TU Delft

Delft University of Technology

LEOPART

A particle library for FENICS

Maljaars, Jakob M.; Richardson, Chris N.; Sime, Nathan

DOI
10.1016/j.camwa.2020.04.023

Publication date
2021

Document Version
Final published version

Published in
Computers and Mathematics with Applications

Citation (APA)
Maljaars, J. M., Richardson, C. N., & Sime, N. (2021). LEOPART: A particle library for FENICS. *Computers and Mathematics with Applications*, *81*, 289-315. https://doi.org/10.1016/j.camwa.2020.04.023

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Computers and Mathematics with Applications 81 (2021) 289–315

Contents lists available at ScienceDirect

Computers and Mathematics with Applications

journal homepage: www.elsevier.com/locate/camwa

LEoPart: A particle library for FEniCS

Jakob M. Maljaars [a,b,*], Chris N. Richardson [c], Nathan Sime [d]

[a] Environmental Fluid Mechanics, Faculty of Civil Engineering and Geosciences, Delft University of Technology, Stevinweg 1, 2600 GA Delft, The Netherlands
[b] CRUX Engineering BV, Pedro de Medinalaan 3, 1086 XK Amsterdam, The Netherlands
[c] University of Cambridge BP Institute, Madingley Road, Cambridge, CB3 0EZ, UK
[d] Department of Terrestrial Magnetism, Carnegie Institution for Science, WA, DC, USA

ARTICLE INFO

Article history:
Available online 27 May 2020

Keywords:
Particle-in-cell
Finite elements
PDE-constrained optimization
Particle tracking
Open-source software
FEniCS

ABSTRACT

This paper introduces LEoPart, an add-on for the open-source finite element software library FEniCS to seamlessly integrate Lagrangian particle functionality with (Eulerian) mesh-based finite element (FE) approaches. LEoPart- which is so much as to say: 'Lagrangian–Eulerian on Particles' - contains tools for efficient, accurate and scalable advection of Lagrangian particles on simplicial meshes. In addition, LEoPart comes with several projection operators for exchanging information between the scattered particles and the mesh and *vice versa*. These projection operators are based on a variational framework, which allows extension to high-order accuracy. In particular, by implementing a dedicated PDE-constrained particle–mesh projection operator, LEoPart provides all the tools for diffusion-free advection, while simultaneously achieving optimal convergence and ensuring conservation of the projected particle quantities on the underlying mesh. A range of numerical examples that are prototypical to passive and active tracer methods highlight the properties and the parallel performance of the different tools in LEoPart. Lastly, future developments are identified. The source code for LEoPart is actively maintained and available under an open-source license at https://bitbucket.org/jakob_maljaars/leopart.
© 2020 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

Passive and active tracer methods find applications in a versatile range of engineering areas such as geophysical flows and environmental fluid mechanics [1–4], experimental fluid mechanics [5–7], and bio-medical applications [8,9], to name a few. In passive tracers methods, the Lagrangian particle motion is fully determined by the carrier flow and there is no feedback mechanism from the particles to the carrier flow. Such a feedback mechanism between the tracer particles and the carrier fluid is, however, typical to active tracer methods, in which particles mutually interact with the carrier fluid. To capture this interaction between the particles and the flow in numerical models, physical information carried by the tracer particles is used to estimate one or more additional source terms in the governing flow equations. In a discrete setting, this typically requires the reconstruction of mesh fields from the scattered particle data when the fluid flow equations are solved using a mesh-based approach, such as finite difference (FD), finite volumes (FV), or finite elements (FE). Application examples of active tracer methods include, among many others, the modeling of turbulent (reacting) flows [10,11], and

* Corresponding author at: Environmental Fluid Mechanics, Faculty of Civil Engineering and Geosciences, Delft University of Technology, Stevinweg 1, 2600 GA Delft, The Netherlands.
E-mail addresses: j.m.maljaars@tudelft.nl, maljaars@cruxbv.nl (J.M. Maljaars), chris@bpi.cam.ac.uk (C.N. Richardson), nsime@carnegiescience.edu (N. Sime).

https://doi.org/10.1016/j.camwa.2020.04.023
0898-1221/© 2020 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

mantle convection problems [4,12–14]. Alternatively, the particle information can be used to solve the advective part of physical transport phenomena, resulting in so-called *particle–mesh operator splitting schemes*, see, e.g., [15,16], and the earlier work by Maljaars et al. [17–19] from which LEoPart has evolved. Eliminating artificial dissipation by using Lagrangian particles for the discretization of the advection operator primarily motivates such methods, rendering the approach promising for simulating advection dominated flows [17,18] or free-surface flows [16,20].

In all the aforementioned methods and applications, particle-based and mesh-based discretization techniques essentially become intertwined. To render such a combination of Lagrangian particle methods in conjunction with mesh-based FD, FV, or FE solvers tractable for simulating practical engineering problems, a suite of dedicated, flexible and efficient tools is indispensable.

The open-source library LEoPart [21] which is presented in this paper provides such a toolbox by integrating particle functionality into the open-source FE library FEniCS [22]. LEoPart - which stands for 'Lagrangian–Eulerian on Particles' - contains utilities for efficiently and flexibly advecting and tracking a set of user-defined particles on simplicial meshes. In addition, LEoPart contains a suite of tools for projecting scattered particle data onto the mesh and *vice versa* in a high-order accurate, efficient and physically sound manner by implementing particle–mesh projection tools developed in the first author's recent work [17,18]. It is particularly the latter feature which sets LEoPart apart from the particle support in, e.g., PETSc [23] or the open-source particle library ASPECT [24,25] which is built on top of the finite element package deal.II [26]. The resulting combination of LEoPart and FEniCS is particularly suited for application to flow problems involving active or passive tracers, or to implement particle–mesh operator splitting schemes, as will be demonstrated by various numerical examples throughout.

The paper is structured as follows. Section 2 gives some background information on the encompassing FEniCS library, and provides a helicopter view of LEoPart. Section 3 describes the implementation of particles, as well as the advection and tracking of particles in LEoPart. Section 4 details the available particle–mesh interaction strategies. Particular attention is paid to the PDE-constrained particle–mesh interaction in Section 4.1.3, which enables the reconstruction of conservative mesh fields from a set of moving particles. Section 5 illustrates some example applications, meanwhile paying attention to the performance and scaling properties of LEoPart. Section 6 closes the paper by presenting conclusions and providing an outlook on future developments.

2. Implementation in FEniCS

2.1. A primer on FEniCS

FEniCS is a Finite Element (FE) framework, written in C++, with a Python interface. One of the major challenges of writing an FE code is that the computation which needs most user configuration is that of the local element tensor, which lies at the innermost part of the assembly loop. The local element tensor describes the matrix entries on each individual element, and relates to the physical equations of the system. FEniCS solves this problem by allowing the user to write these equations in a Domain Specific Language (DSL), which is then automatically compiled to C code to be called by the assembly loop.

In addition to simplifying the construction of the local element tensor, FEniCS makes it easier to run in parallel using MPI. Using the Python interface, running FEniCS code with MPI can be as simple as `mpirun -n 32 python3 demo.py`. The mesh will be partitioned into 32 chunks, and the problem will be distributed across 32 processes for this job.

FEniCS consists of several components:

- Unified Form Language (UFL) - providing the DSL component
- FEniCS Form Compiler (FFC) - which compiles the DSL into C code
- FIAT - the finite element tabulator
- DOLFIN - the main C++/Python package, which integrates I/O, assembly and solvers

The following Python script shows how the DOLFIN interface can simplify the expression of an FE problem and computation of its solution. Consider the weak form of the Poisson equation defined on the unit square $\Omega := (0,1)^2$: find $u \in V$ subject to homogeneous Dirichlet boundary conditions, where V is the appropriate solution space, such that

$$\int_\Omega \nabla u \cdot \nabla v \mathrm{d}\Omega = \int_\Omega v \sin(x) \mathrm{d}\Omega \quad \forall\, v \in V. \tag{1}$$

Python code

```
from dolfin import *
mesh = UnitSquareMesh(20, 20)
V = FunctionSpace(mesh, "Lagrange", 1)
u, v = TrialFunction(V), TestFunction(V)
a = inner(grad(u), grad(v))*dx
f = Expression("sin(x[0])", degree=2)
L = f*v*dx
u = Function(V)
bc = DirichletBC(V, 0.0, "on_boundary")
solve(a==L, u, bc)
```

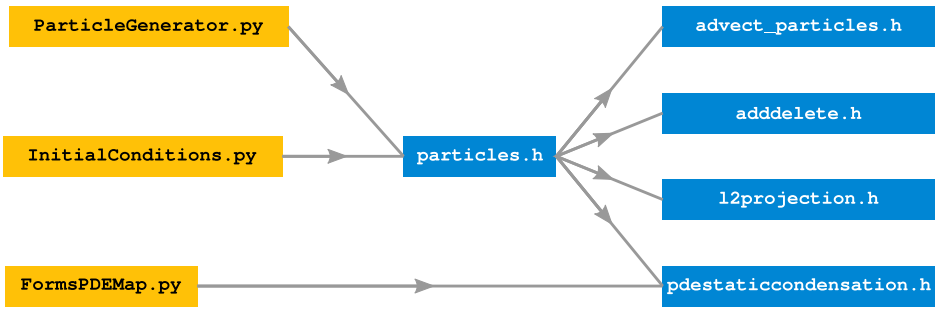


Fig. 1. Code structure LEoPart.

Calling the `solve()` method runs the form compiler (FFC) and compiles the symbolic expressions into C code, which is compiled and loaded into memory. A global matrix equation is then assembled using this generated code for the local element tensor, and finally an LU solver is called to solve the resulting system of equations. Whilst this example is simple and compact, many options exist to expand each part of the problem, for example by applying Dirichlet boundary conditions, or assembling matrix and vector separately, and choosing more sophisticated solvers.

For larger problems, it is important to run in parallel using MPI. Mesh partitioning is performed using PT-SCOTCH or ParMETIS, and there is support for the HDF5 file format, which allows parallel access of large datasets. Third party libraries are used throughout, wherever possible: PETSc is the linear algebra backend of choice, with a large selection of parallel solvers available.

FEniCS is an open-source package, and is available for various platforms. The latest information can be found on the project website `www.fenicsproject.org`.

2.2. LEoPart code structure

LEoPart is built on top of the FEniCS package, and adds new concepts for the advection of particles on simplicial meshes, and the interaction between particles and the mesh. A central paradigm in the design of LEoPart is that it serves as an add-on to FEniCS, using the existing FEniCS tool chain wherever possible. As a result, all the FEniCS functionality remains available for the user. In particular, LEoPart is designed such that it seamlessly integrates with the mesh partitioning in FEniCS facilitating parallelism using MPI.

To provide a fast and user friendly suite of tools, LEoPart wraps C++ code in Python using `pybind`[1] for the computationally demanding parts such as the particle advection and the matrix assembly. Particle pre-processing and post-processing is done in Python. Fig. 1 provides an overview of the different core components in LEoPart.

The remainder of this paper essentially discusses the different components from Fig. 1. Particular attention is paid to the particles class, the advection of the particles, and the tools for the interaction between the particles and the mesh via an ℓ^2-projection or a PDE-constrained projection.

As an aside, we note that LEoPart also contains a Stokes solver, implementing the H(div) conforming hybridized discontinuous Galerkin (HDG) formulation from Rhebergen & Wells [27,28]. Implementation details of this solver are beyond the scope of this paper.

The source code for LEoPart as well as the examples that are shown in what follows, are hosted under an open-source license at `https://bitbucket.org/jakob_maljaars/leopart`.

3. Particle functionality

This section explains the implementation of particles in LEoPart and discusses the particle advection and particle tracking strategy on simplicial meshes.

3.1. Particle initialization

The `particles` class forms the backbone for dealing with the Lagrangian particles in LEoPart. Operations such as particle advection and the particle–mesh projections require as input an instance of this `particles` class, see Fig. 1.

Each particle is assumed to have at least a spatial coordinate attached, which henceforth is denoted by $\mathbf{x}_p$ for a single particle. Moreover, it is presumed that particles always live in a spatial domain, denoted Ω, so that the particle coordinate set is defined as

$$\mathcal{X}_t := \{\mathbf{x}_p(t) \in \Omega\}_{p=1}^{N_p}, \tag{2}$$

[1] `https://github.com/pybind/pybind11`.

with N_p the total number of particles. For notational convenience, we also make use of the index set of particles and the index set of particles hosted by cell K, at a fixed time instant t, which are defined as

$$\mathcal{S}_t \coloneqq \{p \in \mathbb{N} \,:\, \mathbf{x}_p(t) \in \mathcal{X}_t\}, \tag{3}$$

$$\mathcal{S}_t^K \coloneqq \{p \in \mathbb{N} \,:\, \mathbf{x}_p(t) \in \overline{K},\ \mathbf{x}_p(t) \in \mathcal{X}_t\}. \tag{4}$$

Whereas carrying a spatial coordinate might be sufficient for passive particle tracing, additional properties, such as density, concentration, or momentum values, need to be attached to the particles for active particle tracing. For a scalar and vector valued property, such particle quantities are defined as

$$\Psi_t \coloneqq \left\{\psi_p(t) \in \mathbb{R}\right\}_{p=1}^{N_p}, \tag{5}$$

$$\mathcal{V}_t \coloneqq \left\{\mathbf{v}_p(t) \in \mathbb{R}^d\right\}_{p=1}^{N_p}, \tag{6}$$

respectively, where $d = 1, 2, 3$ is the spatial dimension.

LEoPart provides a number of particle generators via `ParticleGenerator.py` to generate a set of point coordinates. Most of the available particle generators create random point locations within a geometric object such as a rectangle (`RandomRectangle`), a circle (`RandomCircle`), and a sphere (`RandomSphere`). These particle generators enable generating particles on parts of the meshed domain, but have the drawback that the point coordinates are generated on one processor and broadcasted to the other processors in parallel computations. This replication on MPI-ranks is circumvented in the `RandomCell` class, which generates random point coordinates within the simplicial (i.e. triangular or tetrahedral) cells of the mesh. For a tetrahedron, random barycentric coordinates within a cell are generated with the algorithm from [29], i.e.

Python code

```python
def _random_bary():
    # Create uniform random numbers
    s = np.random.random()
    t = np.random.random()
    u = np.random.random()

    # Fold space in cube into tetrahedron
    if (s + t) > 1.0:
        s, t = 1.0 - s, 1.0 - t
    if (s + t + u) > 1.0:
        if (t + u) > 1.0:
            t, u = 1.0 - u, 1.0 - s - t
        else:
            s, u = 1.0 - t - u, s + t + u - 1.0
    v = 1.0 - s - t - u
    return (s, t, u, v)
```

Multiplying `s, t, u, v` by the vertex coordinates returns after summation a random point within a tetrahedral cell. Since LEoPart inherits the domain decomposition of the mesh from DOLFIN, the `RandomCell` particle generator takes advantage of the mesh sub-domains, and only creates point coordinates that are within processor boundaries.

The set of Lagrangian particles which is formed by the coordinate set $\mathcal{X}_t$, and an arbitrary number and ordering of scalar and/or vector valued particle quantities is used to instantiate the `particles` class in LEoPart. Upon instantiation of this class, the hosting cell for a particle is found via a cell collision check that is available in FEniCS via `BoundingBoxTree::compute_first_entity_collision`. As soon as the initial hosting cell is known, LEoPart uses a more efficient algorithm for tracking moving particles on the mesh, see Section 3.2.2. The coordinates of a point, together with the optional properties, constitute a `particle`, which is defined in LEoPart as an array of `dolfin::Points`, i.e.

C++ code

```cpp
// Define the particle atom as a vector of dolfin points
namespace dolfin
{
typedef std::vector<Point> particle;
}
```

The first element in this array is always populated by the particle position. Using a `dolfin::Point` for the representation of the particle position allows to conveniently use other DOLFIN functionalities. However, this design choice also restricts the possibilities for the definition of the particle properties. A particle can, for example, neither carry vector valued properties for which the length exceeds three, nor can tensor properties be defined at the particles.

From the user's perspective, instantiating the `particles` class on a unit square mesh from a user-defined coordinate array, together with arrays for a scalar- and a vector-valued property is done as

Python code

```python
from dolfin import UnitSquareMesh
from leopart import particles
import numpy as np

msh = UnitSquareMesh(2, 2)

# Create 4 particles in 2D
xp = np.random.rand(4, 2)
psip = np.zeros(xp.shape[0])
vp = np.zeros(xp.shape)

# Instantiate particles
p = particles(xp, [psi_p, v_p], msh)
```

For the sake of generality it is noted that the ordering and the length of the list with the particle quantities, i.e. `[psi_p, v_p]` in the above example is arbitrary.

3.2. Particle advection

Three different particle advection schemes are currently supported by LEoPart. These advection schemes solve the system of ODEs: given a vector-valued velocity field $\mathbf{a}_h$, solve $\forall\, p \in \mathcal{S}_t$

$$\frac{\mathrm{d}\mathbf{x}_p}{\mathrm{d}t} = \mathbf{a}_h(\mathbf{x}_p, t) \tag{7a}$$

$$\frac{\mathrm{d}\psi_p}{\mathrm{d}t} = 0, \tag{7b}$$

$$\frac{\mathrm{d}\mathbf{v}_p}{\mathrm{d}t} = 0, \tag{7c}$$

where a particle can carry an arbitrary number of scalar- and/or vector-valued quantities that will stay constant throughout the particle advection. The `advect_particles` class solves Eq. (7) with a first-order accurate Euler forward method, and the two and three stage Runge–Kutta methods are available via the `advect_rk2` and the `advect_rk3` classes, respectively. The two multi-stage Runge–Kutta advection schemes inherit from the `advect_particles` class, and a typical constructor for the latter reads

C++ code

```cpp
// Constructor advect_particles
advect_particles(particles& P, FunctionSpace& U, Function& uhi,
           const MeshFunction<std::size_t>& mesh_func);
```

This snippet shows that the particle advection classes require a `particles` instance, a velocity field specified in the `Function` and its corresponding `FunctionSpace`, and a `MeshFunction` for marking the boundaries, see Section 3.3. A complete overview of all the particle advection constructors is found in `advect_particles.h`.

3.2.1. Cell–particle connectivity and particle relocation

Imperative for both the particle advection, as well as the particle–mesh interactions later on, is the evaluation of mesh fields at a potentially large number of points inside the domain. In order to do so, it must be known which cell is hosting the particle. At a meta-level, two options are available to fit this purpose. The first option is that each particle carries a reference to its hosting cell in the mesh. As soon as a particle crosses a cell boundary, this particle-to-cell reference is updated. Alternatively, a cell can be considered as a bucket filled with particles. A particle is removed from the cell's 'particle bucket' as soon as it escapes the cell, and added to the receiving cell's particle bucket. Rather than a particle keeping track of its hosting cell, the bookkeeping is done at the cell level, i.e. each cell contains a list of particles.

The latter method is used in LEoPart, as this enables efficient evaluation of a mesh-field at the particle positions and allows to conveniently use the FEniCS mesh partitioning for storing the particle data on the different processes. Central to the method is the cell–particle connectivity table, for which LEoPart uses

C++ code

```cpp
std::vector<std::vector<particle>> _cell2part;
```

Related to the advection stage, that will be discussed in more detail below, this `_cell2part` structure can be updated with the `particles::relocate` method, for which the declaration reads

C++ code

```cpp
// Relocate particles with known relocation data. Each entry is {cidx, pidx, cidx_recv} using
// numeric_limits::max for cidx_recv to send to another process or crossing an open boundary
void relocate(std::vector<std::array<std::size_t, 3>>& reloc);
```

This method is run once per advection step - or once per sub-step for the multistage advection schemes - and takes as input a relocation array `reloc` of particle indices (`pidx`) that should be relocated from one cell (`cidx`) to another (`cidx_recv`). For each entry in the `reloc` array, LEoPART either copies the particle to a particle collector if the particle escaped through a boundary or creates and pushes a particle to the receiving cell index via

C++ code

```
const std::size_t& cidx = r[0];
const std::size_t& pidx = r[1];
const std::size_t& cidx_recv = r[2];

if (cidx_recv == std::numeric_limits<unsigned int>::max())
{
  if (mpi_size > 1)
    particle_communicator_collect(cidx, pidx);
}
else
{
  particle p = _cell2part[cidx][pidx];
  _cell2part[cidx_recv].push_back(p);
}
```

Once this relocation is done, the particles that needed relocation are erased from the old cell index via `particles::delete_particle`:

C++ code

```
// Remove ith particle from cell c
void delete_particle(int c, int i)
{
_cell2part[c].erase(_cell2part[c].begin() + i);
}
```

In view of this relocation method, the following remarks are made:

- A potential improvement could be to let the cell–particle connectivity `_cell2part` store a list of particle indices per cell rather than a list of `particle` objects, and store the particles within a process in a separate array. This obviates the need to create and erase a `particle` within `_cell2part` when it escapes to another cell on the same process.
- The particle communication between processors via the collection step and a subsequent pushing step will be further detailed when discussing the implementation of internal boundaries in Section 3.3.1.
- An efficient way of finding the new hosting cell `cidx_recv` given the current host `cidx` is crucial from a performance perspective. The implementation of a fast particle tracking algorithm is discussed next.

3.2.2. Particle tracking

A challenge that is specifically related to the particle advection, is to efficiently keep track of the hosting cell for the Lagrangian particles in the unstructured simplicial mesh. Several procedures have been developed in literature, such as superposition of a coarse Cartesian mesh onto the unstructured mesh [30], the tetrahedral walk method [31], or methods based on barycentric interpolation [32]. An alternative method is the convex polyhedron method [33], which assumes that the mesh consists of convex polyhedral cells. This indeed is the case for the simplicial cells used in FEniCS. For each *facet* in the mesh, the midpoint, the unit normal, and the connectivity are pre-computed. Concerning the connectivity, a facet has two neighboring cells for facets internal to the mesh, or only one neighboring cell for exterior boundary facets and internal facets which are located on processor boundaries. From the perspective of a mesh cell, indicated by K, the sign of the facet unit normals is adapted so as to make sure that they are always outwardly directed.

The convex polyhedron particle tracking then proceeds as follows for a particle p, which at time t is located at $\mathbf{x}_p(t)$ within cell K_0, see Fig. 2 for a principle sketch. Assume the particle has a velocity $\boldsymbol{a}\left(\mathbf{x}_p, t\right)$ and the time step used for advecting the particle is $\Delta t^{(0)} = \Delta t$, where the use of a superscript will become clear shortly. As the first step in the particle tracking algorithm, the time to intersect the ith facet of element K_0 is computed as $\Delta t_i = b_i/(\boldsymbol{a}\left(\mathbf{x}_p, t\right) \cdot \boldsymbol{n}_{K_i})$, with b_i the orthogonal distance between the particle and the facet with index i. Next, the minimum, yet positive, time to intersection is computed as $\Delta t_{i_{\min}} = \min\{\max(0, \Delta t_i)\}$, with i the indices of the neighboring cells. If $\Delta t_{i_{\min}} > \Delta t^{(k)}$ the particle is pushed to its new position using timestep $\Delta t^{(k)}$, and the time step is terminated by setting $\Delta t^{(k+1)} = 0$. If $\Delta t_{i_{\min}} < \Delta t^{(k)}$, the particle p is pushed to the facet intersection $\mathbf{x}_p^{(k)}$ using $\Delta t_{i_{\min}}$, and the hosting cell is updated to facet $i_{\min}$, i.e. the facet with the index corresponding to $\Delta t_{i_{\min}}$. Furthermore, the time step is decremented to $\Delta t^{(k+1)} = \Delta t^{(k)} - \Delta t_{i_{\min}}$, after which the particle tracking continues until the time step for a particle has zero time remaining. Algorithm 1 contains the pseudo-code for the convex polyhedron particle tracking, using an Euler method for the particle advection.

The convex polyhedron procedure comes with a number of advantages, also pointed out in [33]. First of all, it is applicable to arbitrary polyhedral meshes, both in two and three spatial dimensions. Even though the current stable

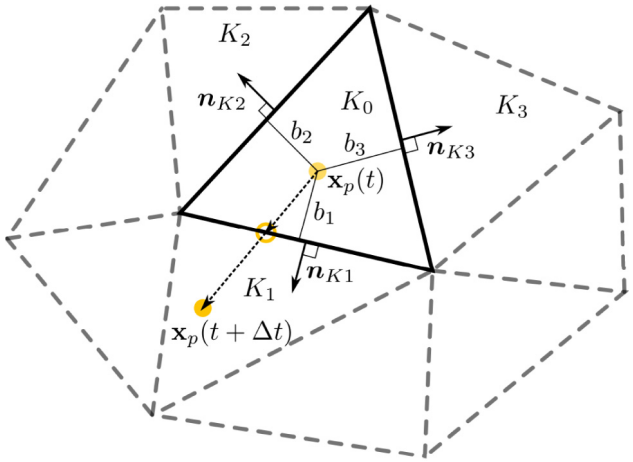


Fig. 2. Particle tracking using the convex polyhedron strategy.

release of FEniCS (2019.1.0) only supports simplicial meshes, this renders a generalization of the particle tracking scheme to other cell shapes straightforward, on the premise that cells are convex. Secondly, by marking the facets on the boundary of the domain, it is straightforward to detect if, when, and where a particle hits a specific boundary. This feature is useful when dealing with external boundaries, as well as internal boundaries, with the latter resulting from the mesh partitioning in parallel computations. Finally, the fraction of the time step spent in a certain cell is explicitly known in the convex polyhedron method. This can facilitate the updating of the particle velocity along its trajectory [34], although it is not further supported by the code in its present form.

Algorithm 1: Convex polyhedron particle tracking: pseudo-code for a single particle initially located in a cell K, using Euler forward for the particle advection.

$k \Leftarrow 0$
$\Delta t^{(k)} \Leftarrow \Delta t$
$\mathbf{x}_p^{(k)} \Leftarrow \mathbf{x}_p(t)$
Hosting cell: K
while $\Delta t^{(k)} > 0$ **do**
  $k \Leftarrow k+1$
  Time to facet intersection: $\Delta t_i = b_i/(\boldsymbol{a}\left(\mathbf{x}_p^0, t\right) \cdot \boldsymbol{n}_{K_i})$, with i the indices of the neighboring cells.
  Minimum, yet positive time: $\Delta t_{i_{\min}} = \min\{\max(0, \Delta t_i)\}$, with i the indices of the neighboring cells
  if $\Delta t_{i_{\min}} > \Delta t^{(k)}$ **then**
    Particle remains in cell K
    $\mathbf{x}_p^{(k)} \Leftarrow \mathbf{x}_p^{(k-1)} + \boldsymbol{a}\left(\mathbf{x}_p^0, t\right) \cdot \Delta t^{(k)}$
    $\Delta t^{(k)} \Leftarrow 0$
  else
    Push particle to facet
    $\mathbf{x}_p^{(k)} \Leftarrow \mathbf{x}_p^{(k-1)} + \boldsymbol{u}\left(\mathbf{x}_p^0, t\right) \cdot \Delta t_{i_{\min}}$
    $\Delta t^{(k)} = \Delta t^{(k-1)} - \Delta t_{i_{\min}}$
    if Facet has two neighboring cells **then**
      Update hosting cell index: $K \Leftarrow K_{i_{\min}}$
    else
      Apply boundary condition, see Section 3.3
    end if
  end if
end while
$\mathbf{x}_p(t+\Delta t) \leftarrow \mathbf{x}_p^{(k)}$

3.3. Boundary conditions at particle level

Apart from enforcing the boundary conditions on the background mesh, modifications at the particle level are also required when a particle hits a specific boundary. This event is detected when a particle is pushed to a facet having only one neighboring cell, see Line 20 in Algorithm 1. On the exterior boundary, the user can mark the different parts

of the boundary as either being "closed" (integer value 1), "open" (integer value 2) or "periodic" (integer value 3) via a `MeshFunction`, where this mesh function is passed to the particle advection class, see, for instance, the unit tests in `test_2d_advect.py`. Internal boundaries, i.e. facets that are on processor boundaries, are assigned an integer value of 0 by LEoPart.

When a particle crosses either an internal, a periodic or an open boundary facet during the advection, we update the list of particle indices that needs to be relocated at the end of the time step as

C++ code

```
reloc.push_back({ci->index(), i,  std::numeric_limits<unsigned int>::max()});
```

Where `ci->index()` the cell index of the hosting cell, `i` the particle index within the hosting cell, and `numeric_limits<unsigned int>::max()` the receiving cell index, with this value indicating that the particle cannot be tracked on the (partition of the) mesh.

The implementation of the different particle boundary conditions is briefly discussed below.

3.3.1. Internal boundaries and particle communication

At the end of each advection substep, LEoPart tries to relocate the particles which escaped the old host by assigning them to a receiving cell via `particles::relocate`, see Section 3.2.1. Particles that crossed a facet on an internal, an open or a periodic boundary , however, have a receiving cell index value of `numeric_limits<unsigned int>::max()`. In this case, the particle is passed to the particle collector `particles::particle_communicator_collect` that prepares the particle for communication between the processors by (I) finding the candidate host processor(s) via `BoundingBox::compute_process_collisions`, and (II) appending the particle to the buffer that will be communicated to the candidate processor(s), i.e.

Algorithm 2: Particle communication I: `particles::particle_communicator_collect`.

if `cidx_recv == numeric_limits<unsigned int>::max()` **then**
  Find candidate hosting processor(s) `procs` via `dolfin::BoundingBox::compute_process_collisions`
  Push a copy of the particle to the buffer `_comm_snd[procs]` that will be sent to the candidate processors
end if

Once all the temporary copies of the particles that need communication are collected in `_comm_snd`, and after deleting these particles from the original hosting cell via `particles::delete_particle`, the actual communication takes place via `particles::particle_communicator_push`. This communicates the particle copies to the candidate processor(s) via `MPI_Alltoallv`, and checks whether the particle can be located on the candidate processor and in which cell via `BoundingBox::compute_first_entity_collision`. If so, the communicated particle is recreated in the hosting cell on the candidate processor, if not, the other candidate processors are checked. In pseudocode, this particle relocation/communication strategy can be summarized as

Algorithm 3: Particle communication II: `particles::particle_communicator_push`.

Communicate `_comm_snd` (Algorithm 2) to the buffer `comm_rcv_vec` on candidate processors via `MPI_Alltoallv`
for all Particles `p` in `comm_rcv_vec` **do**
  if `p` collides with a cell in the candidate processor via `BoundingBox::compute_first_entity_collision` **then**
    Assign particle to receiving cell on candidate processor
  end if
end for

Two remarks are made in view of this collection and pushing algorithm:

- The communication of particles between processors is as yet done via `MPI_Alltoallv`. Although robust for large timesteps, this global communication might be somewhat inefficient since particles in general will move to processors that are close to the current processor. To exploit this, we will probably replace the communication by `MPI_Neighbor_alltoallv` in the future. This modification has, however, moderate priority since the numerical examples in Section 5 demonstrate that the particle advection usually represents a small fraction of run-time.
- The receiving cell index for a particle that escaped through a facet on an open boundary is also set to `numeric_limits<unsigned int>::max()` and hence sent to the particle collector `_comm_snd`. However, no new hosting cell is found in Algorithm 3 for such a particle, and the particle is deleted as desired by the `particles::delete_particle` that is executed in between the `particle_communicator_collect` and the `particle_communicator_push` method.

3.3.2. Periodic boundaries

When a particle crosses a facet which is marked as a periodic boundary, it should reappear at the opposing side of the domain. To implement this, facets on a periodic boundary, need to be matched against facets at the opposing side of the domain. This is taken care of in the advection class when the boundary is marked as a periodic boundary via a `MeshFunction`, and the coordinate-limits of the opposing boundaries are specified pairwise by the user. To illustrate this, a bi-periodic unit square domain is marked in LEoPart, when using the forward Euler particle advection, as

Python code

```
from dolfin import (VectorFunctionSpace, Function, RectangleMesh,
                    SubDomain, MeshFunction, Point)
from leopart import (particles, advect_particles)
import numpy as np


class Boundaries(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary


(xmin, ymin, xmax, ymax) = (0., 0, 1., 1.)

mesh = RectangleMesh(Point(xmin, ymin), Point(xmax, ymax), 10, 10)
V = VectorFunctionSpace(mesh, "CG", 1)
v = Function(V)

# Mark the boundary as periodic
facet_marker = MeshFunction("size_t", mesh, mesh.topology().dim()-1)
facet_marker.set_all(0)
boundaries = Boundaries()
boundaries.mark(facet_marker, 3)

# Specify opposing boundaries, always comes in pairs
lims = np.array([[xmin, xmin, ymin, ymax], [xmax, xmax, ymin, ymax],
                 [xmin, xmax, ymin, ymin], [xmin, xmax, ymax, ymax]])

# Locate dummy particle at (0.5, 0.5) and initialize advection class
p = particles(np.array([0.5, 0.5]), [], mesh)
ap = advect_particles(p, V, v, facet_marker, lims.flatten())
```

Full code examples of how periodic boundaries are applied in 2D and 3D are found in `TaylorGreen_2D.py` and `TaylorGreen_3D.py`.

3.3.3. Open boundaries and particle insertion/deletion

At open boundaries, particles either escape or enter the domain. When a particle escapes through an open boundary facet, it simply is deleted from the list of particles. Inflow boundaries, however, are less straightforward since new particles are to be created. This is done via the `AddDelete` class, which also allows a user to keep control over the number of particles per cell.

`AddDelete` takes as arguments the particle class, a lower and an upper bound for the number of particles per cell, and a list of FEniCS functions which are to be used for initializing the particle values. If a cell is marked as almost empty, i.e. the number of particles is lower than a preset lower bound for the number of particles per cell, the particle deficit is complemented by creating new particles. The locations for the new particles in their hosting cell are determined using a random number generator. To initialize the other particle quantities, two options are at the user's convenience: the particle value is either initialized based on a point interpolation from the underlying mesh field, or the particle value is assigned based on rounding-off the interpolated field value to a lower or upper boundary, i.e.

$$\psi_p = \begin{cases} \psi_{\min} & \text{if} \quad \psi_h(\mathbf{x}_p) \leq \frac{\psi_{\min}+\psi_{\max}}{2}, \\ \psi_{\max} & \text{if} \quad \psi_h(\mathbf{x}_p) > \frac{\psi_{\min}+\psi_{\max}}{2}. \end{cases} \tag{8}$$

This feature is particularly useful when the particles carry binary fields, such as the density in immiscible two-fluid simulations.

The minimal example below demonstrates the LEoPart implementation using the two options for particle insertion/deletion. Results are depicted in Fig. 3.

Python code

```
from dolfin import (UnitSquareMesh, FunctionSpace, Function, Expression,
                    Point, Constant)
from leopart import (particles, RandomRectangle, AddDelete,
                     assign_particle_values)
```

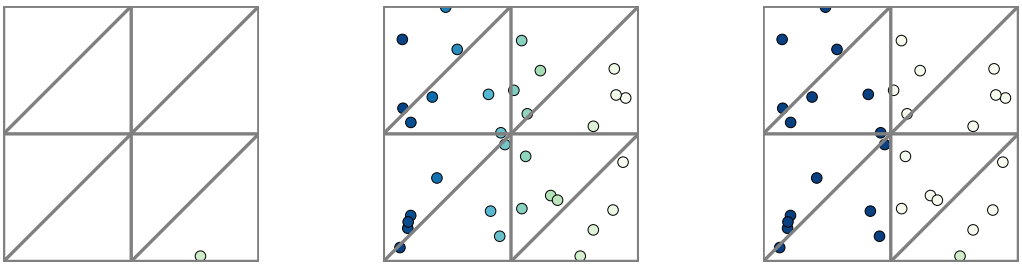


Fig. 3. Particle insertion: initial particle–field (left), particle value assignment by interpolation (middle), binary particle value assignment (right).

```
msh = UnitSquareMesh(2, 2)
Psi = FunctionSpace(msh, "DG", 1)
psi = Function(Psi)

psi_expression = Expression("x[0]", degree=1)
psi.assign(psi_expression)

(p_min, p_max) = (4, 8)
(psi_min, psi_max) = (0., 1.)

# Initialize particles
xp = RandomRectangle(Point(0., 0.), Point(1., 1.)).generate([1, 1])
psip_smooth = assign_particle_values(xp, psi_expression)
psip_binary = assign_particle_values(xp, psi_expression)

p = particles(xp, [psip_smooth, psip_binary], msh)

# Slot 2, psip_binary initialized using binary values
AD = AddDelete(p, p_min, p_max, [psi, psi], [2], [psi_min, psi_max])
# Sweep over mesh to delete/insert particles
AD.do_sweep()

(xp_n, psip_smooth, psip_binary) = (p.return_property(msh, 0),
                                    p.return_property(msh, 1),
                                    p.return_property(msh, 2))
```

The `AddDelete` class can also be used for keeping control over the maximum number of particles per cell by specifying the variable `p_max` in the above presented code. If a cell in the `do_sweep` method is marked to contain more particles than prescribed, the surplus of, say, m particles is removed by deleting m particles with the shortest distance to another particle in that cell. This procedure ensures that particles are removed evenly from the cell interior.

As a final remark: an upwind initialization of the particle value, i.e. initializing the particle value near open boundaries based on the value at the (inflow) boundary facet, is expected to be a useful feature not yet included in LEoPart.

3.3.4. Closed boundaries

When a particle hits a closed boundary during the time step, the particle is reflected by setting the particle velocity to the reflected value, i.e.

$$\mathbf{a}_h\left(\mathbf{x}_p, t\right) = \mathbf{a}_h\left(\mathbf{x}_p, t\right) - 2\left(\mathbf{a}_h\left(\mathbf{x}_p, t\right) \cdot \boldsymbol{n}\right)\boldsymbol{n}, \tag{9}$$

in which $\boldsymbol{n}$ the outwardly directed unit normal vector to a boundary facet.

4. Particle–mesh interaction

Obtaining mesh fields from the scattered particle data and updating the particle values from a known mesh field is essential to active tracer problems. These particle–mesh interaction steps go by various names in the literature such as: 'gather–scatter' steps [31,35], 'forward interpolation - backward estimation' [36] or 'particle weighting' [37].

In line with our earlier work on particle–mesh schemes [17–19], the data transfer operators are consistently coined 'particle–mesh projection' for the data transfer from the set of scattered particles to the mesh, whereas the opposite route is indicated by 'mesh–particle projection'. This convention reflects that the data transfer operators are perceived as projections between different spaces. More precisely, information needs to be projected from a *particle space* onto a *mesh space* and *vice versa*. Adopting this point of view, it readily follows that the data transfer operations are auxiliary steps, which should not deteriorate accuracy, violate consistency, nor compromise on conservation.

To comply with these requirements, LEoPart adopts a variational approach to formulate the particle–mesh and the mesh–particle projections. An ℓ^2 objective function forms the starting point for deriving the mutual particle–mesh

interactions. For a scalar-valued mesh field ψ_h and a scalar-valued particle–field ψ_p, this objective function reads

$$\min J := \sum_{p \in \mathcal{S}_t} \frac{1}{2} \left(\psi_h(\mathbf{x}_p(t), t) - \psi_p(t) \right)^2 , \tag{10}$$

where it remains to specify the minimizer, other than to say that either ψ_h or ψ_p is used to fit this purpose. The implementation of the projection strategies which can be formulated based on Eq. (10) are further highlighted for a scalar quantity ψ in the remainder of this section, and the projections for a vector-valued quantity follow the same path. More specifically, Section 4.1 discusses the various particle–mesh projections available in LEoPart, and in Section 4.2 the available mesh–particle projections are discussed. Throughout, the notation $\mathcal{T} := \{K\}$ is used to indicate the set of disjoint cells K that constitutes a meshing of the domain Ω, and each cell K has a boundary ∂K.

4.1. Particle–mesh projections

Common to the available particle–mesh projections in LEoPart is the minimization of the objective function Eq. (10) with respect to an unknown mesh field ψ_h given a known particle–field ψ_p. This requires the definition of the function space in which ψ_h is approximated. To this end, LEoPart conveniently exploits existing FEniCS tools for defining arbitrary order polynomial function spaces. For reasons that become clear shortly, LEoPart is tailored for projecting the particle data onto discontinuous function spaces at the mesh. In the scalar-valued setting these function spaces are defined by

$$W_h := \left\{ w_h \in L^2(\mathcal{T}),\ \ w_h|_K \in P_k(K)\ \forall\ K \in \mathcal{T} \right\}, \tag{11}$$

where $\mathcal{T}$ is the partitioning of the domain Ω into a set of cells K, and $P_k(K)$ denotes the space spanned by Lagrange polynomials on K, where the subscript $k \geq 0$ indicates the polynomial order.

4.1.1. ℓ^2-Projection

With these definitions, the most elementary particle–mesh projection is found by minimizing Eq. (10) for $\psi_h \in W_h$, which results in the ℓ^2-projection: given the particle values $\psi_p^n \in \Psi_t$ and particle positions $\mathbf{x}_p \in \mathcal{X}_t$, find $\psi_h \in W_h$ such that

$$\sum_{p \in \mathcal{S}_t} \left(\psi_h(\mathbf{x}_p(t), t) - \psi_p(t) \right) w_h(\mathbf{x}_p(t)) = 0 \qquad \forall\ w_h \in W_h. \tag{12}$$

Given the definition for W_h in Eq. (11), $\psi_h, w_h \in W_h$ are discontinuous across cell boundaries. Hence, Eq. (12) is solved in a cellwise fashion, i.e.

$$\sum_K \sum_{p \in \mathcal{S}_t^K} \left(\psi_h(\mathbf{x}_p(t), t) - \psi_p(t) \right) w_h(\mathbf{x}_p(t)) = 0 \qquad \forall\ w_h \in W_h, \tag{13}$$

which only requires the inversion of small, local matrices, thus admitting an efficient, parallel implementation.

The particle–mesh projection via the ℓ^2-projection is implemented in LEoPart in the `l2projection` class, which is instantiated as

C++ code

```
l2projection(particles& P, FunctionSpace& V, const std::size_t idx);
```

in which the integer index `idx` indicates which particle property is projected. Projection onto a discontinuous space as in Eq. (13), is done with the `project` method, which on the Python side can be invoked as

Python code

```
# FEniCS discrete function space
k = 2
Wh = FunctionSpace(msh, "DG", k)
psi = Function(Wh)
...
# LEoPart particles and l2 projection
p = particles(xp, [psi_p], msh)
lstsq_rho = l2projection(p, Wh, 1)
lstsq_rho.project(psi)
```

LEoPart also allows projection of particle data onto a continuous Galerkin space - which leads to a global system for Eq. (12) - by means of the `project_cg` method

Python code

```
Wh_CG = FunctionSpace(msh, "CG", k)
psi_CG = Function(Wh_CG)
...
lstsq_rho = l2projection(p, Wh_CG, 1)
lstsq_rho.project(psi_CG)
```

4.1.2. Bounded ℓ^2-projection

The minimization problem, Eq. (12), can be interpreted as a quadratic programming problem. This class of problems has been thoroughly analyzed in literature, and well-known techniques exist to extend these problems with equality, inequality, and box constraints, see e.g. [38] and references. In the context of the particle–mesh projection, imposing box constraints of the form

$$l \leq \psi_h \leq u, \tag{14}$$

can be particularly useful to ensure that the mesh field is bounded by $[l, u]$.

In LEoPART, the box-constrained ℓ^2-projection is implemented via a specialization of the `project` method, which can be invoked as

Python code

```
(lb, ub) = (0., 1.)
lstsq_rho = l2projection(p, Wh, 1)
lstsq_rho.project(psi, lb, ub)
```

with `lb` and `ub` the user-specified lower- and upper bound, respectively. At the backend, LEoPART uses `QuadProg++`[2] for solving the box-constrained optimalization problem. The bounded ℓ^2-projection is only available when projecting onto discontinuous function spaces.

4.1.3. PDE-constrained particle–mesh interaction

The motivation for introducing Lagrangian particles - particularly when used as active tracers - is to conveniently accommodate advection. The particle–mesh projections presented in the preceding two sections, however, do not possess conservation properties. That is, initializing a particle quantity from an initial mesh field, advecting the particles, and subsequently reconstructing a mesh field from the updated particle positions with the (box-constrained) ℓ^2-projection, results in a reconstructed mesh field with different integral properties. One way to conserve the mesh properties over the sequence of particle steps, is to keep track of the integral quantities on the mesh. This is accomplished by constraining the objective function for the particle–mesh projection, Eq. (10), such that the reconstructed field ψ_h satisfies a hyperbolic conservation law. The resulting PDE-constrained particle–mesh projection, developed in [18], possesses local (i.e. cellwise) and global conservation properties, and essentially involves solving the minimization problem: given ψ_p, find $\psi_h \in W_h$

$$\min_{\psi_h \in W_h} J = \sum_p \frac{1}{2} \left(\psi_h(\mathbf{x}_p) - \psi_p\right)^2 \tag{15a}$$

such that:

$$\frac{\partial \psi_h}{\partial t} + \nabla \cdot (\mathbf{a}\psi_h) = 0 \tag{15b}$$

$$+ \text{BC's} \tag{15c}$$

is satisfied in a weak sense. For brevity, only periodic boundaries or boundaries with vanishing normal velocity (i.e. $\mathbf{a} \cdot \boldsymbol{n} = 0$) are considered in this paper. For a more elaborate discussion on other boundary conditions in Eq. (15), reference is made to [18].

By casting the strong form of the constraint into a weak form by multiplying Eq. (15b) with a Lagrange multiplier field $\lambda_h \in T_h$ - with T_h defined in Eq. (A.2) in the Appendix - and after applying integration by parts, the PDE-constrained optimization problem amounts to finding the stationary points of the Lagrangian functional

$$\begin{aligned}\mathcal{L}(\psi_h, \bar{\psi}_h, \lambda_h) = \sum_p \frac{1}{2} \left(\psi_h(\mathbf{x}_p(t), t) - \psi_p(t) \right)^2 + \sum_K \oint_{\partial K} \frac{1}{2} \beta \left(\bar{\psi}_h - \psi_h\right)^2 \mathrm{d}\Gamma + \sum_K \int_K \frac{1}{2} \zeta \|\nabla \psi_h\|^2 \mathrm{d}\Omega \\ + \int_\Omega \frac{\partial \psi_h}{\partial t} \lambda_h \mathrm{d}\Omega - \sum_K \int_K \mathbf{a}\psi_h \cdot \nabla \lambda_h \mathrm{d}\Omega + \sum_K \oint_{\partial K} \mathbf{a} \cdot \mathbf{n} \bar{\psi}_h \lambda_h \mathrm{d}\Gamma,\end{aligned} \tag{16}$$

in which the first term at the right-hand side is similar to the objective function in Eq. (15a). The second line in Eq. (16) is a weak statement of the constraint equation, Eq. (15b). Furthermore, $\beta > 0$ is a small penalty parameter introduced to establish a coupling between ψ_h, and the control variable $\bar{\psi}_h$, where this control variable is defined on the facets of the cell via the trace space $\bar{W}_h$ from Eq. (A.3), analogous to the flux variable in HDG methods, see, e.g., [27,39–41]. Finally, ζ is a parameter which penalizes gradients, where this parameter is set to zero for smooth problems, and is only invoked when steep gradients in the mesh solution are to be expected, see Section 5.2.

A more in-depth interpretation of Eq. (16) and analysis of the optimality system resulting after taking variations with respect to $\left(\psi_h, \lambda_h, \bar{\psi}_h\right) \in \left(W_h, T_h, \bar{W}_h\right)$, can be found in [18]. The Appendix provides a summary of the resulting variational forms in the fully-discrete setting, yielding a 3×3 block system at the element level, see Eq. (17).

[2] https://github.com/liuq/QuadProgpp.

LEoPart implements the PDE-constrained particle–mesh projection via the `PDEStaticCondensation` class. The weak forms provided by `FormsPDEMap.py` are used to instantiate this class. Using notations similar to Eq. (17), a Python implementation may read

Python code

```
# FEniCS
W = FunctionSpace(mesh, "DG", k)
T = FunctionSpace(mesh, "DG", 0)
Wbar = FunctionSpace(mesh, "DGT", k)

psi_h, psi0_h = Function(W), Function(W)
lambda_h = Function(T)
psibar_h = Function(Wbar)

bc = DirichletBC(Wbar, Constant(0.), "on_boundary")
...
# LEoPart
FuncSpace_adv = {"FuncSpace_local": W, "FuncSpace_lambda": T, "FuncSpace_bar": Wbar}
forms_pde = FormsPDEMap(mesh, FuncSpace_adv).forms_theta_linear(psi0_h, a_advection,
                                dt, Constant(1.0))
pde_projection = PDEStaticCondensation(mesh, p,
                     forms_pde["N_a"], forms_pde["G_a"], forms_pde["L_a"],
                     forms_pde["H_a"],
                     forms_pde["B_a"],
                     forms_pde["Q_a"], forms_pde["R_a"], forms_pde["S_a"],
                     [bc], property_idx)
```

Assembly of the matrices and vectors is done via the `assemble` method

Python code

```
pde_projection.assemble()
```

which serves a two-fold purpose: first of all it computes the element contributions for each cell K in the 3×3 block matrix

$$\begin{bmatrix} \boldsymbol{M}_p + \boldsymbol{N} & \boldsymbol{G}(\theta) & \boldsymbol{L} \\ \boldsymbol{G}(\theta)^\top & \boldsymbol{0} & \boldsymbol{H} \\ \boldsymbol{L}^\top & \boldsymbol{H}^\top & \boldsymbol{B} \end{bmatrix} \begin{bmatrix} \boldsymbol{\psi}^{n+1} \\ \boldsymbol{\lambda}^{n+1} \\ \bar{\boldsymbol{\psi}}^{n+1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\chi}_p \boldsymbol{\psi}_p^n \\ \boldsymbol{G}(1-\theta)^\top \boldsymbol{\psi}^n \\ \boldsymbol{0} \end{bmatrix}, \tag{17}$$

where the different contributions readily follow from Eq. (A.4).

Secondly, the `assemble` method assembles the local contributions into a global matrix–vector system. Since ψ_h and λ_h are local to a cell, the resulting global system of equations is expressed in terms of the flux variable $\bar{\psi}_h$ only. That is, the global system is assembled as follows

$$\bigwedge_K \left(\mathbf{B} - \begin{bmatrix} \mathbf{L} \\ \mathbf{H} \end{bmatrix}^\top \begin{bmatrix} \mathbf{M}_p + \mathbf{N} & \mathbf{G}(\theta) \\ \mathbf{G}(\theta)^\top & \mathbf{0} \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{L} \\ \mathbf{H} \end{bmatrix} \right) \bar{\boldsymbol{\psi}}^{n+1} = - \bigwedge_K \begin{bmatrix} \mathbf{L} \\ \mathbf{H} \end{bmatrix}^\top \begin{bmatrix} \mathbf{M}_p + \mathbf{N} & \mathbf{G}(\theta) \\ \mathbf{G}^\top(\theta) & \mathbf{0} \end{bmatrix}^{-1} \begin{bmatrix} \boldsymbol{\chi}_p \boldsymbol{\psi}_p^n \\ \boldsymbol{G}(1-\theta)^\top \boldsymbol{\psi}^{*,n} \end{bmatrix}, \tag{18}$$

in which the wedge $\bigwedge$ denotes assembly of the cell contributions into the global matrix, where this requires the inversion of a small saddle-point problem for each cell K independently, so that the assembly procedure is amenable to a fast parallel implementation.

The method `solve_problem`

Python code

```
pde_projection.solve_problem(psibar_h, psi_h, solver="none", preconditioner="default")
```

solves the resulting global system, Eq. (18), for $\bar{\boldsymbol{\psi}}^{n+1}$. The `solver` and `preconditioner` can be specified by the user and defaults to the MUltifrontal Massively Parallel sparse direct Solver (MUMPS). In addition to solving the global problem, the `solve_problem` method also applies the back substitution

$$\begin{bmatrix} \boldsymbol{\psi}^{n+1} \\ \boldsymbol{\lambda}^{n+1} \end{bmatrix} = \begin{bmatrix} \mathbf{M}_p + \mathbf{N} & \mathbf{G}(\theta) \\ \mathbf{G}^\top(\theta) & \mathbf{0} \end{bmatrix}^{-1} \left(\begin{bmatrix} \boldsymbol{\chi}_p \boldsymbol{\psi}_p^{n+1} \\ \boldsymbol{G}(1-\theta)^\top \boldsymbol{\psi}^{*,n} \end{bmatrix} - \begin{bmatrix} \mathbf{L} \\ \mathbf{H} \end{bmatrix} \bar{\boldsymbol{\psi}}^{n+1} \right), \tag{19}$$

for obtaining the local unknowns $\boldsymbol{\psi}^{n+1}$ and (optionally) the Lagrange multiplier unknowns $\boldsymbol{\lambda}^{n+1}$.

The sequence of steps for instantiating, assembling and solving the PDE-constrained particle mesh-projection with LEoPart can be summarized in the algorithm:

Algorithm 4: PDE-constrained projection algorithm.

Instantiate `PDEStaticCondensation`
Assemble with `PDEStaticCondensation::assemble`:
  Global matrix $\mathbf{A}_g = 0$
  Global vector $\mathbf{f}_g = 0$
  for all cells K in mesh **do**
    Assemble local contributions $\mathbf{N}, \mathbf{G}, \mathbf{L}, \mathbf{H}, \mathbf{B}$ from Eq. (17) with `dolfin::LocalAssembler`
    Assemble particle contributions $\mathbf{M}_p, \boldsymbol{\chi}_p$ with `particles::get_particle_contributions`
    Use `Eigen::inverse` to compute

$$\begin{bmatrix} \mathbf{M}_p + \mathbf{N} & \mathbf{G}(\theta) \\ \mathbf{G}^\top(\theta) & \mathbf{0} \end{bmatrix}^{-1}$$

    Add local contributions to global matrix: $\mathbf{A}_g \overset{+}{=}$ LHS(Eq. (18))
    Add local contribution to global vector $\mathbf{f}_g \overset{+}{=}$ RHS(Eq. (18))
  end for
Solve using `PDEStaticCondensation::solve`:
  $\bar{\boldsymbol{\psi}}^{n+1} \Leftarrow \mathbf{A}_g^{-1}\mathbf{f}_g$
  for all cells in mesh **do**
    Compute $\boldsymbol{\psi}^{n+1}, \boldsymbol{\lambda}^{n+1}$ by backsubstitution into Eq. (19).
  end for

4.2. Mesh–particle projection

The mesh–particle projections, for updating particle properties from a given mesh field, also take the objective functional Eq. (10) as their starting point. Contrary to the particle–mesh projections, the particle–field is the unknown, so that the objective function needs minimization with respect to the particle–field ψ_p, for the projection of a scalar-valued quantity. Performing the minimization results in: given $\psi_h \in W_h$, find $\psi_p \in \Psi_t$ such that

$$\sum_{p \in \mathcal{S}_t} \left(\psi_h\left(\mathbf{x}_p(t), t\right) - \psi_p(t)\right)\, \delta\psi_p = 0 \qquad \forall\, p \in \mathcal{S}_t. \tag{20}$$

Since this equation must hold for arbitrary variations $\delta\psi$, the particularly simple result for the mesh–particle projection becomes

$$\psi_p(t) = \psi_h(\mathbf{x}_p(t), t) \qquad \forall\, p \in \mathcal{S}_t, \tag{21}$$

i.e. particles values are obtained via interpolation of the mesh field. Interpolating a mesh field to particles is done in LEoPart via the `interpolate` method in the `particle` class, i.e.

Python code

```
p = particles(xp, [psi_p], mesh)
p.interpolate(psi_h, 1)
```

An interpolation overwrites the particle quantities with the interpolated mesh values. However, one of the assets of combining a high resolution particle–field with a comparatively low-resolution mesh field is that the particle–field may provide sub-grid information to the mesh [34,42,43]. In order to take advantage of this, the particles need to have a certain degree of independence from the mesh field. Analogous to the FLIP method [44], this is achieved by *updating* the particle quantities by projecting the *change* in the mesh field, rather than *overwriting* particle quantities. For a scalar valued quantity, this incremental update reads

$$\dot{\psi}_p = \dot{\psi}_h(\mathbf{x}_p) \qquad \forall\, p \in \mathcal{S}_t, \tag{22}$$

in which $\dot{\psi}_h \in W_h$ the time derivative of the mesh field.

A fully-discrete counterpart of Eq. (22) is implemented in LEoPart using the θ method for the time discretization:

$$\psi_p^{n+1} = \psi_p^n + \Delta t\left((1-\theta)\dot{\psi}_h^n\left(\mathbf{x}_p^n\right) + \theta\dot{\psi}_h^{n+1}\left(\mathbf{x}_p^{n+1}\right)\right) \qquad \forall\, p \in \mathcal{S}_t, \tag{23}$$

where Δt the time step, $0 \leq \theta \leq 1$, and $\dot{\psi}_h^n \in W_h$ is defined as $\dot{\psi}_h^n = (\psi_h^n - \psi_h^{n-1})/\Delta t$. Eq. (23) is available in LEoPart via the `increment` method in the `particles` class, and can be used as

Python code

```
# Particle
p = particles(xp, [psi_p, dpsi_p_dt], msh)

# Incremental update with theta method
theta = 0.5
...
step = 2
p.increment(psih_new, psih_old, [1, 2], theta, step)
```

Two closing remarks are made in view of this incremental update:

- For step 1, $\theta = 1$ since $\dot{\psi}_h^0$ is usually not defined.
- The increment from the old time level, i.e. $\dot{\psi}_h^n\left(\mathbf{x}_p^n\right)$ is stored at the particle level between consecutive time steps, for efficiency reasons. This requires an additional slot on the particles, i.e. `dpsi_p_dt`. The integer array in the `increment` call indicates which particle slots are used for the incremental update, i.e. `p.increment(psih_new, psih_old, [1, 2], theta, step)`.

5. Example applications

This section demonstrates the performance of LEoPart in terms of accuracy, conservation and computational run time for a number of advection-dominated problems. On a per-time-step basis, the particle–mesh approach typically comprises the following sequence of steps:

1. Lagrangian advection of the particles, as outlined in Section 3.2.
2. particle–mesh projection to project the scattered particle data onto an Eulerian mesh field using either the ℓ^2-projection, discussed in Section 4.1.1, the bound constrained ℓ^2-projection from Section 4.1.2, or the PDE-constrained projection, Section 4.1.3.
3. (optional) solve the physical problem - e.g. a diffusion or Stokes problem - at the mesh, using the reconstructed mesh field either as a source term or as an intermediate solution to the discrete equations.
4. (optional) update the particles given the solution at the mesh, using the mesh–particle interaction tools from Section 4.2.

Step 1 and 2 are sufficient to solve an advection problem at the particles and to test the reconstruction of mesh fields from the moving particles. The sequence of steps 1–4 can be used for active tracer modeling or a particle–mesh operator splitting for, e.g., the advection–diffusion equation or the incompressible Navier–Stokes equations, see also Maljaars [17,18]. For all the examples presented below, reference is made to the corresponding computer code in the LEoPart repository on Bitbucket.

5.1. Translation of a periodic pulse

As a straightforward, yet illustrative example, the translation of the sinusoidal profile

$$\psi(\mathbf{x}, 0) = \sin 2\pi x \sin 2\pi y \tag{24}$$

on the bi-periodic unit square is considered, in analogy to LeVeque [45]. Owing to its simplicity, this test allows to assess the accuracy and the convergence properties of the ℓ^2- and the PDE-constrained particle–mesh projection. Furthermore, it is used to illustrate the performance of the scheme by means of a strong-scaling study. Test results can be reproduced by running `SineHump_convergence.py` for the convergence study, and `SineHump_hires.py` for the scaling study.

In this example, the advective velocity field $\mathbf{a} = [1, 1]^\top$ is used, so that at $t = 1$ the initial data should be recovered.

To investigate convergence, we consider a range of triangular meshes obtained by splitting a regular $n \times n$ Cartesian mesh into $2n^2$ triangles. We construct 5 different meshes with $n = (11, 22, 44, 88, 176)$, respectively. Different polynomial orders $k = 1, 2, 3$ are used for the discontinuous function space W_h, Eq. (11), onto which the particle data is projected. For the PDE-constrained projection, the polynomial order for the Lagrange multiplier space T_h, Eq. (A.2), is $l = 0$ in all cases. Particles are seeded in a regular lattice on the mesh, such that each cell contains approximately 15 particles, independent of the mesh resolution, see Fig. 4 as an example. An Euler scheme suffices for exact particle advection, and the time step corresponds to a CFL-number of approximately 1. Furthermore, in the PDE-constrained particle–mesh projection, the β-parameter is set to 1e-6, and ζ is set to 0. All computations use a direct sparse solver (`SuperLU`) to solve the global system of equations. Also, note that for this advection problem, the scalar valued property ψ_p, attached to each particle p, needs no updating and stays constant throughout the computation.

The accuracy of the method is assessed at $t = 1$, using both the ℓ^2-particle–mesh projection from Section 4.1.1, and the PDE-constrained projection from Section 4.1.3 upon refining the mesh and the time step, see Table 1. Optimal convergence rates of order $k + 1$ are observed for both projections strategies, thus highlighting the accuracy of the particle–mesh projections in conjunction with particle advection.

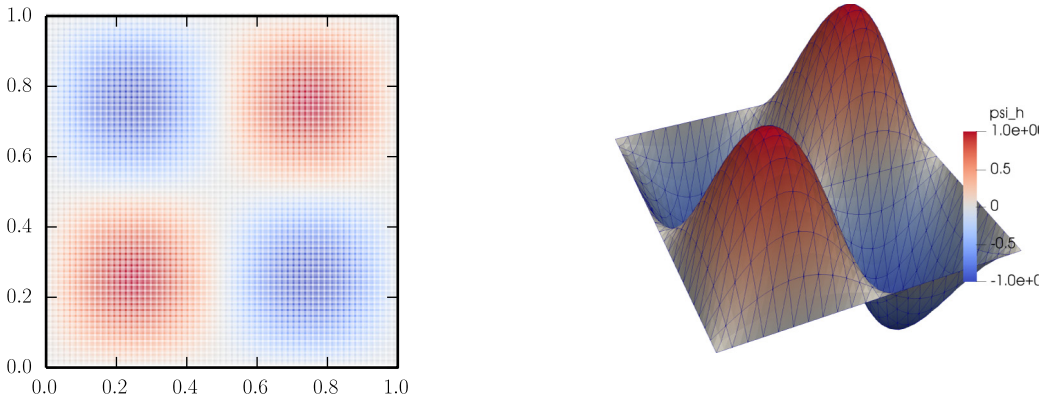


Fig. 4. Sinusoidal Pulse: particle–field (left) and the reconstructed solution at the mesh (right) using the PDE-constrained projection with polynomial order $k = 2$ and a mesh containing 968 cells.

Table 1
Translating pulse: overview of model runs with the associated L^2-error $\|\psi - \psi_h\|$, and convergence rate at time $t = 1$ for different polynomial orders k. For the PDE-constrained projection, the polynomial order in the Lagrange multiplier space is $l = 0$ in all cases.

Projection	Δt	Cells	Parts.	$k = 1$		$k = 2$		$k = 3$	
				Error	Rate	Error	Rate	Error	Rate
ℓ^2	1e−1	242	3 984	3.3e−2	–	1.7e−3	–	9.4e−5	–
	5e−2	968	14 542	8.3e−3	2.0	2.1e−4	3.0	5.9e−6	4.0
	2.5e−2	3 872	57 663	2.1e−3	2.0	2.7e−5	3.0	3.7e−7	4.0
	1.25e−2	15 488	230 428	5.2e−4	2.0	3.3e−6	3.0	2.3e−8	4.0
	6.25e−2	61 952	921 837	1.3e−4	2.0	4.1e−7	3.0	1.4e−9	4.0
PDE	1e−1	242	3 984	3.3e−2	–	1.7e−3	–	9.4e−5	–
	5e−2	968	14 542	8.3e−3	2.0	2.1e−4	3.0	5.9e−6	4.0
	2.5e−2	3 872	57 663	2.1e−3	2.0	2.7e−5	3.0	3.7e−7	4.0
	1.25e−2	15 488	230 428	5.2e−4	2.0	3.3e−6	3.0	2.3e−8	4.0
	6.25e−2	61 952	921 837	1.3e−4	2.0	4.1e−7	3.0	1.4e−9	4.0

Table 1 shows that the error levels for the ℓ^2- and the PDE-constrained particle–mesh are similar. The difference between these projections becomes clear, however, by investigating the mass error

$$\epsilon_{\Delta\psi_\Omega} = |\int_\Omega (\psi_h(\mathbf{x}, T) - \psi_h(\mathbf{x}, 0))\, \mathrm{d}\Omega|, \tag{25}$$

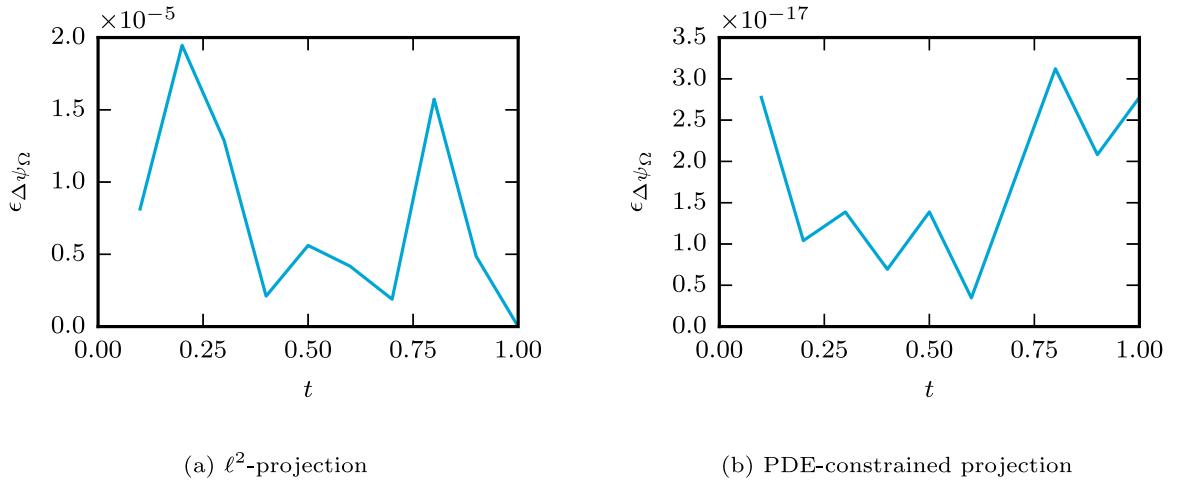
which is plotted as a function of time for the ℓ^2-projection in Fig. 5a, and for the PDE-constrained projection in Fig. 5b. Evident from these figures is that the ℓ^2-projection yields a nonzero mass error, whereas for the PDE-constrained projection the mass error is zero to machine precision.

The trade-off between the non-conservative ℓ^2-projection and the conservative PDE-constrained is elucidated by investigating the computational times. Wallclock times for the high-resolution case - polynomial order $k = 3$ with 61,952 cells, 921,837 particles and 160 time steps - run on different number of Intel Xeon CPU E5-2690 v4 processors are presented in Fig. 6. Solving the global system for the PDE-constrained projection using a direct solver is computationally much more demanding compared to the (local) ℓ^2-projection. This illustrates the need for an efficient iterative solver for the PDE-constrained projection step.

Table 2 further investigates the scaling of the different components by summarizing the speed-up for the different tests relative to the run on one processor. The particle advection and the assembly step - with the latter only relevant for the PDE-constrained projection - exhibit excellent scaling properties, which is explained by the locality of these operations, i.e. these steps are performed cellwise. This also holds true for the ℓ^2-projection, which amounts to a cellwise projection of the particle properties onto a discontinuous function space, see Section 4.1.1. Clearly, the direct sparse solver for the PDE-constrained projection does not possess optimal scaling properties, thus limiting the scalability of the constrained projection step.

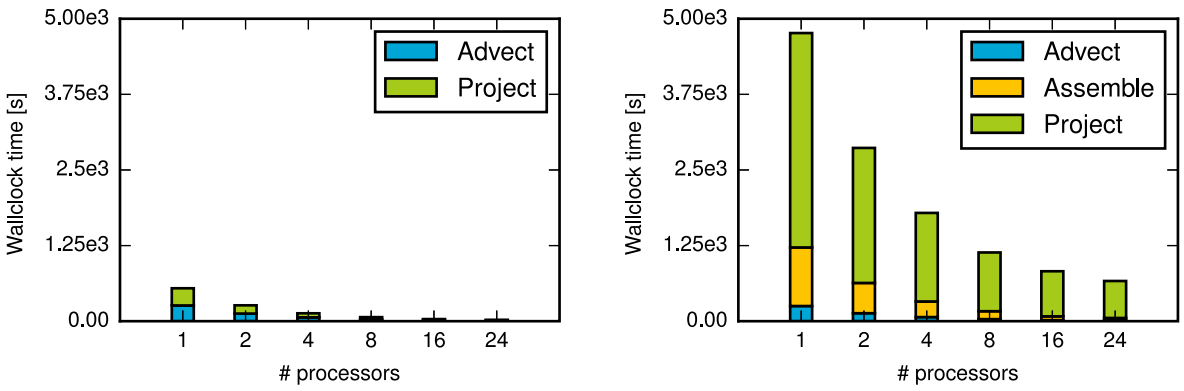
5.2. Slotted disk

Combining particle-based and mesh-based techniques appears particularly promising for applications in which sharp flow features are to be preserved. The solid body rotation of a slotted disk after Zalesak [46] is a prototypical example of



(a) ℓ^2-projection (b) PDE-constrained projection

Fig. 5. Translating pulse: mass error over time for different particle–mesh projections.



(a) ℓ^2-projection (b) PDE-constrained projection

Fig. 6. Translating sinusoidal hump: strong scaling study.

Table 2
Translating sinusoidal hump: speed-up of the different model parts in parallel computations benchmarked against 1 processor run.

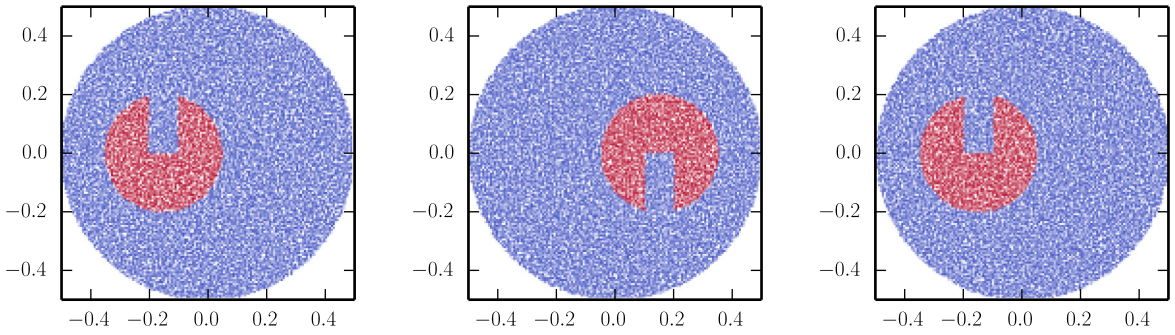
# Processors	ℓ^2-projection						PDE-constrained projection					
	1	2	4	8	16	24	1	2	4	8	16	24
Advect particles	1	2.1	4.1	8.0	15.1	22.4	1	1.9	3.7	7.7	14.8	21.9
Assembling	–	–	–	–	–	–	1	1.9	3.8	7.4	15.4	23.2
Solve projection	1	2.1	4.3	8.2	16.1	25.1	1	1.6	2.4	3.6	4.8	5.8
Total	1	2.1	4.2	8.1	15.6	23.8	1.	1.7	2.7	4.2	5.8	7.2

such problems, and often serves as a benchmark for interface tracking schemes, see [47,48], among many others. We now use this test to demonstrate the various tools that LEoPart offers for tracking sharp discontinuities in material properties, such as a density jump in immiscible multi-fluid flows.

The problem set-up is as follows. A disk with radius 0.2 - from which a slot with a width of 0.1 and depth 0.2 is cut out - is initially centered at $(x, y) = (-0.15, 0)$ on the domain of interest $\Omega := \{(x, y) | x^2 + y^2 \leq 0.5\}$. This domain is triangulated into 14,464 cells on which 438,495 particles are seeded. The advective velocity field is given by

$$\mathbf{a} = \pi \, (-y, x)^\top . \tag{26}$$

The time step is set to 0.02, so that one full rotation is performed in 100 steps. The three-stage Runge–Kutta scheme, available via the `advect_rk3` class, is used for the particle advection.



(a) $t = 0$ (b) $t = 1$ (c) $t = 2$

Fig. 7. Slotted disk: particle–field (case independent) at different time instants. Plot shows every 25th particle for clarity, and color values range between 0 (blue) and 1 (red).

Table 3
Slotted disk: overview of test cases, with k the polynomial order in the function space W_h (Eq (A.1)) and l the polynomial order for the Lagrange multiplier space T_h (Eq. (A.2)) in the PDE-constrained projection.

	particle–mesh projection	k	l	ζ
Case 1	Bounded ℓ^2	1	–	–
Case 2	PDE-constrained	1	0	0
Case 3	PDE-constrained	1	0	30

Table 4
Slotted Disk: runtime, area error $\epsilon_{\Delta\psi_\Omega}$ at $t = 2$ and minimum and maximum values for ψ_h for a bounded ℓ^2-projection (Case 1), or a PDE-constrained projection with $\zeta = 0$ (Case 2) and $\zeta = 30$ (Case 3).

	Cells	Particles	Wallclock time (s)	$\epsilon_{\Delta\psi_\Omega}(t = 2)$	$\psi_{h,\min}(x, t)$	$\psi_{h,\max}(x, t)$
Case 1	14,464	438,495	150	1.4e−5	−2.6e−16	1.00
Case 2	14,464	438,495	182	2.0e−15	−11.2	8.04
Case 3	14,464	438,495	180	1.3e−15	−0.01	1.02

Different particle–mesh projection strategies that are available in LEoPart are investigated in this example, see Table 3. Note that Case 2 and Case 3 only differ in terms of the ζ parameter, see Eq. (16). The computer code in `SlottedDisk_rotation_l2.py` reproduces Case 1, the computer code for reproducing Case 2 and Case 3 is found in `SlottedDisk_rotation_PDE.py`. The test is run using 8 Intel Xeon CPU E5-2690 v4 processors.

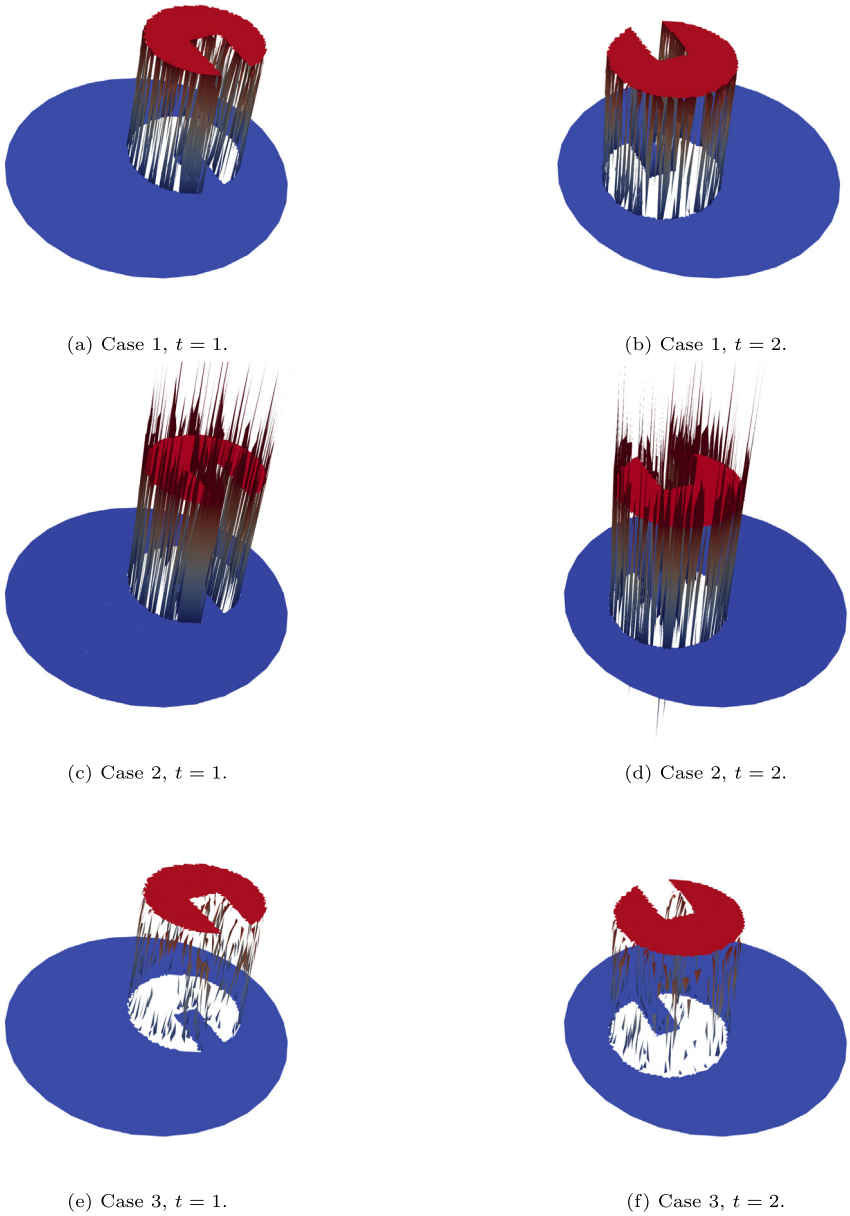
As for the previous example, there is no updating of the scalar valued property attached to a particle for this advection problem. Hence, sharp features at the particle level pertain, and the particle advection part for all the different cases is equal. Fig. 7 shows the particle–field at $t = 0$ (initial field), $t = 1$ (half rotation) and after a full rotation at $t = 2$.

Fig. 8 compares the reconstructed mesh fields for the three different cases at time instants $t = 1$ and $t = 2$. For the bounded ℓ^2-projection (Case 1), the discontinuity in the particle–field is captured at the mesh without under- or overshoot, and values stay within the prescribed bounds $0 \leq \psi_h \leq 1$ to machine precision, see Table 4. Another advantage of this approach is that it is fast, and easy to implement in parallel. However, as reported in Table 4, the mass error for the bounded ℓ^2-projection is non-zero. The latter issue is overcome by using the conservative PDE-constrained particle–mesh projection for the reconstruction of the mesh fields, Case 2 and Case 3. However, for Case 2 - in which $\zeta = 0$ - localized over- and undershoot is observed near the discontinuities. As argued in [18], this artifact is a resolution issue with the mesh being too coarse to capture the sharp discontinuity at the particle level monotonically.

By setting ζ to a value of 30, i.e. approximately equal to the number of particles per cell, this issue is effectively mitigated, see also the minimum and the maximum values for ψ_h over the entire computation reported in Table 4. This table also confirms global mass conservation for Case 2 and Case 3, local conservation properties for similar examples were demonstrated in [18,19] and not further shown here.

5.3. Lock exchange test

As an example which is geared towards practical multi-fluid applications, the lock-exchange test is considered. Driven by gravity, a dense fluid current moves underneath a lighter fluid, where a thin vertical membrane initially separates the two fluids. Using LEoPart, density tracking and momentum advection is done using Lagrangian particles, and the



(a) Case 1, $t = 1$.

(b) Case 1, $t = 2$.

(c) Case 2, $t = 1$.

(d) Case 2, $t = 2$.

(e) Case 3, $t = 1$.

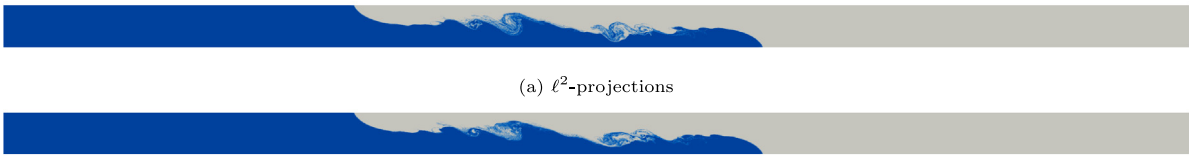
(f) Case 3, $t = 2$.

Fig. 8. Slotted Disk: reconstructed mesh field ψ_h at different time instants using a particle–mesh projection based on a bounded ℓ^2-projection (Case 1), or a PDE-constrained projection with $\zeta = 0$ (Case 2) and $\zeta = 30$ (Case 3).

incompressible, unsteady Stokes equations are discretized on the mesh using the hybridized discontinuous Galerkin (HDG) method from [27,28]. The computer code for this test can be found in `LockExchange.py`.

For this test, the density ratio γ between the two fluids is 0.92. Furthermore, the domain of interest is $\Omega := [0, 30] \times [-0.5, 0.5]$, which is triangulated into $2{,}000 \times 80 \times 2 = 320{,}000$ regular triangular cells. Using the HDG method [28] with $k = 1$ for solving the Stokes equations, the number of dofs in the global system amounts to 1,288,322. A direct sparse matrix solver (`SuperLU`) is used to solve the Stokes system. The total number of particles amounts to 9,408,000, where this number stays constant throughout the computation. Furthermore, 800 time steps of size $\Delta t^* = \Delta t\sqrt{g'/H} = 1.25\text{e}{-2}$ are performed, in which $H = 1$ the channel height, and $g' = g(1 - \gamma)$ the reduced gravity with $\gamma = 0.92$ and $g = 9.81$.

Two different particle–mesh projection configurations are considered. In a first configuration, the local ℓ^2-projections are used to project density and momentum data from the particles to the mesh. The density projection is rendered bound preserving by imposing box constraints on the local least squares problem. The density values attached to a particle



(a) ℓ^2-projections

(b) PDE-constrained projections

Fig. 9. Lock exchange: density field at the mesh at $t^* = 10$ using ℓ^2 or PDE constrained particle–mesh projections.

stay constant throughout the computation, whereas the specific momentum value attached to a particle is updated using Eq. (23) following the Stokes solve.

In the second configuration, the PDE-constrained projection is used for the projection of density and momentum data from the particles to the mesh. This results in a global problem with 964,160 unknowns for the density projection, and 1,928,320 unknowns for the momentum projection. The global systems resulting from the PDE-constrained projections are solved using a GMRES solver in conjunction with an algebraic multigrid preconditioner, where this solver/preconditioner pair is used as a black-box. Furthermore, for the density projection, $\zeta = 20$ to penalize over- and undershoot in the reconstructed density fields. Analogous to the other configuration, the specific momentum value attached to a particle is updated using Eq. (23).

Visually, the mesh density fields at $t^* = 10$ are comparable in terms of the bulk behavior for both projections, Fig. 9, although differences in the small scale features are observed. No further attempts are made to interpret and value these small scale differences between the local ℓ^2-projections and the PDE-constrained projections, other than to say that the PDE-constrained approach results in mass- and momentum-conservative fields, whereas this is not so for the ℓ^2-projection, see Fig. 10.

Timings are reported in Fig. 11, using 32, 64 and 128 CPU cores on the Peta 4 supercomputing facility of the University of Cambridge. Peta 4 contains 768 nodes, equipped with 2 Intel Xeon Skylake 6142 16-core processors each.

Results provide insight into the performance of the different parts, and indicate which parts of the scheme are critical for obtaining higher performance. Clearly, the computational time is dominated by the global solves for the Stokes system, and the PDE-constrained particle–mesh projections, Fig. 11a. The advantage of using iterative solvers for the PDE-projections also becomes clear from this figure. Even though the system for the momentum projection is larger than the system for the Stokes projection, the wallclock time for the momentum projection is considerably smaller and appears to possess better scaling compared to the Stokes solve, see Table 5. Therefore, implementing the iterative solver for the Stokes solver proposed in [49] is believed to be an important step for improving the performance, and is probably indispensable for problems in three spatial dimensions. Noteworthy to mention is that the ℓ^2 particle–mesh projections exhibit excellent scaling properties, on top of their low computational footprint, see Fig. 11b and Table 5.

5.4. Rayleigh–Taylor instability benchmark

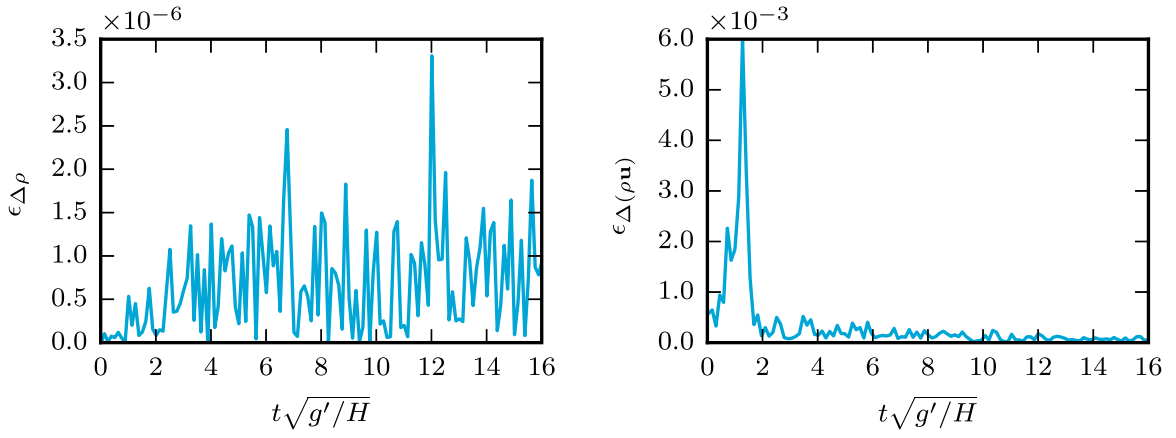
In this example, the applicability of LEoPART for simulating active particle tracing problems is demonstrated. A well-established benchmark from the geodynamics community is used to fit this purpose, and we consider the Rayleigh–Taylor instability community benchmark proposed by van Keken and co-workers [12]. This benchmark involves the evolution of a geodynamic laminar flow whose initial state is a compositionally light material, situated under a considerably thicker and denser layer. Tools from LEoPART are used to discretize the Stokes system using the HDG method, and Lagrangian particles are used for a diffusion-free tracking of the chemical composition field. Conservative composition fields at the mesh are reconstructed from the particle representation using the PDE-constrained projection. The reconstructed composition field is subsequently used to compute a source term $\mathbf{f}$ and a composition-dependent viscosity η in the Stokes equations. The code for this numerical experiment can be found in `RayleighTaylorInstability.py`, considering the following problem description.

Let the domain Ω be the $[0, L]\times[0, 1]$ rectangle, where $L = 0.9142$ is the aspect ratio. $\psi : \Omega \rightarrow [0, 1]$ is the continuum representation of the chemical composition function, with values 0 and 1 corresponding to the light and dense layer. The source term in the momentum component of Stokes' equations is given by

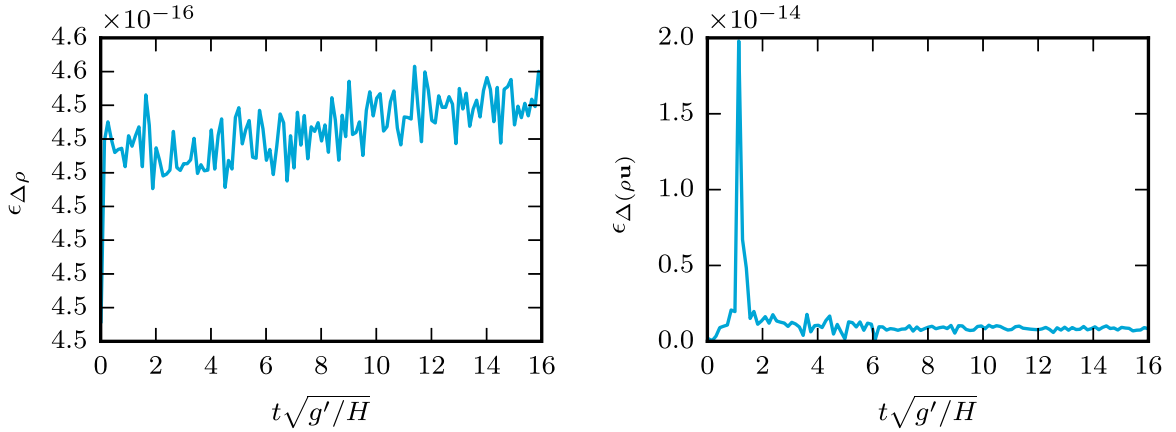
$$\mathbf{f} = \mathrm{Rb}\, \psi\, \hat{\mathbf{g}}, \tag{27}$$

where $\mathrm{Rb} = 1$ is the compositional Rayleigh number and $\hat{\mathbf{g}} = (0, -1)^\top$ is the unit vector acting in the direction of gravity. The viscosity of the Stokes system is dependent on the chemical composition function

$$\eta = \eta_{\text{bottom}} + \psi(\eta_{\text{top}} - \eta_{\text{bottom}}), \tag{28}$$

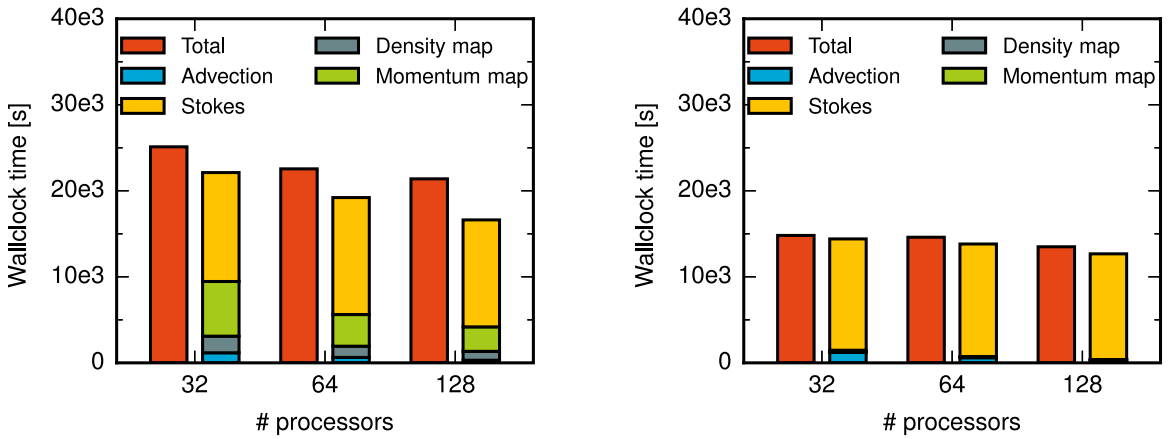


(a) ℓ^2-projection: mass- (left) and momentum error (right).



(b) PDE-constrained projection: mass- (left) and momentum error (right).

Fig. 10. Lock exchange: mass- and momentum conservation error as a function of time for two different particle–mesh projection strategies.



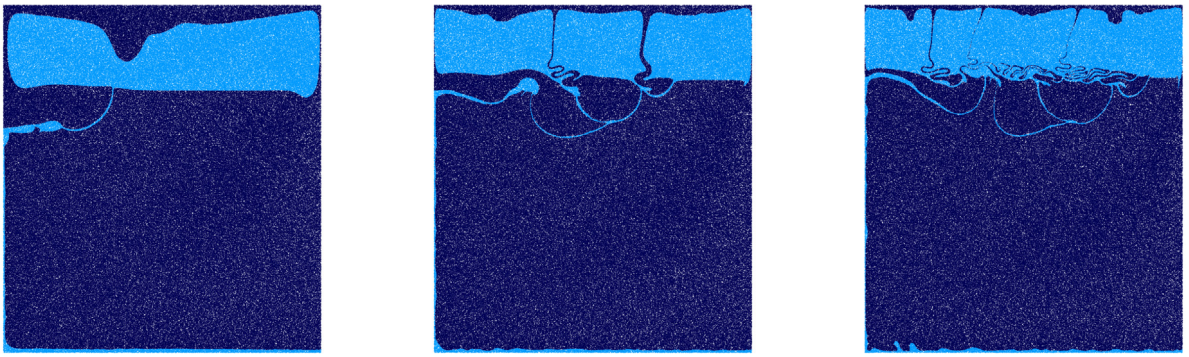
(a) PDE-constrained projections

(b) ℓ^2-projections

Fig. 11. Lock exchange: strong scaling study.

Table 5
Lock exchange test: speed-up of the different model parts in parallel computations, benchmarked against 32 processors run.

	PDE-projections			ℓ^2 projections		
	# Processors			# Processors		
	32	64	128	32	64	128
Particle advection	1	1.85	3.87	1	2.06	3.95
Density projection	1	1.48	1.87	1	1.75	3.75
Momentum projection	1	1.73	2.25	1	1.79	3.99
Stokes solve	1	0.93	1.17	1	0.99	1.05
Total	1	1.11	1.17	1	1.01	1.1



(a) $t = 500$ (b) $t = 1000$ (c) $t = 1500$

Fig. 12. Rayleigh–Taylor instability: the 1 280 000 particles distributed on the fine resolution mesh with $n = 160$ at given simulation times. The dark and light colors represent compositionally dense $\psi = 1$ and light $\psi = 0$ regions, respectively. Cf. [12].

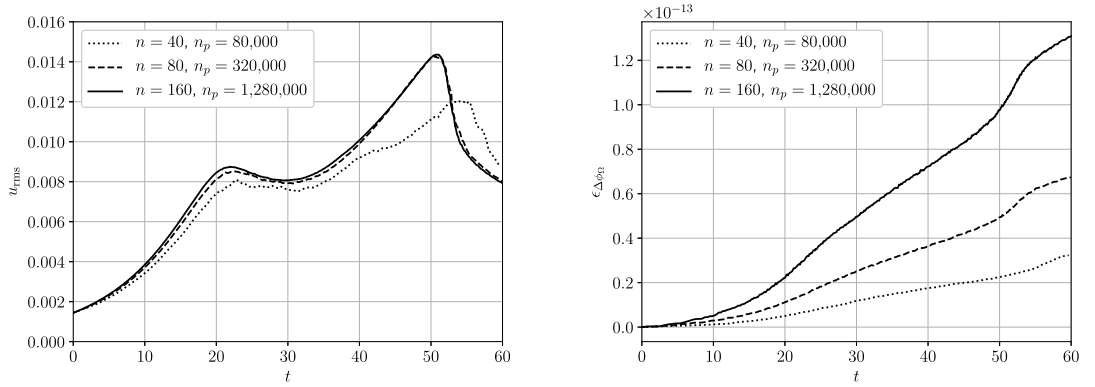
where η_{top} and η_{bottom} are the viscosities of the initial heavy top and light bottom layers, respectively. The initial state of ψ is a small perturbation from the rest state of a light layer of depth $d_b = 0.2$,

$$\psi(\mathbf{x}, t = 0) = \begin{cases} 0 & y < d_b + 0.02 \cos\left(\frac{\pi x}{L}\right), \\ 1 & \text{otherwise.} \end{cases} \tag{29}$$

The boundary conditions are set such that $\mathbf{u} = \mathbf{0}$ on the bottom ($y = 0$) and top ($y = 1$) boundaries, and on the left ($x = 0$) and right ($x = L$) boundaries, freeslip conditions are applied, i.e. $\mathbf{u} \cdot \boldsymbol{n} = 0$ and $\boldsymbol{t} \cdot \left(\nabla \mathbf{u} + \nabla \mathbf{u}^{\top}\right) \boldsymbol{n} = \mathbf{0}$ where $\boldsymbol{n}$ the outward pointing unit normal vector and $\boldsymbol{t}$ is the unit tangent vector to the boundary. Furthermore, the boundary conditions imposed on the chemical composition function are $\psi = 1$ on top ($y = 1$) and $\psi = 0$ on the bottom ($y = 0$) boundaries, respectively.

The domain is triangulated into $2n^2$ regular simplices, with $n = (40, 80, 160)$, yielding three meshes with 3200, 12800, and 51200 cells respectively. The mesh is then displaced in order to align the cells with the initial viscosity discontinuity described in (29). Each cell is assigned 25 particles, carrying a composition value ψ_p, resulting in 80 000, 320 000 and 1 280 000 particles in the meshes, respectively. This number of particles remains constant throughout the simulation. Furthermore, the time step size is chosen based on the Courant–Friedrichs–Lewy (CFL) criterion, i.e. $\Delta t_j = C_{\text{CFL}} h_{\min} / \max_{\Omega} |\mathbf{u}(t_j)|$ where t_j is the jth time step, $h_{\min}$ is the minimum mesh cell diameter and $C_{\text{CFL}} > 0$ is the CFL parameter, chosen here to be $C_{\text{CFL}} = 0.5$. The HDG method is used with $k = 1$ to solve the Stokes system. After computing the solution approximation of the Stokes system, the particles are advected. Given ψ_p, the conservative PDE-constrained projection is used to update the composition field ψ_h on the mesh and thereby the source term $\mathbf{f}$ and the viscosity η. The composition function ψ_h is represented by the $k = 1$ DG-finite element space, and we choose $\zeta = 25$ to penalize over- and undershoot of the reconstructed field.

The benchmark [12] considers three cases, $\eta_{\text{top}}/\eta_{\text{bottom}} \in \{1, 10, 100\}$. For brevity we document the case with a 100–fold difference in the viscosity layers, namely, $\eta_{\text{top}} = 1$ and $\eta_{\text{bottom}} = 0.01$. For comparison with [12], the distribution of the 1 280 000 particles in the $n = 160$ mesh at simulation times $t = 500$, 1000 and 1500 are shown in Fig. 12. Noteworthy to mention is that the particle distribution remains uniform throughout. This feature owes to HDG discretization of the Stokes system, yielding pointwise div-free velocity fields by which the particles are advected [17].



(a) RMS velocity (b) Mass conservation error

Fig. 13. The computed RMS velocity functional and mass conservation error of the Rayleigh–Taylor instability problem. Here n is the number of cells along one axis of the mesh and n_p is the number of particles used in the simulation.

Table 6
Computed functionals from the Rayleigh–Taylor instability simulation and comparison with other works.

Code	Grid	Method	γ	$t\,(\max u_{\text{rms}})$	$\max u_{\text{rms}}$
van Keken et al. [12]	100 × 100	Splines / marker chain	0.1024	51.12	0.01385
	80 × 80	$C1$–element / marker chain	0.1025	51.23	0.01448
Vynnytska et al. [50]	40 × 40	Taylor–Hood / DG	0.0976	57.21	0.01140
	80 × 80	Taylor–Hood / DG	0.1018	52.11	0.01444
	160 × 160	Taylor–Hood / DG	0.1039	51.55	0.01458
This work	40 × 40	HDG / PDE-constrained projection	0.08707	54.81	0.01208
	80 × 80	HDG / PDE-constrained projection	0.09161	50.68	0.01428
	160 × 160	HDG / PDE-constrained projection	0.09677	50.80	0.01436

A quantitative comparison with literature results from [12,50] is made by investigating the root mean square (RMS) velocity, u_{rms}, the mass conservation error, $\epsilon_{\Delta\psi_\Omega}$, and growth rate, γ, where

$$u_{\text{rms}} = \sqrt{\frac{\int_\Omega \mathbf{u}\cdot\mathbf{u}\,\mathrm{d}\Omega}{\int_\Omega \mathrm{d}\Omega}}, \quad \epsilon_{\Delta\psi_\Omega} = \left|\int_\Omega (\psi_h(\mathbf{x},t) - \psi_h(\mathbf{x},0))\,\mathrm{d}\Omega\right|, \quad \gamma = \frac{\ln(u_{\text{rms}}(t_1)) - \ln(u_{\text{rms}}(0))}{t_1}, \tag{30}$$

and t_1 is the simulation time at the first time step. The computed quantities (30) are shown in Fig. 13, and key results are reported in Table 6. This includes comparison to the marker chain method of [12] and the case with no artificial diffusion in [50], showing good agreement for the growth rate γ and the RMS-velocity. On top of this, discrete conservation for the composition field ψ_h can be ensured, Fig. 13b, when using the PDE-constrained particle–mesh projection provided by LEoPart.

5.5. Rayleigh–Taylor instability – 3D

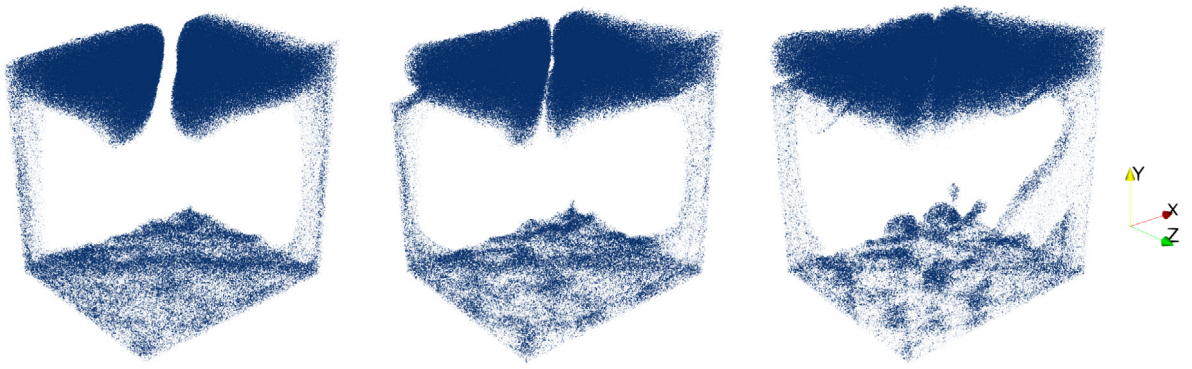
We finally demonstrate the use of LEoPart for 3D simulations. The 3D example in this section is constructed by extruding the Rayleigh–Taylor instability benchmark from the previous section. Let the domain $\Omega := [0, L_x]\times[0,1]\times[0,L_z]$ where $L_x = 0.9142$ and $L_z = 0.8142$. The momentum source and viscosity model is as described in (27) and (28), respectively. The initial perturbed composition field is

$$\psi(\mathbf{x}, t=0) = \begin{cases} 0 & y < d_b + 0.02\cos\left(\frac{\pi x}{L_x}\right)\cos\left(\frac{\pi z}{L_z}\right), \\ 1 & \text{otherwise.} \end{cases} \tag{31}$$

The boundary conditions are imposed as described in the previous section with the addition of a free slip boundary condition enforced on the near ($z = 0$) and far ($z = L_z$) boundaries. All other parameters are as described in the previous section. The code for this example can be found in `RayleighTaylorInstability3D.py`

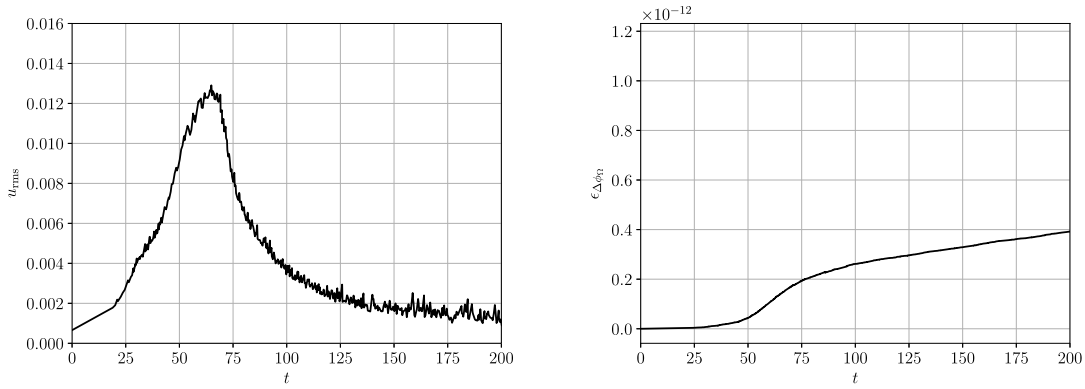
The domain is divided into $6n^3$ tetrahedra where $n = 20$. The mesh is displaced to align with the viscosity discontinuity in Eq. (31). Each cell is assigned 50 particles, carrying a composition value ψ_p, such that there is a total of 2 400 000 particles used in the simulation.

Evidently, Fig. 13a shows that high resolution meshes are mandatory for accurate results. To solve large 3D problems efficiently we refer to an example of a preconditioner designed for the iterative solution of the HDG system [49].



(a) $t = 250$ (b) $t = 500$ (c) $t = 1000$

Fig. 14. 3D Rayleigh–Taylor instability: the compositionally light ($\psi = 0$) particles distributed on the mesh where $n = 20$ at given simulation times.



(a) RMS velocity (b) Mass conservation error

Fig. 15. The computed RMS velocity functional and mass conservation error of the 3D Rayleigh–Taylor instability problem. Here $n = 20$ is the number of cells along one axis of the mesh and $n_p = 2400000$ is the number of particles used in the simulation.

Implementation of this preconditioned system forms a programme of future development in LEoPart. In this example we use the direct solver MUMPS to solve the underlying linear system.

The tracer distribution of the compositionally light ($\psi = 0$) material is shown in Fig. 14. We see the formation of two opposing diapirs competing for space at the top of the domain. The interface separating the two diapirs is formed by a downwelling of compositionally dense material ($\psi = 1$) from the top of the domain.

We show RMS velocity and mass conservation in Fig. 15. Here we confirm the mass conserving property of the PDE-constrained projection of the particle composition values to the composition function. Furthermore the two competing diapirs evolve more slowly than the single diapir exhibited in the previous section.

6. Conclusion and outlook

This paper introduced LEoPart [21], an open-source library which integrates a suite of tools for Lagrangian particle advection and different particle–mesh interaction strategies in FEniCS. To efficiently implement the particle advection, particles are tracked on the simplicial mesh using the convex polyhedron method. As demonstrated in the numerical examples, the particle advection exhibits near optimal performance for distributed-memory parallel runs. Furthermore, several options are available for the projection of particle data onto mesh fields and *vice versa*. In particular, the PDE-constrained particle–mesh projection allows to track particle quantities on the mesh in an accurate, diffusion-free, and conservative manner.

A number of application examples in two and three spatial dimensions demonstrated how LEoPart can be used to track sharp interfaces in immiscible, multiphase flows or long time scale processes such as pertaining to geodynamics. Yet, we believe that the particle(-mesh) functionality in LEoPart can be of practical relevance to a much wider range of flow problems, including groundwater modeling, atmospheric modeling, and reproducing particle image velocimetry measurements in numerical simulations. Of particular interest are applications characterized by low physical diffusion, for

which the presented particle–mesh tools allow to maintain sharp features at subgrid level without introducing numerical diffusion.

LEoPart [21] will be maintained at the cited URL, and the community is invited to contribute to this project. Upcoming developments in LEoPart include:

- **Update to dolfin-x**: Recent developments in FEniCS have been taking place in the dolfin-x repository, which has diverged significantly from the original dolfin. It is planned to migrate LEoPart to the new underlying library. This will require many changes, including basic geometry handling and assembly of forms. However, we can expect to see some performance enhancements as a result, especially as it will be easier to assemble block diagonal matrices and preconditioners to solve the Stokes equations iteratively.
- **Iterative solvers:** as demonstrated in the numerical examples, all the components exhibit excellent scaling for distributed-memory parallel runs, except for solving the global systems which arise in the PDE-constrained projection and the incompressible Stokes equations. To improve the performance for these steps, the implementation of scalable iterative solvers heads our wish list. The optimal preconditioner presented by [49] will serve as a starting point for the Stokes system, whereas implementation of a GMRES-based solver is considered for the PDE-constrained projection.
- **Particle advection:** at the time of writing LEoPart supports an explicit Euler, and a two- or three-stage Runge–Kutta scheme for the particle-advection. In view of particle–mesh operator splitting applications, LEoPart will benefit from supporting multi-step schemes as this opens the way for implementing implicit–explicit (IMEX) operator splitting schemes [51]. Theoretically, such schemes can push particle–mesh operator splitting techniques beyond the second order time accuracy as reported in [17,18].

CRediT authorship contribution statement

Jakob M. Maljaars: Conceptualization, Methodology, Software, Writing - original draft, Writing - review & editing, Visualization, Supervision. **Chris N. Richardson:** Software, Resources, Writing - review & editing. **Nathan Sime:** Software, Visualization, Writing - review & editing.

Acknowledgments

The Netherlands Organisation for Scientific Research (NWO) is acknowledged for supporting JMM through the JMBC-EM Graduate Programme research grant. JMM further acknowledges CRUX Engineering for their time investment in the revision stage of the manuscript. CNR is supported by EPSRC, UK Grant EP/N018877/1. NS is supported by the Carnegie Institution for Science President's Fellowship. NS further wishes to thank Peter E. van Keken and Cian R. Wilson for their advice.

Appendix. PDE-constrained particle–mesh interaction

This appendix presents the discrete optimality system which is obtained by equating the variations of Eq. (16) with respect to the three unknowns $(\psi_h, \lambda_h, \bar{\psi}_h) \in (W_h, T_h, \bar{W}_h)$ to zero, and performing a θ time integration. A detailed derivation can be found in [18].

To set the stage, let $\mathcal{T} := \{K\}$ be the triangulation of Ω into open, non-overlapping cells K, having outward pointing unit normal vector $\boldsymbol{n}$ on its boundary ∂K. Adjacent cells K_i and K_j $(i \neq j)$ share a common facet $F = \partial K_i \cap \partial K_j$. The set of all facets (including the exterior boundary facets $F = \partial K \cap \partial\Omega$) is denoted by $\mathcal{F}$. The following scalar finite element spaces are defined on $\mathcal{T}$ and $\mathcal{F}$:

$$W_h := \left\{ w_h \in L^2(\mathcal{T}),\ w_h|_K \in P_k(K)\ \forall\ K \in \mathcal{T} \right\}, \tag{A.1}$$

$$T_h := \left\{ \tau_h \in L^2(\mathcal{T}),\ \tau_h|_K \in P_l(K)\ \forall\ K \in \mathcal{T} \right\}, \tag{A.2}$$

$$\bar{W}_h := \left\{ \bar{w}_h \in L^2(\mathcal{F}),\ \bar{w}_h|_F \in P_k(F)\ \forall\ F \in \mathcal{F} \right\}, \tag{A.3}$$

in which $P(K)$ and $P(F)$ denote the spaces spanned by Lagrange polynomials on K and F, respectively, and $k \geq 1$ and $l \geq 0$ indicating the polynomial order. Note that W_h in Eq. (A.1) is equal to Eq. (11).

Given these function space definitions, the fully-discrete optimality system is obtained after taking variations of the Lagrangian functional Eq. (16) with respect to $\left(\psi_h, \lambda_h, \bar{\psi}_h\right) \in \left(W_h, T_h \bar{W}_h\right)$ and using a θ-method for the time discretization. The fully-discrete co-state equation in this optimality system reads: given the particle–field $\psi_p^n \in \Psi_t$, the particle positions $\mathbf{x}_p^{n+1} \in \mathcal{X}_t$, and the intermediate field $\psi_h^{*,n} \in W_h$, find $\left(\psi_h^{n+1}, \lambda_h^{n+1}, \bar{\psi}_h^{n+1}\right) \in \left(W_h, T_h, \bar{W}_h\right)$ such that

$$\sum_{p \in \mathcal{S}_t} \left(\psi_h^{n+1}(\mathbf{x}_p^{n+1}) - \psi_p^n\right) w_h(\mathbf{x}_p^{n+1}) - \sum_K \oint_{\partial K} \beta \left(\bar{\psi}_h^{n+1} - \psi_h^{n+1}\right) w_h \mathrm{d}\Gamma + \sum_K \int_{\partial K} \zeta \nabla \psi_h^{n+1} \cdot \nabla w_h \mathrm{d}\Omega$$
$$+ \int_\Omega \frac{w_h}{\Delta t_n} \lambda_h^{n+1} \mathrm{d}\Omega - \theta \sum_K \int_K (\mathbf{a} w_h) \cdot \nabla \lambda_h^{n+1} \mathrm{d}\Omega = 0 \quad \forall\ w_h \in W_h\ . \tag{A.4a}$$

Correspondingly, the fully-discrete counterpart of the state equation becomes:

$$\int_\Omega \frac{\psi_h^{n+1}-\psi_h^{*,n}}{\Delta t_n}\tau_h \mathrm{d}\Omega - \theta\sum_K\int_K \left(\mathbf{a}\psi_h^{n+1}\right)\cdot\nabla\tau_h \mathrm{d}\Omega + \sum_K \oint_{\partial K}\mathbf{a}\cdot\boldsymbol{n}\,\bar{\psi}_h^{n+1}\tau_h \mathrm{d}\Gamma = (1-\theta)\sum_K\int_K\left(\mathbf{a}\psi_h^{*,n}\right)\cdot\nabla\tau_h\mathrm{d}\Omega \quad \forall\ \tau_h\in T_h. \tag{A.4b}$$

Finally, the fully-discrete optimality condition reads

$$\sum_K\oint_{\partial K}\mathbf{a}\cdot\boldsymbol{n}\,\lambda_h^{n+1}\bar{w}_h\mathrm{d}\Gamma + \sum_K\oint_{\partial K}\beta\left(\bar{\psi}_h^{n+1}-\psi_h^{n+1}\right)\bar{w}_h\mathrm{d}\Gamma = 0 \qquad \forall\ \bar{w}_h\in\bar{W}_h. \tag{A.4c}$$

Note that the Lagrange multiplier λ_h and the control variable $\bar{\psi}_h$ are conveniently chosen at time level $n+1$, which is allowed since these variables are fully-implicit, not requiring differentiation in time. Furthermore, the choice $l=0$ for the polynomial order of the Lagrange multiplier field λ_h bears specific advantage in that the terms involving the time-stepping parameter θ can be dropped, i.e. for $l=0$ Eq. (A.4) becomes independent of θ.

Eq. (A.4) can be casted in a 3×3 block system, see Eq. (17). This system of equations is solved for the three unknowns $\left(\psi_h^{n+1},\lambda_h^{n+1},\bar{\psi}_h^{n+1}\right)\in\left(W_h,T_h,\bar{W}_h\right)$ via a static condensation procedure, see the discussion in Section 4.1.3.

References

[1] A.C. Bagtzoglou, D.E. Dougherty, A.F.B. Tompson, Application of particle methods to reliable identification of groundwater pollution sources, Water Resour. Manag. 6 (1) (1992) 15–23, http://dx.doi.org/10.1007/BF00872184.

[2] E.J. Delhez, J.-M. Campin, A.C. Hirst, E. Deleersnijder, Toward a general theory of the age in ocean modelling, Ocean Model. 1 (1) (1999) 17–27, http://dx.doi.org/10.1016/S1463-5003(99)00003-7.

[3] E. Deleersnijder, J.-M. Campin, E.J. Delhez, The concept of age in marine modelling: I. Theory and preliminary model results, J. Mar. Syst. 28 (3–4) (2001) 229–267, http://dx.doi.org/10.1016/S0924-7963(01)00026-4.

[4] P.J. Tackley, S.D. King, Testing the tracer ratio method for modeling active compositional fields in mantle convection simulations, Geochem. Geophys. Geosyst. 4 (4) (2003) http://dx.doi.org/10.1029/2001GC000214.

[5] N.T. Ouellette, H. Xu, E. Bodenschatz, A quantitative study of three-dimensional Lagrangian particle tracking algorithms, Exp. Fluids 40 (2) (2006) 301–313, http://dx.doi.org/10.1007/s00348-005-0068-7.

[6] J. Westerweel, G.E. Elsinga, R.J. Adrian, Particle image velocimetry for complex and turbulent flows, Annu. Rev. Fluid Mech. 45 (1) (2013) 409–436, http://dx.doi.org/10.1146/annurev-fluid-120710-101204.

[7] M. Raffel, C.E. Willert, F. Scarano, C.J. Kähler, S.T. Wereley, J. Kompenhans, Particle Image Velocimetry: A Practical Guide, Springer, 2018.

[8] E. Hathway, C. Noakes, P. Sleigh, L. Fletcher, CFD simulation of airborne pathogen transport due to human activities, Build. Environ. 46 (12) (2011) 2500–2511, http://dx.doi.org/10.1016/J.BUILDENV.2011.06.001.

[9] D. Cohen Stuart, C. Kleijn, S. Kenjereš, An efficient and robust method for Lagrangian magnetic particle tracking in fluid flow simulations on unstructured grids, Comput. Fluids 40 (1) (2011) 188–194, http://dx.doi.org/10.1016/J.COMPFLUID.2010.09.001.

[10] S. Pope, PDF methods for turbulent reactive flows, Prog. Energy Combust. Sci. 11 (2) (1985) 119–192, http://dx.doi.org/10.1016/0360-1285(85)90002-4.

[11] Y. Zhang, D. Haworth, A general mass consistency algorithm for hybrid particle/finite-volume PDF methods, J. Comput. Phys. 194 (1) (2004) 156–193, http://dx.doi.org/10.1016/j.jcp.2003.08.032.

[12] P.E. van Keken, S.D. King, H. Schmeling, U.R. Christensen, D. Neumeister, M.-P. Doin, A comparison of methods for the modeling of thermochemical convection, J. Geophys. Res.: Solid Earth 102 (1997) 22, http://dx.doi.org/10.1029/97JB01353.

[13] U.R. Christensen, A.W. Hofmann, Segregation of subducted oceanic crust in the convecting mantle, J. Geophys. Res.: Solid Earth 99 (1994) 19, http://dx.doi.org/10.1029/93JB03403.

[14] J.P. Brandenburg, P.E. van Keken, Deep storage of oceanic crust in a vigorously convecting mantle, J. Geophys. Res.: Solid Earth 112 (2007) B06403, http://dx.doi.org/10.1029/2006JB004813.

[15] E. Edwards, R. Bridson, A high-order accurate particle-in-cell method, Internat. J. Numer. Methods Engrg. 90 (9) (2012) 1073–1088, http://dx.doi.org/10.1002/nme.3356.

[16] D.M. Kelly, Q. Chen, J. Zang, PICIN: a particle-in-cell solver for incompressible free surface flows with two-way fluid-solid coupling, SIAM J. Sci. Comput. 37 (3) (2015) 403–424, http://dx.doi.org/10.1137/140976911.

[17] J.M. Maljaars, R.J. Labeur, M. Möller, A hybridized discontinuous Galerkin framework for high-order particle–mesh operator splitting of the incompressible Navier–Stokes equations, J. Comput. Phys. 358 (2018) 150–172, http://dx.doi.org/10.1016/j.jcp.2017.12.036.

[18] J.M. Maljaars, R.J. Labeur, N. Trask, D. Sulsky, Conservative, high-order particle-mesh scheme with applications to advection-dominated flows, Comput. Methods Appl. Mech. Engrg. 348 (2019) 443–465, http://dx.doi.org/10.1016/J.CMA.2019.01.028.

[19] J.M. Maljaars, R.J. Labeur, N. Trask, D. Sulsky, Optimization based Particle-Mesh algorithm for high-order and Conservative Scalar Transport, in: A. Corsini, S. Perotto, H.E. van Brummelen, G. Rozza (Eds.), Numerical Methods for Flows - FEF 2017 Selected Contributions, in: Lecture Notes in Computational Science and Engineering, Springer, 2019, http://dx.doi.org/10.1007/978-3-030-30705-9_23, Ch. 23.

[20] J. Maljaars, R.J. Labeur, M. Möller, W. Uijttewaal, Development of a hybrid particle-mesh method for simulating free-surface flows, J. Hydrodyn. Ser. B 29 (3) (2017) 413–422, http://dx.doi.org/10.1016/S1001-6058(16)60751-5.

[21] Leopart source code, https://bitbucket.org/jakob_maljaars/leopart/.

[22] A. Logg, K.-A. Mardal, G.N. Wells, Automated Solution of Differential Equations by the Finite Element Method, Vol. 84, Springer Science & Business Media, 2012, p. 724, http://dx.doi.org/10.1007/978-3-642-23099-8.

[23] S. Balay, S. Abhyankar, M.F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W.D. Gropp, D. Karpeyev, D. Kaushik, M.G. Knepley, D.A. May, L.C. McInnes, R.T. Mills, T. Munson, K. Rupp, P. Sanan, B.F. Smith, S. Zampini, H. Zhang, H. Zhang, PETSc Users Manual, Tech. Rep. ANL-95/11 - Revision 3.12, Argonne National Laboratory, 2019, URL https://www.mcs.anl.gov/petsc/petsc-current/docs/manual.pdf.

[24] W. Bangerth, J. Dannberg, R. Gassmoeller, T. Heister, et al., ASPECT v2.1.0 [software], 2019, http://dx.doi.org/10.5281/zenodo.2653531.

[25] R. Gassmöller, H. Lokavarapu, E. Heien, E.G. Puckett, W. Bangerth, Flexible and scalable particle-in-cell methods with adaptive mesh refinement for geodynamic computations, Geochem. Geophys. Geosyst. 19 (9) (2018) 3596–3604, http://dx.doi.org/10.1029/2018GC007508.

[26] D. Arndt, W. Bangerth, T.C. Clevenger, D. Davydov, M. Fehling, D. Garcia-Sanchez, G. Harper, T. Heister, L. Heltai, M. Kronbichler, R.M. Kynch, M. Maier, J.-P. Pelteret, B. Turcksin, D. Wells, The `deal.ii` library, version 9.1, J. Numer. Math. (2019) http://dx.doi.org/10.1515/jnma-2019-0064.
[27] S. Rhebergen, G.N. Wells, A hybridizable discontinuous Galerkin method for the Navier–Stokes equations with pointwise divergence-free velocity field, J. Sci. Comput. (2018) 1–18, http://dx.doi.org/10.1007/s10915-018-0671-4.
[28] S. Rhebergen, G.N. Wells, An embedded–hybridized discontinuous Galerkin finite element method for the Stokes equations, Comput. Methods Appl. Mech. Engrg. 358 (2020) 1–18, http://dx.doi.org/10.1016/j.cma.2019.112619.
[29] C. Rocchini, P. Cignoni, Generating random points in a tetrahedron, J. Graph. Tools 5 (4) (2000) 9–12.
[30] M. Muradoglu, A.D. Kayaalp, An auxiliary grid method for computations of multiphase flows in complex geometries, J. Comput. Phys. 214 (2) (2006) 858–877, http://dx.doi.org/10.1016/J.JCP.2005.10.024.
[31] R. Löhner, J. Ambrosiano, A vectorized particle tracer for unstructured grids, J. Comput. Phys. 91 (1) (1990) 22–31, http://dx.doi.org/10.1016/0021-9991(90)90002-I.
[32] J. Maljaars, A hybrid particle-mesh method for simulating free surface flows, 2016, URL http://resolver.tudelft.nl/uuid:d894370d-f6df-4433-8ee0-7692a43e857a.
[33] D. Haworth, Progress in probability density function methods for turbulent reacting flows, Prog. Energy Combust. Sci. 36 (2) (2010) 168–259, http://dx.doi.org/10.1016/j.pecs.2009.09.003.
[34] P.P. Popov, R. McDermott, S.B. Pope, An accurate time advancement algorithm for particle tracking, J. Comput. Phys. 227 (20) (2008) 8792–8806, http://dx.doi.org/10.1016/j.jcp.2008.06.021.
[35] Z. Chen, Z. Zong, M.B. Liu, H.T. Li, A comparative study of truly incompressible and weakly compressible SPH methods for free surface incompressible flows, Internat. J. Numer. Methods Fluids 73 (9) (2013) 625–638.
[36] R. Garg, C. Narayanan, D. Lakehal, S. Subramaniam, Accurate numerical estimation of interphase momentum transfer in Lagrangian-Eulerian simulations of dispersed two-phase flows, Int. J. Multiph. Flow 33 (12) (2007) 1337–1364, http://dx.doi.org/10.1016/j.ijmultiphaseflow.2007.06.002.
[37] G. Jacobs, J. Hesthaven, High-order nodal discontinuous Galerkin particle-in-cell method on unstructured grids, J. Comput. Phys. 214 (1) (2006) 96–121, http://dx.doi.org/10.1016/j.jcp.2005.09.008.
[38] D. Goldfarb, A. Idnani, A numerically stable dual method for solving strictly convex quadratic programs, Math. Program. 27 (1) (1983) 1–33, http://dx.doi.org/10.1007/BF02591962.
[39] R.J. Labeur, G.N. Wells, Energy stable and momentum conserving hybrid finite element method for the incompressible Navier–Stokes equations, SIAM J. Sci. Comput. 34 (2) (2012) 889–913, http://dx.doi.org/10.1137/100818583.
[40] G.N. Wells, Analysis of an interface stabilized finite element method: The Advection-Diffusion-Reaction equation, SIAM J. Numer. Anal. 49 (1) (2011) 87–109, http://dx.doi.org/10.1137/090775464.
[41] N. Nguyen, J. Peraire, B. Cockburn, An implicit high–order hybridizable discontinuous Galerkin method for linear convection–diffusion equations, J. Comput. Phys. 228 (9) (2009) 3232–3254, http://dx.doi.org/10.1016/j.jcp.2009.01.030.
[42] D. Snider, An incompressible three-dimensional multiphase particle-in-cell model for dense particle flows, J. Comput. Phys. 170 (2) (2001) 523–549, http://dx.doi.org/10.1006/jcph.2001.6747.
[43] R. McDermott, S. Pope, The parabolic edge reconstruction method (PERM) for Lagrangian particle advection, J. Comput. Phys. 227 (11) (2008) 5447–5491, http://dx.doi.org/10.1016/j.jcp.2008.01.045.
[44] J. Brackbill, H. Ruppel, FLIP: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions, J. Comput. Phys. 65 (2) (1986) 314–343, http://dx.doi.org/10.1016/0021-9991(86)90211-1.
[45] R.J. LeVeque, High-Resolution Conservative algorithms for advection in incompressible flow, SIAM J. Numer. Anal. 33 (2) (1996) 627–665, http://dx.doi.org/10.1137/0733033.
[46] S.T. Zalesak, Fully multidimensional flux-corrected transport algorithms for fluids, J. Comput. Phys. 31 (3) (1979) 335–362, http://dx.doi.org/10.1016/0021-9991(79)90051-2.
[47] D. Enright, R. Fedkiw, J. Ferziger, I. Mitchell, A hybrid particle level set method for improved interface capturing, J. Comput. Phys. 183 (1) (2002) 83–116, http://dx.doi.org/10.1006/jcph.2002.7166.
[48] R. Scardovelli, S. Zaleski, Interface reconstruction with least-square fit and split eulerian-Lagrangian advection, Internat. J. Numer. Methods Fluids 41 (3) (2003) 251–274, http://dx.doi.org/10.1002/fld.431.
[49] S. Rhebergen, G.N. Wells, Preconditioning of a hybridized discontinuous Galerkin finite element method for the Stokes equations, J. Sci. Comput. 77 (3) (2018) 1936–1952, http://dx.doi.org/10.1007/s10915-018-0760-4.
[50] L. Vynnytska, M.E. Rognes, S.R. Clark, Benchmarking FEniCS for mantle convection simulations, Comput. Geosci. 50 (2013) 95–105, http://dx.doi.org/10.1016/j.cageo.2012.05.012.
[51] G.E. Karniadakis, M. Israeli, S.A. Orszag, High-order splitting methods for the incompressible Navier-Stokes equations, J. Comput. Phys. 97 (2) (1991) 414–443, http://dx.doi.org/10.1016/0021-9991(91)90007-8.