The particle-in-cell method for hydrodynamic calculations.Technical report, Los Alamos National Lab NM, 195

2. R. Gassmöller, H. Lokavarapu, E. Heien, E. G. Puckett, and W. Bangerth. Flexible and scalable particle-in-cell methods with adaptive mesh refinement for geodynamic computations. Geochemistry, Geophysics, Geosystems, 19(9):3596–3604, Sept. 2018

Thanks for your attention