LEoPart

Release 0.0.0

Dec 06, 2021

Contents:

LEoPart is a particle add-on for the FEniCS finite element library. Along with installation instructions and a tutorial to helpy you getting started, this page provides the documentation for the Python API of LEoPart!

Detailed background information can be found in one of the following papers:

```
@article{maljaars2020,
  title={LEoPart: a particle library for FEniCS},
  author={Maljaars, Jakob M and Richardson, Chris N and Sime, Nathan},
  journal={arXiv preprint arXiv:1912.13375},
  year={2019}
}

@article{Maljaars2019,
  author = {Maljaars, Jakob M. and Labeur, Robert Jan and Trask, Nathaniel and Sulsky,
↪ Deborah},
  doi = {10.1016/J.CMA.2019.01.028},
  issn = {0045-7825},
  journal = {Comput. Methods Appl. Mech. Eng.},
  month = {jan},
  pages = {443--465},
  publisher = {North-Holland},
  title = {{Conservative, high-order particle-mesh scheme with applications to␣
↪advection-dominated flows}},
  volume = {348},
  year = {2019}
}
```

CHAPTER 1

Contributors

- Jakob Maljaars
- Chris Richardson
- Nate Sime

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

2.1 Installation

Clone LEoPart using `git` into a subdiractory of you location of choice with:

```
git clone https://bitbucket.org/jakob_maljaars/leopart/
cd leopart
```

To compile and run LEoPart, the following dependencies are required:

- FEniCS
- pybind11
- CMAKE

A `conda` environment is provided containing all the dependencies. To get this environment up-and-running:

```
conda create -f envs/environment.yml
conda activate leopart
```

Nexct, Compile the `cpp` source code by running:

```
cd source/cpp
cmake . && make
```

And install as python package:

```
cd ../..
[sudo] python3 setup.py install
```

You now should be able to use `leopart` from `python` as:

```
import leopart as lp
```

Or any appropriate import syntax.

2.2 Getting started

For illustration purpposes, let's go step by step through a simple example where a slotted disk rotates through a (circle-shaped) mesh. The rotation of the slotted disk is governed by the solid body rotation

$$\mathbf{u} = \pi[-y, x]^\top$$

The slotted disk is represented on a set of particles, see the following picture of what could be the initial particle field.

Our objective is to project the scattered particle representation of the (rotating!) slotted disk onto a FEM mesh in such a way that the area is preserved (i.e. mass conservation). Apart from conservation, we certainly want the particle-mesh projection to be accurate as well. To achieve this, the PDE-constrained particle-mesh projection from Maljaars et al [2019] is used. This projection is derived from the Lagrangian functional

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

An in-depth interpretation and analysis of this functional and the optimality system that results 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 the aformentioned reference.

First, let's import the required tools from `leopart`:

```
from leopart import (
    particles,
    advect_rk3,
    PDEStaticCondensation,
    FormsPDEMap,
    RandomCircle,
    SlottedDisk,
    assign_particle_values,
    l2projection,
)
```

And import a number of tools from `dolfin` and other libraries:

```
from dolfin import (
Expression,
Point,
VectorFunctionSpace,
Mesh,
Constant,
FunctionSpace,
assemble,
dx,
refine,
```

(continues on next page)

(continued from previous page)

```
Function,
assign,
DirichletBC,
)

from mpi4py import MPI as pyMPI
import numpy as np

comm = pyMPI.COMM_WORLD
```

Next is to further specify the geometry of the domain and the slotted disk. Furthermore, we import the disk-shaped mesh, which is refined twice in order to increase the spatial resolution

```
(x0, y0, r) = (0.0, 0.0, 0.5)
(xc, yc, r) = (-0.15, 0.0)
(r, rdisk) = (0.5, 0.2)
rwidth = 0.05
(lb, ub) = (0.0, 1.0)

# Mesh
mesh = Mesh("./../../meshes/circle_0.xml")
mesh = refine(refine(refine(mesh)))
```

The slotted cylinder is created with the `SlottedDisk` class, this is nothing more than just a specialized `dolfin.UserExpression` and provided in the `leopart` package just for convenience:

```
psi0_expr = SlottedDisk(
    radius=rc, center=[xc, yc], width=rwidth, depth=0.0, degree=3, lb=lb, ub=ub
)
```

The timestepping parameters are chosen such that we precisely make one rotation:

```
Tend = 2.0
dt = Constant(0.02)
num_steps = np.rint(Tend / float(dt))
```

In order to prepare the PDE-constrained projection, the function spaces W_h, T_h and $\bar{W}_h$ are defined. Note that $\bar{W}_h$ is defined only on the mesh facets. Apart from the function space definitions, we also set a homogeneous Dirichlet boundary condition on $\bar{W}_h$, and we define the advective velocity field.

```
W = FunctionSpace(mesh, "DG", 1)
T = FunctionSpace(mesh, "DG", 0)
Wbar = FunctionSpace(mesh, "DGT", 1)

bc = DirichletBC(Wbar, Constant(0.0), "on_boundary")

(psi_h, psi_h0, psi_h00) = (Function(W), Function(W), Function(W))
psibar_h = Function(Wbar)

V = VectorFunctionSpace(mesh, "DG", 3)
uh = Function(V)
uh.assign(Expression(("-Uh*x[1]", "Uh*x[0]"), Uh=np.pi, degree=3))
```

We are now all set to define the particles. So we start with creating a set of point locations and setting a scalar property that, for instance, defines the concentration. Note that `leopart` comes shipped with a number of particle generators, of which the `RandomCircle` method is just one.

```
x = RandomCircle(Point(x0, y0), r0).generate([750, 750])
s = assign_particle_values(x, psi0_expr)
```

. . . and define both the particle object and a particle-advection scheme (in this case a Runge-Kutta 3 scheme)

The optimality system that results from minimizing the Lagrangian can be represented as a 3x3 block matrix at the element level

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

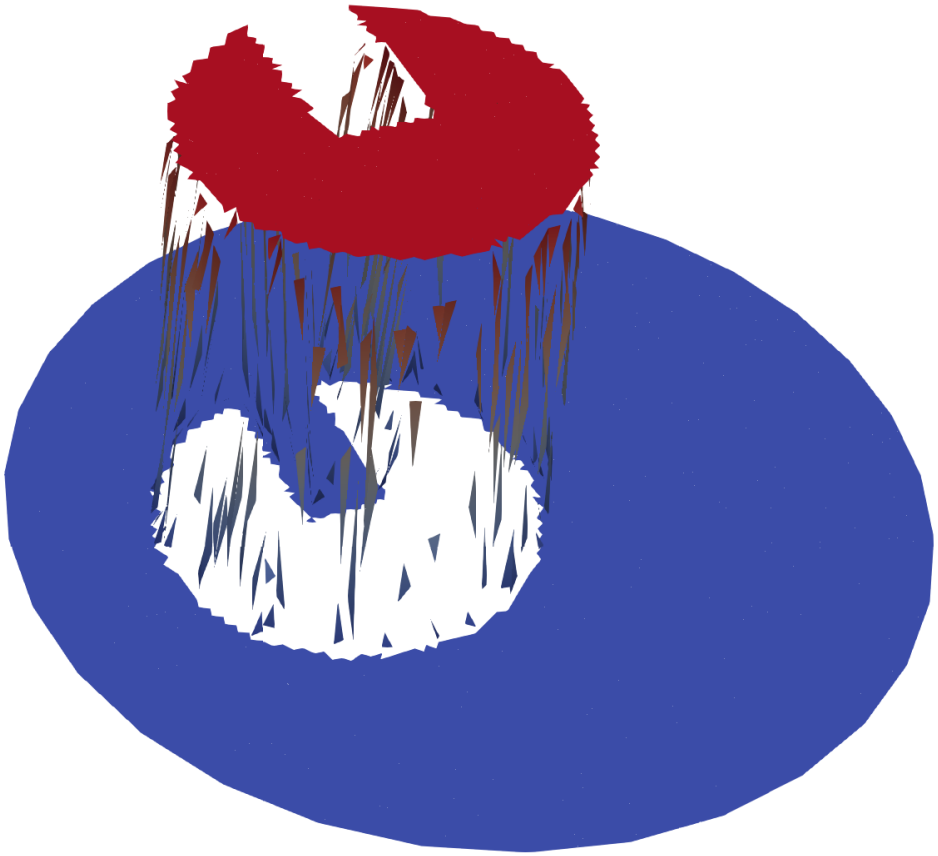
The ufl-forms for the different contributions in this algebraic problem can be obtained with the `FormsPDEMap` class. Furthermore, we feed these forms into the `PDEStaticCondensation` class that will be used for the acutal projection:

Note that in the snippet above, the ζ parameter which penalizes over and undershoot is set to a value of 30. Other than scaling this parameter with the approximate number of particles per cell, there is (as yet) not much more intelligence behind it).

We are almost ready for running the advection problem, the only thing which we need is an initial condition for ψ_h on the mesh. In order to obtain this mesh field from the particles, the bounded ℓ^2 projection that is available in `leopart` is used, i.e.

```
lstsq_psi = l2projection(p, W, 1)

lstsq_psi.project(psi_h0, lb, ub)
assign(psi_h00, psi_h0)
```

This results in the initial mesh field as shown below:



Now we are ready to enter the time loop and solve the PDE-constrained projection in every time step for reconstructing a conservative mesh field ψ_h from the moving particle field.

```
step = 0
t = 0.0
area_0 = assemble(psi_h0 * dx)

while step < num_steps:
    step += 1
    t += float(dt)

    if comm.Get_rank() == 0:
        print(f"Step {step}")
```

(continues on next page)

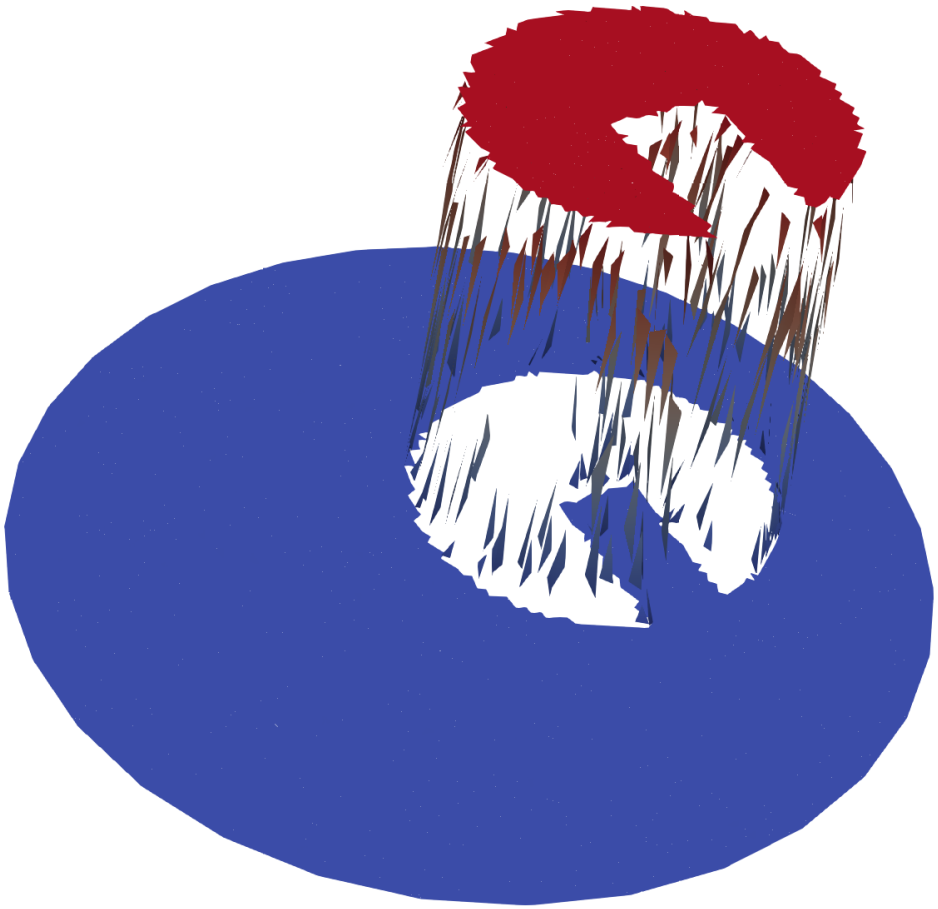
(continued from previous page)

```
    ap.do_step(float(dt))

    pde_projection.assemble(True, True)
    pde_projection.solve_problem(psibar_h, psi_h, "mumps", "default")

    assign(psi_h0, psi_h)
```

Finally, we want to check if we indeed can keep our promise of being conservative and accurate, so let's check by printing:

```
area_end = assemble(psi_h * dx)
num_part = p.number_of_particles()
l2_error = np.sqrt(abs(assemble((psi_h00 - psi_h) * (psi_h00 - psi_h) * dx)))
if comm.Get_rank() == 0:
    print(f"Num particles {num_part}")
    print(f"Area error {abs(area_end - area_0)}")
    print(f"L2-Error {l2_error}")
```

That's all there is! The code for running this example can be found on Bitbucket. Just to convince you that it works, this is the reconstructed mesh field at $t = 1$, after a half rotation:



2.3 Reference

2.3.1 Particle Generators

class ParticleGenerator.**RandomRectangle**(*ll*, *ur*)

Overloads the RandomGenerator class for generating random particle locations within a rectangular object.

__init__(*ll*, *ur*)

Initialize Random Rectangle object.

Parameters

- **ll** (*dolfin.Point*) – Point containing lower-left x and y coordinate of rectangle.

- **ur** (*dolfin.Point*) – Point containing upper-right x and y coordinate of rectangle.

generate(*N*, *method='full'*)
Generate points

Parameters

- **N** (*int*) – Number of points to generate.
- **method** (*str, optional*) – Method that is used for generating the random point locations either "full" or "tensor". Defaults to "full"

Returns Numpy array of generated points.

Return type np.array

class ParticleGenerator.**RandomCircle**(*center*, *radius*)
Overloads the RandomGenerator class for generating random particle locations within a rectangular object.

__init__(*center*, *radius*)
Initialize RandomCircle class

Parameters

- **center** (*Point, list, np.ndarray*) – Center coordinates
- **radius** (*float*) – Radius of circle

generate(*N*, *method='full'*)
Generate points

Parameters

- **N** (*int*) – Number of points to generate.
- **method** (*str, optional*) – Method that is used for generating the random point locations either "full" or "tensor". Defaults to "full"

Returns Numpy array of generated points.

Return type np.array

class ParticleGenerator.**RandomBox**(*ll*, *ur*)
Overloads the RandomGenerator class for generating random particle locations within a box-shaped object.

__init__(*ll*, *ur*)
Initialize RandomBox object.

Parameters

- **ll** (*dolfin.Point*) – Lower left coordinate of box
- **ur** (*dolfin.Point*) – Upper left coordinate of box

generate(*N*, *method='full'*)
Generate points

Parameters

- **N** (*int*) – Number of points to generate.
- **method** (*str, optional*) – Method that is used for generating the random point locations either "full" or "tensor". Defaults to "full"

Returns Numpy array of generated points.

Return type np.array

class ParticleGenerator.**RandomSphere**(*center*, *radius*)
Overloads the RandomGenerator class for generating random particle locations within a sphere.

__init__(*center*, *radius*)
Initialize RandomSphere object.

Parameters

- **center** (*dolfin.Point*) – Center coordinates of sphere.
- **radius** (*float*) – Radius of sphere

generate(*N*, *method='full'*)
Generate points

Parameters

- **N** (*int*) – Number of points to generate.
- **method** (*str, optional*) – Method that is used for generating the random point locations either "full" or "tensor". Defaults to "full"

Returns Numpy array of generated points.

Return type np.array

class ParticleGenerator.**RegularRectangle**(*ll*, *ur*)
Class for generating points on a regular lattice in a rectangle

__init__(*ll*, *ur*)
Initialize RegularRectangle object.

Parameters

- **ll** (*dolfin.Point*) – Lower left corner of rectangle
- **ur** (*dolfin.Point*) – Upper right corner of rectangle

generate(*N*, *method='open'*)
Generate points on regular lattice in rectangle.

Parameters

- **N** (*list*) – Number of points to generate in each dimension
- **method** (*str, optional*) – Which method to use. Either "open" [endpoints not included], "closed" [endpoints included] or "half open"

Returns Numpy array with coordinates

Return type np.ndarray

class ParticleGenerator.**RegularBox**(*ll*, *ur*)
Class for generating points on a regular lattice in a box

__init__(*ll*, *ur*)
Initialize RegularBox instance.

Parameters

- **ll** (*dolfin.Point*) – Lower left coordinate of regular box.
- **ur** (*dolfin.Point*) – Upper right coordinate of regular box.

generate(*N*, *method='open'*)
Generate points on regular lattice in box.

Parameters

- **N** (*list*) – Number of points to generate in each dimension
- **method** (*str, optional*) – Which method to use. Either "open" [endpoints not included], "closed" [endpoints included] or "half open"

Returns Numpy array with coordinates

Return type np.ndarray

class ParticleGenerator.**RandomCell**(*mesh*)

Generate random particle locations within a dolfin.cell (as yet, only simplicial meshes supported).

__init__(*mesh*)

Initialize RandomCell generator

Parameters **mesh** (*dolfin.Mesh*) – Mesh on which to generate particles.

generate(*N*)

Generate a random set of N points per cell.

Parameters **N** (*int*) – Number of points per cell.

Returns Coordinate array of points.

Return type np.ndarray

2.3.2 Particles

class ParticleFun.**particles**(*xp*, *particle_properties*, *mesh*)

Python interface to cpp::particles.h

__init__(*xp*, *particle_properties*, *mesh*)

Initialize particles.

Parameters

- **xp** (*np.ndarray*) – Particle coordinates
- **particle_properties** (*list*) – List of np.ndarrays with particle properties.
- **mesh** (*dolfin.Mesh*) – The mesh on which the particles will be generated.

interpolate(**args*)

Interpolate field to particles. Example usage for updating the first property of particles. Note that first slot is always reserved for particle coordinates!

```
p.interpolate(psi_h , 1)
```

Parameters

- **psi_h** (*dolfin.Function*) – Function which is used to interpolate
- **idx** (*int*) – Integer value indicating which particle property should be updated.

increment(**args*)

Increment particle at particle slot by an incrementatl change in the field, much like the FLIP approach proposed by Brackbill

The code to update a property psi_p at the first slot with a weighted increment from the current time step and an increment from the previous time step, can for example be implemented as:

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

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

Parameters

- **psih_new** (*dolfin.Function*) – Function at new timestep
- **psih_old** (*dolfin.Function*) – Function at old time step
- **slots** (*list*) – Which particle slots to use? list[0] is always the quantity that will be updated
- **theta** (*float, optional*) – Use weighted update from current increment and previous increment/ theta = 1: only use current increment theta = 0.5: average of previous increment and current increment
- **step** (*int*) – Which step are you at? The theta=0.5 increment only works from step >=2

return_property(*mesh*, *index*)

Return particle property by index.

FIXME: mesh input argument seems redundant.

Parameters

- **mesh** (*dolfin.Mesh*) – Mesh
- **index** (*int*) – Integer index indicating which particle property should be returned.

Returns Numpy array which stores the particle property.

Return type np.array

number_of_particles()

Get total number of particles

Returns Global number of particles

Return type int

2.3.3 Forms PDE-constrained projection

class FormsPDEMap.**FormsPDEMap**(*mesh*, *FuncSpace_dict*, *beta_map=<sphinx.ext.autodoc.importer._MockObject object>*, *ds=<sphinx.ext.autodoc.importer._MockObject object>*)

” Class for defining the forms related to the PDE-constrained projection

Attributes:

W

Function space for the local unknown

Type dolfin.FunctionSpace

T

FunctionSpace for the Lagrange multiplier space

Type dolfin.FunctionSpace

Wbar

Function space for the control variable

Type dolfin.FunctionSpace

n
Symbolic facet normal for mesh

Type dolfin.FacetNormal

beta_map
Penalty/Regularizatio term to establish coupling between local unknown and control

Type dolfin.Constant

ds
ds Measure of mesh

Type dolfin.Measure

gdim
Geometric dimension of mesh

Type int

__init__(*mesh*, *FuncSpace_dict*, *beta_map=<sphinx.ext.autodoc.importer._MockObject object>*, *ds=<sphinx.ext.autodoc.importer._MockObject object>*)
Instantiate FormsPDEMap

Parameters

- **mesh** (*dolfin.Mesh*) – Dolfin Mesh
- **FuncSpace_dict** (*dict*) –

  Dictionary containing the function space definitions. Following keys are required:

  - FuncSpace_local: function space for local variable
  - FuncSpace_lambda: function space for Lagrange multiplier
  - FuncSpace_bar: function space for control variable
- **beta_map** (*dolfin.Constant, optional*) – Penalty/Regularizatio term to establish coupling between local unknown and control. Defaults to Constant(1e-6)
- **ds** (*dolfin.Measure, optional*) – ds Measure of mesh

forms_theta_linear(*psih0*, *uh*, *dt*, *theta_map*, *theta_L=<sphinx.ext.autodoc.importer._MockObject object>*, *dpsi0=<sphinx.ext.autodoc.importer._MockObject object>*, *dpsi00=<sphinx.ext.autodoc.importer._MockObject object>*, *h=<sphinx.ext.autodoc.importer._MockObject object>*, *neumann_idx=99*, *zeta=<sphinx.ext.autodoc.importer._MockObject object>*)
Set PDEMap forms for a linear advection problem.

Parameters

- **psih0** (*dolfin.Function*) – dolfin Function storing the solution from the previous step
- **uh** (*Constant, Expression, dolfin.Function*) – Advective velocity
- **dt** (*Constant*) – Time step value
- **theta_map** (*Constant*) – Theta value for time stepping in PDE-projection according to theta-method **NOTE** theta only affects solution for Lagrange multiplier space polynomial order >= 1
- **theta_L** (*Constant, optional*) – Theta value for reconstructing intermediate field from the previous solution and old increments. Defaults to Constan(1.)

- **dpsi0** (*dolfin.Function, optional*) – Increment function from last time step. Defaults to Constant(0)
- **dpsi00** (*dolfin.Function*) – Increment function from second last time step. Defaults to Constant(0)
- **h** (*Constant, dolfin.Function, optional*) – Expression or Function for non-homogenous Neumann BC. Defaults to Constant(0.)
- **neumann_idx** (*int, optional*) – Integer to use for marking Neumann boundaries. Defaults to value 99
- **zeta** (*Constant, optional*) – Penalty parameter for limiting over/undershoot. Defaults to 0

Returns Dictionary with forms

Return type dict

forms_theta_nlinear(*v0*, *Ubar0*, *dt*, *theta_map=<sphinx.ext.autodoc.importer._MockObject object>*, *theta_L=<sphinx.ext.autodoc.importer._MockObject object>*, *duh0=<sphinx.ext.autodoc.importer._MockObject object>*, *duh00=<sphinx.ext.autodoc.importer._MockObject object>*, *h=<sphinx.ext.autodoc.importer._MockObject object>*, *neumann_idx=99*)

Set PDEMap forms for a non-linear (but linearized) advection problem,

Parameters

- **v0** (*dolfin.Function*) – dolfin.Function storing solution from previous step
- **Ubar0** (*dolfin.Function*) – Advective velocity at facets
- **dt** (*Constant*) – Time step
- **theta_map** (*Constant, optional*) – Theta value for time stepping in PDE-projection according to theta-method. Defaults to Constant(1.) **NOTE** theta only affects solution for Lagrange multiplier space polynomial order >= 1
- **theta_L** (*Constant, optional*) – Theta value for reconstructing intermediate field from the previous solution and old increments. Defaults to Constant(1.)
- **duh0** (*dolfin.Function, optional*) – Increment function from last time step
- **duh00** (*dolfin.Function, optional*) – Increment function from second last time step
- **h** (*Constant, dolfin.Function, optional*) – Expression or Function for non-homogenous Neumann BC. Defaults to Constant(0.)
- **neumann_idx** (*int, optional*) – Integer to use for marking Neumann boundaries. Defaults to value 99

Returns Dictionary with forms

Return type dict

forms_theta_nlinear_np(*v0*, *v_int*, *Ubar0*, *dt*, *theta_map=<sphinx.ext.autodoc.importer._MockObject object>*, *theta_L=<sphinx.ext.autodoc.importer._MockObject object>*, *duh0=<sphinx.ext.autodoc.importer._MockObject object>*, *duh00=<sphinx.ext.autodoc.importer._MockObject object>*, *h=<sphinx.ext.autodoc.importer._MockObject object>*, *neumann_idx=99*)

Set PDEMap forms for a non-linear (but linearized) advection problem, assumes however that the mass matrix can be obtained from the mesh (and not from particles)

NOTE Documentation upcoming.

forms_theta_nlinear_multiphase(*rho*, *rho0*, *rho00*, *rhobar*, *v0*, *Ubar0*, *dt*, *theta_map*, *theta_L=<sphinx.ext.autodoc.importer._MockObject object>*, *duh0=<sphinx.ext.autodoc.importer._MockObject object>*, *duh00=<sphinx.ext.autodoc.importer._MockObject object>*, *h=<sphinx.ext.autodoc.importer._MockObject object>*, *neumann_idx=99*)

Set PDEMap forms for a non-linear (but linearized) advection problem including density.

Parameters

- **rho** (*dolfin.Function*) – Current density field
- **rho0** (*dolfin.Function*) – Density field at previous time step.
- **rho00** (*dolfin.Function*) – Density field at second last time step
- **rhobar** (*dolfin.Function*) – Density field at facets
- **v0** (*dolfin.Function*) – Specific momentum at old time level
- **Ubar0** (*dolfin.Function*) – Advective field at old time level
- **dt** (*Constant*) – Time step
- **theta_map** (*Constant*) – Theta value for time stepping in PDE-projection according to theta-method **NOTE** theta only affects solution for Lagrange multiplier space polynomial order >= 1
- **theta_L** (*Constant, optional*) – Theta value for reconstructing intermediate field from the previous solution and old increments.
- **duh0** (*dolfin.Function, optional*) – Increment from previous time step.
- **duh00** (*dolfin.Function, optional*) – Increment from second last time step
- **h** (*Constant, dolfin.Function, optional*) – Expression or Function for non-homogenous Neumann BC. Defaults to Constant(0.
- **neumann_idx** (*int, optional*) – Integer to use for marking Neumann boundaries. Defaults to value 99

Returns Dict with forms

Return type dict

facet_integral(*integrand*)

Facet integral of mesh

Parameters **integrand** (*UFL*) –

Returns

Return type UFL Form

2.3.4 Forms HDG Stokes

class FormsStokes.**FormsStokes**(*mesh*, *FuncSpaces_L*, *FuncSpaces_G*, *alpha*, *beta_stab=<sphinx.ext.autodoc.importer._MockObject object>*, *ds=<sphinx.ext.autodoc.importer._MockObject object>*)

Initializes the forms for the unsteady Stokes problem following Labeur and Wells (2012) and Rhebergen and Wells (2016,2017).

Note that we can easiliy change between the two formulations since pressure stabilization term required for Labeur and Wells (2012) formulation is supported.

It defines the forms in correspondence with following algebraic form:

```
|  A   B   C   D  | | Uh    |
|  B^T F   0   H  | | Ph    |     |Q|
|                 | |       | =   | |
|  C^T 0   K   L  | | Uhbar |     |S|
|  D^T H^T L^T P  | | Phbar |
```

With part above blank line indicating the contributions from local momentum- and local mass conservation statement respectively. Part below blank line indicates the contribution from global momentum and global mass conservation statement.

__init__(*mesh*, *FuncSpaces_L*, *FuncSpaces_G*, *alpha*, *beta_stab=<sphinx.ext.autodoc.importer._MockObject object>*, *ds=<sphinx.ext.autodoc.importer._MockObject object>*)

Initialize self. See help(type(self)) for accurate signature.

forms_steady(*nu*, *f*)

Steady Stokes

forms_unsteady(*ustar*, *dt*, *nu*, *f*)

Forms for Backward-Euler time integration

forms_multiphase(*rho*, *ustar*, *dt*, *mu*, *f*)

Forms for Backward-Euler time integration two-fluid formulation Stokes

Index

Symbols

B

D

F

G

I

N

P

R

T

W