Proteus Documentation

The Proteus Team

Dec 18, 2019

GENERAL

Proteus is a Python package for rapidly developing computer models and numerical methods. It is focused on models of continuum mechanical processes described by partial differential equations and on discretizations and solvers for computing approximate solutions to these equations. Proteus consists of a collection of Python modules and scripts. Proteus also uses several C, C++, and Fortran libraries, which are either external open source packages or part of Proteus, and several open source Python packages.

The design of Proteus is organized around two goals:

- Make it easy to solve new model equations with existing numerical methods
- Make it easy to solve existing model equations with new numerical methods

We want to improve the development process for models *and* methods. Proteus is not intended to be an expert system for solving partial differential equations. In fact, effective numerical methods are often physics-based. Nevertheless many physical models are mathematically represented by the same small set of differential operators, and effective numerical methods can be developed with minor adjustments to existing methods. The problem with much existing software is that the physics and numerics are completely intertwined, which makes it difficult to extend (and maintain). In Proteus the description of the physical model and initial-boundary value problems are nearly "method agnostic". This approach has been used in the developement of a variety of mathematical models and numerical methods, both of which are described in more detail below (*Capabilities*).

CHAPTER

ONE

OBTAINING AND INSTALLING PROTEUS

For learning and experimenting there is an Docker image.

Proteus is available as source from our public GitHub repository. If you already have compilers (C,C++, and Fortran!) and Git installed on your system, you can install Proteus with the following commands.:

```
% git clone https://github.com/erdc/proteus
% cd proteus
% make develop
% make test
```

More information is available on our Wiki, and you can ask for help on the Developers' Mailing List.

CHAPTER

TWO

RUNNING

If you have successfully compiled and tested Proteus then you should be able to do:

```
% cd $PROTEUS/tests/ci
% $PROTEUS_PREFIX/bin/parun poisson_3d_p.py poisson_3d_c0p1_n.py
```

The solution will be saved in a file ending in .xmf, which can be opened with ParaView or Ensight.

CHAPTER

THREE

CAPABILITIES

Test problems and some analytical solutions have been implemented for

- Poisson's equation
- The heat equation
- Linear advection-diffusion-reaction equations
- Singly degenerate nonlinear advection-diffusion-reaction equations (including various forms of Burger's equation)
- Doubly degenerate nonlinear advection-diffusion-reaction equations
- The eikonal (signed distance) equation
- The diffusive wave equations for overland flow
- 1D and 2D Shallow Water Equations
- 2D Dispersive Shallow Water Equations
- Richards' equation (mass conservative head- and saturation-based)
- Two-phase flow in porous media with diffuse interface (fully coupled and IMPES formulations)
- Two-phase flow in porous media with a sharp interface (level set formulation)
- Stokes equations
- Navier-Stokes equations
- Reynolds-Averged Navier-Stokes equations
- Two-phase Stokes/Navier-Stokes/RANS flow with a sharp interface (level set/VOF formulation)
- Linear elasticity

These problems are solved on unstructured simplicial meshes. Simple meshes can be generated with tools included with Proteus, and more complex meshes can by imported from other mesh generators. The finite elements implemented are

Classical methods with various types of stabilization (entropy viscosity, variational multiscale, and algebraic methods)

- C_0P_1
- C_0P_2
- C_0Q_1
- C_0Q_2

Discontinuous Galerkin methods

- $C_{-1}P_0$
- $C_{-1}P_1$ (Lagrange Basis)
- $C_{-1}P_2$ (Lagrange Basis)
- $C_{-1}P_k$ (Monomial Basis)

Non-conforming and mixed methods

- P_1 non-conforming
- $C_0P_1C_0P_2$ Taylor-Hood

The time integration methods are

- Backward Euler
- Forward Euler
- Θ Methods
- Strong Stability Preserving Runge-Kutta Methods
- Adaptive BDF Methods
- Pseudo-transient continuation

The linear solvers are

- Jacobi
- Gauss-Seidel
- Alternating Schwarz
- Full Multigrid
- Wrappers to LAPACK, SuperLU, and PETSc

The nonlinear solvers are

- Jacobi
- Gauss-Seidel
- Alternating Schwarz
- Newton's method
- Nonlinear Multigrid (Full Approximation Scheme)
- Fast Marching and Fast Sweeping

Additional tools are included for pre- and post-processings meshes and solutions files generated by Proteus and other models, including methods for obtaining locally-conservative velocity fields from C_0 finite elements.

CHAPTER

FOUR

RELEASE POLICY

The releases are numbered major.minor.revision

- A revision increment indicates a bug fix or extension that shouldn't break any models working with the same major.minor number.
- A minor increment introduces significant new functionality but is mostly backward compatible
- A major increment may require changes to input files and significantly change the underlying Proteus implementation.

These are not hard and fast rules, and there is no time table for releases.

CHAPTER

FIVE

REFERENCES

- Robust explicit relaxation technique for solving the Green-Naghdi equations (2019) J.-L. Guermond, B. Popov, E. Tovar, C.E. Kees, *Journal of Computational Physics*
- An Unstructured Finite Element Model for Incompressible Two-Phase Flow Based on a Monolithic Conservative Level Set Method (2019) M. Quezada de Luna, J. H. Collins, and C.E. Kees.
- Preconditioners for Two-Phase Incompressible Navier-Stokes Flow (2019) N. Bootland, C.E. Kees, A. Wathen, A. Bentley *SIAM Journal on Scientific Computing*, In Press.
- Modeling Sediment Transport in Three-Phase Surface Water Systems (2019) C.T. Miller, W.G. Gray, C.E. Kees, I.V. Rybak, B.J. Shepherd, *Journal of Hydraulic Engineering*
- Fast Random Wave Generation in Numerical Tanks (2019) A. Dimakopoulos, T. de Lataillade, C.E. Kees, *Proceedings of the Institution of Civil Engineers - Engineering and Computational Mechanics*, 1-29.
- A Partition of Unity Approach to Adaptivity and Limiting in Continuous Finite Element Methods (2019) D. Kuzmin, M. Quezada de Luna, C.E. Kees, *Computers and Mathematics with Applications*.
- Simulating Oscillatory and Sliding Displacements of Caisson Breakwaters Using a Coupled Approach (2019) G. Cozzuto, A. Dimakopoulos, T. de Lataillade, P.O. Morillas, and C.E. Kees, *Journal of Waterway, Port, Coastal, and Ocean Engineering*.
- A Monolithic Conservative Level Set Method with Built-In Redistancing (2019) M. Quezada de Luna, D. Kuzmin, C.E. Kees, *Journal of Computational Physics*, 379, 262-278.
- Computational Model for Wave Attenuation by Flexible Vegetation (2018) S.A. Mattis, C.E. Kees, M.V. Wei, A. Dimakopoulos, and C.N. Dawson, *Journal of Waterway, Port, Coastal, and Ocean Engineering* 145(1), p.04018033.
- Well-Balanced Second-Order Finite Element Approximation of the Shallow Water Equations with Friction (2018) J.L. Guermond, M.Q. de Luna, B. Popov, C.E. Kees, and M.W. Farthing *SIAM Journal on Scientific Computing* 40(6), A3873-A3901.
- Dual-Scale Galerkin Methods for Darcy Flow (2018) G. Wang, G. Scovazzi, L. Nouveau, C.E. Kees, Simone Rossi, O. Colomes, and A. Main (2018) *Journal of Computational Physics* 354, 111-134.
- Implementation details of the level set two-phase Navier-Stokes equations in Proteus (2017) A. Bentley, N. Bootland, A. Wathen, C. Kees, *Technical-Report-TR2017-10-ab.nb.aw.ck*.
- Evaluation of Galerkin and Petrov-Galerkin Model Reduction for Finite Element Approximations of the Shallow Water Equations (2017) A. Lozovsky, M. W. Farthing, and C.E. Kees, *Computational Methods in Applied Mechanics and Engineering* 318, 537-571.
- POD-Based Model Reduction for Stabilized Finite Element Approximations of Shallow Water Flows (2016) A. Lozovskiy, M.W. Farthing, C.E. Kees, E. Gildin *Journal of Computational and Applied Mathematics*, 302, 50-70.

- An Immersed Structure Approach for Fluid-Vegetation Interaction (2015) S.A. Mattis, C.N. Dawson, C.E. Kees, M.W. Farthing, *Advances in Water Resources*, 80,1-16.
- Finite Element Methods for Variable Density Flow And Solute Transport (2013) T.J. Povich, C.N. Dawson, M.W. Farthing, C.E. Kees *Computational Geosciences* 17(3), 529-549.
- Numerical simulation of water resources problems: Models, methods, and trends (2013) Cass T. Miller, Clint N. Dawson, Matthew W. Farthing, Thomas Y. Hou, Jingfang Huang, Christopher E. Kees, C.T. Kelley, and Hans Petter Langtangen *Advances in Water Resources*, 51, 405-437,
- Numerical modeling of drag for flow through vegetated domains and porous structures (2012) S.A. Mattis, C. N. Dawson, C. E. Kees, and M. W. Farthing, *Advances in Water Resources*, 39, pp44-59
- Parallel Computational Methods and Simulation for Coastal and Hydraulic Applications Using the Proteus Toolkit (2011) C. E. Kees and M. W. Farthing (2011) *Supercomputing11: Proceedings of the PyHPC11 Workshop*
- A Conservative Level Set Method for Variable-Order Approximations and Unstructured Meshes (2011) C.E. Kees, I. Akkerman, Y. Bazilevs, and M. W. Farthing *Journal of Computational Physics* 230(12), pp4536–4558
- Locally Conservative, Stabilized Finite Element Methods For Variably Saturated Flow (2008) Kees, C.E., M. W. Farthing, and C. N. Dawson, *Computer Methods in Applied Mechanics and Engineering*, 197, pp4610-4625
- Locally Conservative, Stabilized Finite Element Methods for a Class of Variable Coefficient Navier-Stokes Equations (2009) C. E. Kees, M. W. Farthing, and M. T. Fong, *ERDC/CHL TR-09-12*
- Evaluating Finite Element Methods for the Level Set Equation (2009) M. W. Farthing and C. E. Kees, *ERDC/CHL TR-09-11*
- A Review of Methods for Moving Boundary Problems (2009) C. E. Kees, M. W. Farthing, T. C. Lackey, and R. C. Berger, *ERDC/CHL TR-09-10*
- Implementation of Discontinuous Galerkin Methods for the Level Set Equation on Unstructured Meshes (2008) M. W. Farthing and C. E. Kees, *ERDC/CHL CHETN-XIII-2*

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

SEVEN

SOURCE CODE DOCUMENTATION

7.1 BoundaryConditions

source: `proteus.BoundaryConditions` and `proteus.mprans.BoundaryConditions`

7.1.1 Usage

This module offers a framework to deal with boundary conditions that can either be set manually or by using predefined functions that have been optimised through cython/C++ code.

Import

The base *BoundaryConditions* module can be imported as follows:

```
from proteus import BoundaryConditions as bc
```

The *mprans* version of *BoundaryConditions*, adding functionality specific to multi-phase flows, can be imported as follows:

```
from proteus.mprans import BoundaryConditions as bc
```

Note: The base `proteus.BoundaryConditions` only initialises empty `proteus.BoundaryConditions.BC_Base` instances. In practice, `proteus.mprans.BoundaryConditions` is always the module imported for multi-phase flow applications, with pre-populated `proteus.mprans.BoundaryConditions.BC_RANS` that include the most common BCs used in Proteus. It is also usually used through *SpatialTools* without being directly imported.

7.2 SpatialTools

source: `proteus.SpatialTools` and `proteus.mprans.SpatialTools`

7.2.1 Usage

Using spatial tools

1. Create a domain from `proteus.Domain`.
2. Create geometries that will be part of this domain from `proteus.SpatialTools` or `proteus.mprans.SpatialTools`.
3. Assemble domain with `proteus.SpatialTools.assembleDomain()` or `proteus.mprans.SpatialTools.assembleDomain()`.

Import

The base *SpatialTools* module can be imported as follows:

```
from proteus import SpatialTools as st
```

The *mprans* version of *SpatialTools*, adding functionality specific to multi-phase flows, can be imported as follows:

```
from proteus.mprans import SpatialTools as st
```

Choosing the domain

The domain class instance is what will hold all geometrical information once it is assembled, and must be passed as an argument to all shapes created from SpatialTools. Usually, the following class types are used from *proteus.Domain*:

- 2D: `PlanarStraightLineGraphDomain`
- 3D: `PiecewiseLinearComplexDomain`

Those classes should be instantiated with no argument, e.g.:

```
from proteus import Domain
domain = Domain.PlanarStraightLineGraphDomain().
```

Assembling the domain

A very important final step not to forget is to assemble the domain once all the desired geometries have been defined, so that the `Domain` instance from *proteus.Domain* holds all the relevant geometrical information necessary for running the simulation.

```
st.assembleDomain(domain)
```

BoundaryConditions with SpatialTools

Usage

The mprans version of SpatialTools also associate geometry boundaries with *BoundaryCondition* instances from mprans.BoundaryConditions. These *BoundaryCondition* instances are easily accessible through a dictionary with predefined names, e.g. for a *Rectangle* shape instance named *my_rectangle*, `my_rectangle.BC['x-']` is for accessing the boundary conditions of the left segment, `'x+'` for the right, `'y+'` for the top, and `'y-'` for the bottom.

Refer to *BoundaryConditions* documentation for more information about the BoundaryConditions module.

Note: The boundary conditions associated with the geometry do not have to be modified/set before assembling the domain, but removing/adding boundary conditions to a geometry on top of the predefined ones must be done before.

Linking it to _p.py files

Warning: This does not apply to the *TwoPhaseFlow* module, which takes care of setting this automatically.

Linking the boundary conditions to the physical options _p.py files is done the following way (here for *RANS2P* boundary condition dictionnaries):

```
dirichletConditions = {0: lambda x, flag: domain.bc[flag].p_dirichlet.uOfXT,
                       1: lambda x, flag: domain.bc[flag].u_dirichlet.uOfXT,
                       2: lambda x, flag: domain.bc[flag].v_dirichlet.uOfXT,
                       3: lambda x, flag: domain.bc[flag].w_dirichlet.uOfXT}

advectiveFluxBoundaryConditions = {0: lambda x, flag: domain.bc[flag].p_advective.
↪uOfXT,
                                   1: lambda x, flag: domain.bc[flag].u_advective.
↪uOfXT,
                                   2: lambda x, flag: domain.bc[flag].v_advective.
↪uOfXT,
                                   2: lambda x, flag: domain.bc[flag].w_advective.
↪uOfXT}

diffusiveFluxBoundaryConditions = {0:{},
                                   1:{1: lambda x, flag: domain.bc[flag].u_diffusive.
↪uOfXT},
                                   2:{2: lambda x, flag: domain.bc[flag].v_diffusive.
↪uOfXT},
                                   3:{3: lambda x, flag: domain.bc[flag].w_diffusive.
↪uOfXT}}
```

This is always the same in the _p files, as long as it is pointing to the right boundary conditions (e.g. p_dirichlet for pressure dirichlet). The boundary conditions themselves can and should be manipulated externally (not from the _p.py file), such as in the file where the geometries are first defined.

7.2.2 Complete Examples

2D

```
from proteus import Domain
from proteus.mprans import SpatialTools as st

domain = Domain.PlanarStraightLineGraphDomain()

my_tank = st.Tank2D(domain=domain,
                    dim=[10.,5.])

my_rectangle = st.Rectangle(domain=domain,
                            dim=[1.,1.]
```

(continues on next page)

(continued from previous page)

```
                            coords=[5.,2.5],
                            barycenter=[5.,2.5])
my_rectangle.rotate(rot=3.14/4)
my_rectangle.translate(trans=[0.1,0.1])

st.assembleDomain(domain)

my_tank.BC['x-'].setNoSlip()
my_tank.BC['x-'].u_dirichlet.uOfXT = lambda x, t: 0.1*x
my_tank.BC['x-'].p_dirichlet.uOfXT = lambda x, t: -0.1*x
my_tank.BC['x+'].setFreeSlip()
my_tank.BC['y-'].setFreeSlip()
my_tank.BC['y+'].setAtmosphere()

my_rectangle.BC['x-'].setNoSlip()
my_rectangle.BC['x+'].setNoSlip()
my_rectangle.BC['y-'].setNoSlip()
my_rectangle.BC['y+'].setNoSlip()
```

3D

```
from proteus import Domain
from proteus.mprans import SpatialTools as st

domain = Domain.PiecewiseLinearComplexDomain()

my_tank = st.Tank3D(domain=domain,
                    dim=[10.,10.,5.])

my_cylinder = st.Cylinder(domain=domain,
                          radius=1.,
                          height=3.,
                          nPoints=20,
                          coords=[5.,5.,2.5],
                          barycenter=[5.,5.,2.5])
my_cylinder.rotate(rot=3.14/4,
                   axis=[1.,0.,0.],
                   pivot=my_cylinder.barycenter)
my_cylinder.translate(trans=[0.1,0.1,0.1])

st.assembleDomain(domain)

my_tank.BC['x-'].setNoSlip()
my_tank.BC['x-'].u_dirichlet.uOfXT = lambda x, t: 0.1*x
my_tank.BC['x-'].p_dirichlet.uOfXT = lambda x, t: -0.1*x
my_tank.BC['x+'].setFreeSlip()
my_tank.BC['y-'].setFreeSlip()
my_tank.BC['y+'].setFreeSlip()
my_tank.BC['z-'].setFreeSlip()
my_tank.BC['z+'].setFreeSlip()

my_cylinder.BC['x-'].setNoSlip()
my_cylinder.BC['x+'].setNoSlip()
my_cylinder.BC['y-'].setNoSlip()
my_cylinder.BC['y+'].setNoSlip()
```

3D with STL

```
#See complete case in  https://github.com/erdc/air-water-vv/tree/master/3d/STLShape
from proteus import Domain
from proteus.mprans import SpatialTools as st
from proteus.Profiling import logEvent as log

domain = Domain.PiecewiseLinearComplexDomain()

SG=st.ShapeSTL(domain,'Blocks.stl')
log("Boundary Tags are:" + str(SG.boundaryTags))


# All boundaries are free-slip for simplicity.
#See https://github.com/erdc/air-water-vv/tree/master/3d/STLShape for more advanced BC
↪'s
SG.BC['Top0'].setFreeSlip()
SG.BC['Wall0'].setFreeSlip()
SG.BC['Bed0'].setFreeSlip()
SG.BC['Concrete0'].setFreeSlip()
SG.BC['Inlet0'].setFreeSlip()
SG.BC['Outlet0'].setFreeSlip()
```

7.2.3 Classes

Base classes

The following classes are accessible with an import from *proteus.SpatialTools* and/or *proteus.mprans.SpatialTools*. Importing them from the mprans module adds functionality such as the possibility to set multi-phase flow boundary conditions and relaxation zones.

This is the same procedure as creating a *Domain* from scratch, with the added benefit of being able to add more shapes to the domain as separate instances and easy access to boundary conditions.

CustomShape

The most flexible type of shape, where everything is defined by the user. Any geometry can be created with this. The minimum arguments necessary for setting a custom geometry in 2D are: `domain`, `boundaryTags`, `vertices`, `vertexFlags`, `segments`, and `segmentFlags`. In 3D, the necessary arguments are: `domain`, `boundaryTags`, `vertices`, `vertexFlags`, `facets`, and `facetFlags`. For additional arguments, please refer to the source code in *proteus.SpatialTools*.

```
boundaryTags = {'my_tag1': 1,
                'my_tag2': 2,
                'my_tag3': 3}
vertices = [[0.,0.],
            [1.,0.],
            [1.,1.],
            [0.,1.]]
vertexFlags = [boundaryTags['my_tag1'],
               boundaryTags['my_tag1'],
               boundaryTags['my_tag2'],
               boundaryTags['my_tag2']]
segments = [[0, 1],
```

(continues on next page)

(continued from previous page)

```
            [1, 2],
            [2, 3],
            [3, 0]]
# flags can also be set from numbers included in the boundaryTags dictionary
segmentFlags = [1, 2, 3, 2]
my_customshape = st.CustomShape(domain=domain,
                                vertices=vertices,
                                vertexFlags=vertexFlags,
                                segments=segments,
                                segmentFlags=segmentFlags,
                                boundaryTags=boundaryTags)
my_customshape.BC['my_tag1'].setNoSlip()
```

Rectangle

A simple rectangular shape.

```
my_rectangle = st.Rectangle(domain=domain,
                            dim=[10.,2.],
                            coords=[5.,1.],
                            barycenter=[5.,1.])
```

Circle

A simple circular shape.

```
my_circle = st.Circle(domain=domain,
                      radius=5.,
                      coords=[5.,5.],
                      barycenter=[5.,5.],
                      nPoints=20)
```

Cuboid

A simple cuboidal shape.

```
my_cuboid = st.Cuboid(domain=domain,
                      dim=[10.,10.,2.],
                      coords=[5.,5.],
                      barycenter=[5.,5.])
```

Cylinder

A simple cylindrical shape.

```
my_cylinder = st.Circle(domain=domain,
                        radius=5.,
                        height=10.
                        nPoints=20,
                        coords=[5.,5.,7.5],
                        barycenter=[5.,5.,7.5])
```

Sphere

A simple spherical shape.

```
my_sphere = st.Sphere(domain=domain,
                      radius=5.,
                      coords=[2.,2.],
                      barycenter=[2.,2.],
                      nSectors=10)
```

ShapeSTL

For importing STL geometries. It needs a *.stl* ASCII file, and does not currently work with binary files. The STL geometry is converted in a Proteus readable format, automatically creating vertices and facets, and a single boundary tag/flag for the whole STL geometry.

```
my_stl = st.ShapeSTL(domain=domain,
                     filename='path/to/my/file.stl')
```

The function has the capability of reading multi-block stl ASCII files. Multiblock files can be created by concatenating multiple STL files containing a single geometry block into one file. You can do this quickly in a bash shell as follows:

```
cat file_1.stl file_2.stl file_3.stl > block.stl
```

```
solid block0
...
facet normal ni nj nk
   outer loop
     vertex v1x v1y v1z
     vertex v2x v2y v2z
     vertex v3x v3y v3z
     endloop
endfacet
...
endsolid
solid block1
...
endsolid
...
solid blockFinal
...
endsolid
```

When reading the block stl file, the `ShapeSTL` class will read also the stl blocks and, in addition to vertices and facets, tags for boundaryTags, vertices and facets will be assigned. The names of boundaries are given according to the naming of the blocks. E.g. block0 will form boundary block0 etc.

The user should be able to mesh a whole domain, as long as the STL files create a watertight domain. A simple example of setting up a 3D domain is given in the beginning of this section and the air-water-vv repository https://github.com/erdc/air-water-vv/blob/master/3d/STLShape/

mprans specific

The following classes are for use with multi-phase flow and can only be imported from *proteus.mprans.SpatialTools*.

Tank2D

The *Tank2D* class can be used to create a rectangular tank. This class allows for "sponge layers", or "relaxation zones" that are usually used for wave absorption or wave generation to get rid of reflected waves in the domain. The lower left corner of the tank is at the origin *[0.,0.]* when created (but it can still be translated later on), and sponge layers extend outwards of the numerical tank. A *Tank2D* of dimensions [10.,2.] and sponge layers of length 3. on both sides will have a total domain size of [16,2], spanning from x=-3 to x=13.

```
my_tank = st.Tank2D(domain=domain,
                    dim=[10.,2.])
# make sponge layers
my_tank.setSponge(x_n=3., x_p=3.)
# set absorption zone (x_p -> x+)
my_tank.setAbsorptionZones(dragAlpha=1.e6,
                           x_p=True)
# set generation zone
from proteus import WaveTools as wt (x_n -> x-)
my_wave = wt.MonochromaticWave()
he = 0.01
my_tank.setGenerationZones(dragAlpha=1.e6,
                           smoothing=3*he,
                           wave=wave,
                           x_n=True)
# set boundary conditions
my_tank.BC['y+'].setAtmosphere()
my_tank.BC['y-'].setFreeSlip()
my_tank.BC['x+'].setFreeSlip()
my_tank.BC['x-'].setUnsteadyTwoPhaseVelocityInlet(wave=my_wave
                                                  smoothing=3*he)
my_tank.BC['sponge'].setNonMaterial()
```

Important: *Tank2D* instances should not be rotated as this can lead to problems with relaxation zones and boundary conditions.

Tank3D

Very similar to the *Tank3D*, it is a cuboid for 3D domains with the possibility of adding sponge layers.

```
my_tank = st.Tank2D(domain=domain,
                    dim=[10.,10., 2.])
# make sponge layers
my_tank.setSponge(x_n=3., x_p=3., y_p=3., y_n=3.)
# set absorption zones
my_tank.setAbsorptionZones(dragAlpha=1.e6,
                           x_p=True,
                           y_p=True,
                           y_n=True)
# set generation zone
from proteus import WaveTools as wt (x_n -> x-)
my_wave = wt.MonochromaticWave()
he = 0.01
my_tank.setGenerationZones(dragAlpha=1.e6,
                           smoothing=3*he,
                           wave=wave,
                           x_n=True)
# set boundary conditions
my_tank.BC['z+'].setAtmosphere()
my_tank.BC['z-'].setFreeSlip()
my_tank.BC['y-'].setFreeSlip()
my_tank.BC['x+'].setFreeSlip()
my_tank.BC['x-'].setUnsteadyTwoPhaseVelocityInlet(wave=my_wave
                                                  smoothing=3*he)
my_tank.BC['sponge'].setNonMaterial()
```

Important: *Tank3D* instances should not be rotated as this can lead to problems with relaxation zones and boundary conditions.

TankWithObstacle2D

7.3 TwoPhaseFlow

7.4 WaveTools

source: `proteus.WaveTools`

7.4.1 Usage

This module offers a framework for calculating the free-surface elevation and wave velocities based on various wave theories. Wave theories are organised in classes within the module. The module is written in Python / Cython and C++ for optimising calculation speed.

Import in command line

Once installed Proteus, you can open a python or ipython command line and type

```
from proteus import WaveTools as wt
```

You can see information on the module (including available classes) by typing:

```
help(wt)
```

and you can see a list of classes and functions by typing

```
wt.__all__
```

Each function or class has documentation info which is accessible by typing

```
help(wt.function)
help(wt.class)
help(wt.class.function)
```

List of wave theories

Available classes that correspond to wave theories are

`SteadyCurrent` - Introduce steady currents with ramp time

`SolitaryWave` - Generate 1st order solitary waves

`MonochromaticWaves` - Generate linear and nonlinear monochromatic waves. Nonlinear wave theory according to Fenton's Fourier transform

`NewWave` - Generate waves according to NewWave theory Tromans et al 1991

`RandomWaves` - Generate plane random waves from JONSWAP or Pierson Moskovitch

`MultiSpectraRandomWaves` - Generate random waves by overlaying multiple frequency spectra. Wave spectra can meet in different angles

`DirectionalWaves` - Generate random waves using JOSNWAP / PM spectra for frequencies and cos-2s/Mitsuyashu spectra for directions

`TimeSeries` - Generate waves from a given free-surface time series. Time series can be reconstructed using direct or windowed methods

`RandomWavesFast` - Same as `RandomWaves` only much more computationally efficient, see Dimakopoulos et al 2019

`RandomNLWaves` - Generate plane random waves from JONSWAP or RM, using 2nd order theory, following Dalzell's formulae

`RandomNLWavesFast` - Same as `RandomNLWaves` only much more computationally efficient, by using the approach of Dimakopoulos et al 2019

`CombineWaves` - Generate waves by combining any of the wave theories above

How to use in Proteus

The wave tools module is loaded at the preample as in the case of the command line:

Then the target wave theory is set, by initializing the relevant class as follows

The wave theory is passed through the `setUnsteadyTwoPhaseVelocityInlet` boundary condition as follows:

```
tank.BC['x-'].setUnsteadyTwoPhaseVelocityInlet(wave, smoothing=smoothing, vert_axis=1)
```

If the relaxation zone method is used, then the class should be passed through the relevant `setGenerationZones` function

```
tank.setGenerationZones(x_n=True, waves=wave, dragAlpha=dragAlpha, smoothing =␣
↪smoothing)
```

Guidance for using the `setUnsteadyTwoPhaseVelocityInlet` and `setGenerationZones` functions are given in the BoundaryConditions and RelaxationZone sections of the documentation

Simple examples of usage within the context of a 2D numerical tank can be found in air-water-vv

How to use as stand-alone tool

After importing the tool in a python interface (command line, editor) following the examples above, you can load a class that corresponds to a wave theory, as follows:

```
wave = wt.RandomWavesFast(Tstart=0.,
                          Tend=5000.,
                          x0=np.array([0.,0.,0.])
                          Tp=2.5,
                          Hs=0.1,
                          mwl=0.5,
                          depth=0.5,
                          waveDir=np.array([1,0,0]),
                          g=np.array([0,-9.81,0]),
                          N=2000,
                          bandFactor=2.,
                          spectName="JONSWAP",
                          Lgen=1.,
                          Nwaves=16,
                          Nfreq=32,
                          checkAcc=True,
                          fast=True)
```

Then the free surface and velocity for a point in space and time can be calculated as follows:

```
x0 = [1.,0.,0.]
t0 = 0.
U = wave.u(x0,t0)
```

Full time series can be calculated and plotted by appropriately manipulating the calculations and storing in arrays, e.g.:

```
x0 = [1.,0.,0.]
time_array = np.linspace(0,10,1000)
eta = np.zeros(len(time_array),)
for i,t in enumerate(time_array):
     eta[i] = wave.eta(x0,t)
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
plt.plot(time_array,eta)
plt.xlabel("Time (s)")
plt.ylabel("Free-surface elevation (m)")
plt.savefig("Free-surface.pdf")
plt.show()
```

7.5 Body Dynamics

7.5.1 Implementation

The module described in this page can be imported as follows:

```
from proteus.mbd import CouplingFSI as fsi
```

Proteus uses wrappers and modified/derived classes from the Chrono engine, an open-source multibody dynamics library available at https://github.com/projectchrono/chrono.

The bodies and cables classes described below can interact with proteus models such as Navier-Stokes to retrieve forces and moments, moving (ALE) mesh for moving the domain with the structure, and added-mass model.

7.5.2 Classes

ProtChSystem

The *ProtChSystem* class has a pointer to a Chrono *ChSystem*, and holds the general options for the Chrono simulation, such as time step size, gravity, etc. All the physical bodies described must be associated to a *ProtChSystem* instance.

```
import pychono
from proteus.mbd import CouplingFSI as fsi

my_system = fsi.ProtChSystem()
g = pychrono.ChVectorD(0., 0., -9.81)
my_system.ChSystem.Set_G_acc(g)
my_system.setTimeStep(0.001)  # the time step for Chrono calculations
```

Important: The *ProtChSystem* instance itself must be added to the *auxiliaryVariables* list of the Navier-Stokes model in order to calculate and retrieve the fluid forces from the fluid pressure field provided by Proteus at the boundaries of the different bodies.

ProtChBody

Class for creating a rigid body. It has a Chrono *ChBody* body variable (*ProtChBody.ChBody*) accessible within python with some of the functionalities/functions of Chrono *ChBody*. It must be associated to a *ProtChSystem* instance in order to be included in the multibody dynamics simulation. This can be done with the passing of the *system* argument as the *ProtChBody* instance is created (see example below). Otherwise, the function *ProtChSystem.addProtChBody(my_body)* can be called separately.

```
my_body = fsi.ProtChBody(system=my_system)
my_body.attachShape(my_shape)  # sets everything automatically
my_body.setRecordValues(all_values=True) # record everything
```

When set up properly and running with a Proteus Navier-Stokes simulation, the fluid pressure will be applied on the boundaries of the rigid body. The ChBody will be moved accordingly, as well as its boundaries (supposing that a moving mesh or immersed boundaries are used).

Attention: The *ProtChBody.ChBody* variable is actually using a derived class from the base Chrono *ChBody* in order to add the possibility of using an added-mass matrix (see *ChBodyAddedMass* in proteus.mbd.ChRigidBody.h).

ProtChMesh

This class creates a *ChMesh* that is needed to create moorings.

```
my_mesh = fsi.ProtChMesh(system=my_system)
```

ProtChMoorings

This class is for easily creating cables. The following properties must be known in order to instantiate a *ProtChMoorings*: *ProtChSystem* instance, *Mesh* instance, *length* for the length of the cable/segment, *nb_elems* for the number of elements along the cable/segment, *d* for the diameter of the cable/segment, *rho* for the density of the cable/segment, *E* for the Young modulus of the cable/segment.

```
my_mooring = fsi.ProtChMoorings(system=my_system,
                                mesh=my_mesh,
                                length=np.array([10.]),
                                nb_elems=np.array([10], dtype=np.int_32),
                                d=np.array([0.01]),
                                rho=np.array([300.2]),
                                E=np.array([1e9]))
# set function to place the nodes along cable ('s' is the position along the 1D cable)
fpos = lambda s: np.array([s, 1., 0.])  # position along cable
ftan = lambda s: np.array([1., 0., 0.])  # tangent of cable along cable
my_mooring.setNodesPositionFunction(function_position=fpos,
                                    function_tangent=ftan)
# set the nodes position from the function
my_mooring.setNodesPosition()
# add a body as fairlead
my_mooring.attachBackNodeToBody(my_body)
# fix front node as anchor
my_mooring.fixFrontNode(True)
```

Setting the position function is useful when a relatively complex layout of the cable is desired, such as a catenary shape.

Note: The reason for the array structure for the *length*, *nb_elems*, *d*, *rho*, and *E* parameters is that a cable can be multi-segmented (different sections of the same cable having different material properties).

ProtChAddedMass

A class to deal with the added mass model from proteus.mprans.AddedMass. This class should not be instantiated manually and will be automatically instantiating as a variable of *ProtChSystem* (accessible as *my_system.ProtChAddedMass*). It is used to build the added mass matrix for the rigid bodies.

Important: This class instance must be passed to the *AddedMass* model *auxiliaryVariables* to have any effect (*auxiliaryVariables.append(my_system.ProtChAddedMass*)

7.5.3 Postprocessing Tools

ProtChBody

The data related to mooring cables is saved in an csv file, usually `[my_body.name].csv`. Additionally, if the added mass model was used, the values of the added mass matrix are available in `[my_body.name]_Aij_.csv`

ProtChMoorings

The data related to mooring cables is saved in an hdf5 file, usually `[my_mooring.name].h5`, which can be read directly with h5py. Another way to read and visualise the data is to use the associated `[my_mooring.name].xmf`. The following script must be first ran (note that there is no extension for the file name): .. code-block:

```
{PROTEUS_DIR}/scripts/gatherTimes.py -f [my_mooring.name]
```

where `{PROTEUS_DIR}` is the root directory of the Proteus installation. This will create `[my_mooring.name]_complete.xmf` which can be opened in Paraview to navigate the time steps that have been recorded.

7.6 Free Surface

There are two implementations for dealing with the free surface.

7.6.1 VOF - NCLS

In this implementation, the following 4 models must be solved in order:

- Volume of Fluid (VOF): `proteus.mprans.VOF`
- Non-conservative level set (NCLS): `proteus.mprans.NCLS`
- Redistancing: `proteus.mprans.RDLS`
- Mass correction: `proteus.mprans.MCorr`

7.6.2 CLSVOF

In this implementation, a conservative level set is used and only the following model must be solved: `proteus.mprans.CLSVOF`.

7.7 Mesh Adaptivity

7.8 Mesh Motion

`proteus.mprans.MoveMesh`

`proteus.mprans.MoveMeshMonitor`

7.9 Navier-Stokes

7.9.1 Description

There are currently 3 implementations of Navier-Stokes equations in proteus:

- Two-phase flow (e.g. air/water)
- Three-phase flow (e.g. air/water/sediment)
- Two-phase flow with immersed boundaries (solid)

7.9.2 Two-Phase

The two-phase implementation of Navier-Stokes, with source documentation available here: `proteus.mprans.RANS2P`.

7.9.3 Three-Phase

The three-phase implementation of Navier-Stokes, with source documentation available here: `proteus.mprans.RANS3P`.

7.9.4 Dealing with Moving Domains

When dealing with moving domains, the option `movingDomain` must be set to `True`. This is necessary to signal to the model that mesh nodes velocity is to be expected from an external model.

Moving (ALE) Mesh

In the current implementation, if a model for moving the mesh is used such as `proteus.mprans.MoveMesh`, it should be the first model to be solved, as the mesh velocity is calculated from the previous time step.

Immersed Boundaries

The immersed boundary (three-phase) implementation of Navier-Stokes, with source documentation available here: `proteus.mprans.RANS2P_IB`.

7.10 Shallow Water Flows

7.10.1 Description

There are currently 2 different models that describe shallow water flow in Proteus:

- Classical Saint-Venant equations (e.g. Shallow Water equations)
- Dispersive shallow water model based on the Green-Naghdi equations (e.g. mGN equations)

7.10.2 Shallow Water equations

The Shallow Water equations are a set of partial differential equations that form a hyperbolic system. They can be used to describe a body of water evolving under the action of gravity under the assumption that the deformations of the free surface are small compared to the water height.

The implementation of the SWEs with source documentation is available here: `proteus.mprans.SW2DCV`.

7.10.3 Modified Green-Naghdi equations

The modified Green-Naghdi equations are a set of partial differential equations that form a hyperbolic system and are an O(h) approximation to the traditional Green-Naghdi equations, where h is the mesh size. The Green-Naghdi equations are used to describe dispersive water waves.

The implementation of the mGN equations with source documentation is available here: `proteus.mprans.GN_SW2DCV`.

7.10.4 Running the tests

All tests that concern shallow water flows can be found at `proteus.SWFlows.tests`.