cbcflow Documentation

Release 1.3.0

Martin Alnaes and Oeyvind Evju

March 05, 2014

Contents:

CHAPTER 1

Installation

1.1 Quick Install

Install using git clone:

```
git clone https://bitbucket.org/simula_cbc/cbcflow.git
cd cbcflow
python setup.py install
```

Install using pip:

```
pip install git+https://bitbucket.org/simula_cbc/cbcflow.git
```

1.2 Dependencies

The installation of cbcflow requires the following environment:

- Python 2.7
- Numpy
- Scipy
- (FEniCS) 1.3.0

To install FEniCS, please refer to the FEniCS download page. cbcflow follows the same version numbering as FEniCS, so make sure you install the correct FEniCS version. Backwards compatibility is not guaranteed.

In addition, cbcflow can utlize other libraries for added functionality

- fenicstools 1.3.0 (highly recommended, tools to inspect parts of a solution)
- pytest >2.4.0 (required to run test suite)

fenicstools can be installed using pip:

```
pip install https://github.com/mikaem/fenicstools/archive/v1.3.0.zip
```

CHAPTER 2

Demos

To get started, we recommend starting with the demos. To get access to all the demos, execute the following command in a terminal window:

```
cbcflow-get-demos
```

To list and run all the demos, execute

```
cd cbcflow-demos/demo
./cbcflow_demos.py --list
./cbcflow_demos.py --run
```

If you have downloaded the development version, it is sufficient to download the demo data in the root folder of the repository:

```
cbcflow-get-data
```

If you are unfamiliar with FEniCS, please refer to the FEniCS Tutorial for the FEniCS-specifics of these demos.

Documented demos:

2.1 Flow Around A Cylinder

This tutorial demonstrate how one can use cbcflow to solve a simple problem, namely a flow around a cylinder, inducing a vortex street behind the cylinder.

The source code for this can be found in `FlowAroundCylinder.py`.

We start by importing cbcflow and dolfin:

```
from cbcflow import *
from dolfin import *
```

2.1.1 Specifying the domain

The meshes for this problem is pregenerated, and is specified at the following locations:

```
from os import path

files = [path.join(path.dirname(path.realpath(__file__)),"../../../cbcflow-data/cylinder_0.6k.xml.gz"
         path.join(path.dirname(path.realpath(__file__)),"../../../cbcflow-data/cylinder_2k.xml.gz"),
         path.join(path.dirname(path.realpath(__file__)),"../../../cbcflow-data/cylinder_8k.xml.gz"),
```

```
        path.join(path.dirname(path.realpath(__file__)),"../../../cbcflow-data/cylinder_32k.xml.gz")
        path.join(path.dirname(path.realpath(__file__)),"../../../cbcflow-data/cylinder_129k.xml.gz"
       ]
```

This requires that you have installed the demo data, as specified in *Demos*.

The domain is based on a rectangle with corners in (0,0), (0,1), (10,0) and (10,1). The cylinder is centered in (2,0.5) with radius of 0.12. The different boundaries of the domain is specified as:

```
class LeftBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and near(x[0], 0.0)

class RightBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and near(x[0], 10.0)

class Cylinder(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and (sqrt((x[0]-2.0)**2+(x[1]-0.5)**2) < 0.12+DOLFIN_EPS)

class Wall(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and (near(x[1], 0.0) or near(x[1], 1.0))
```

2.1.2 Defining a NSProblem

To define a problem class recognized by cbcflow, the class must inherit from `NSProblem`:

```
class FlowAroundCylinder(NSProblem):
```

Parameters

This class inherit from the `Parameterized` class, allowing for parameters in the class interface. We supply default parameters to the problem:

```
@classmethod
def default_params(cls):
    params = NSProblem.default_params()
    params.replace(
        # Time parameters
        T=5.0,
        dt=0.1,
        # Physical parameters
        rho=1.0,
        mu=1.0/1000.0,
        )
    params.update(
        # Spatial parameters
        refinement_level=0,
        )
    return params
```

This takes the default parameters from NSProblem and replaces some parameters common for all NSProblems. We set the end time to 5.0 with a timestep of 0.1, the density $\rho = 1.0$ and dynamic viscosity $\mu = 0.001$. In addition, we add a new parameter, refinement_level, to determine which of the previously specified mesh files to use.

Constructor

To initiate a FlowAroundCylinder-instance, we load the mesh and initialize the geometry:

```
def __init__(self, params=None):
    NSProblem.__init__(self, params)

    # Load mesh
    mesh = Mesh(files[self.params.refinement_level])

    # Create boundary markers
    facet_domains = FacetFunction("size_t", mesh)
    facet_domains.set_all(4)
    Wall().mark(facet_domains, 0)
    Cylinder().mark(facet_domains, 0)
    LeftBoundary().mark(facet_domains, 1)
    RightBoundary().mark(facet_domains, 2)

    # Store mesh and markers
    self.initialize_geometry(mesh, facet_domains=facet_domains)
```

The first call to NSProblem.__init__ updates the default parameters with any parameters passed to the constructor as a dict or `ParamDict`. This sets params as an attribute to self. We load the mesh from a string defined in the files-list, and define its domains. Finally, we call self.initialize_geometry to attach facet_domains to the mesh, and the mesh to self.

Initial conditions

At the initial time, the fluid is set to rest, with a zero pressure gradient. These initial conditions are prescribed by

```
def initial_conditions(self, spaces, controls):
    c0 = Constant(0)
    u0 = [c0, c0]
    p0 = c0
    return (u0, p0)
```

The argument *spaces* is a `NSSpacePool` helper object used to construct and contain the common function spaces related to the Navier-Stokes solution. This is used to limit the memory consumption and simplify the interface, so that you can, for example, call spaces.DV to get the tensor valued gradient space of the velocity regardless of velocity degree.

The argument *controls* is used for adjoint problems, and can be disregarded for simple forward problems such as this.

Boundary conditions

As boundary conditions, we set no-slip conditions on the cylinder, at y=0.0 and y=1.0. At the inlet we set a uniform velocity of (1.0,0.0), and zero-pressure boundary condition at the outlet.

To determine domain to apply boundary condition, we utilize the definition of *facet_domains* from the constructor.

```
def boundary_conditions(self, spaces, u, p, t, controls):
    c0 = Constant(0)
    c1 = Constant(1)

    # Create no-slip boundary condition for velocity
    bcu0 = ([c0, c0], 0)
    bcu1 = ([c1, c0], 1)
```

```
    # Create boundary conditions for pressure
    bcp0 = (c0, 2)

    # Collect and return
    bcu = [bcu1, bcu2]
    bcp = [bcp0]
    return (bcu, bcp)
```

The way these boundary conditions are applied to the equations are determined by the scheme used to solve the equation.

2.1.3 Setting up the solver

Now that our *FlowAroundCylinder*-class is sufficiently defined, we can start thinking about solving our equations. We start by creating an instance of *FlowAroundCylinder* class:

```
problem = FlowAroundCylinder({"refinement_level": 2})
```

Note that we can pass a dict to the constructor to set, in this example, the desired refinement level of our mesh.

Selecting a scheme

Several schemes are implemented in cbcflow, but only a couple are properly tested and validated, and hence classified as *official*. Use

```
show_schemes()
```

to list all schemes available, both official and unofficial.

In our application we select a very efficient operator-splitting scheme, `IPCS_Stable`,

```
scheme = IPCS_Stable()
```

Setting up postprocessing

The postprocessing is set up to determine what we want to do with our obtained solution. We start by creating a `NSPostProcessor` to handle all the logic:

```
casedir = "results_demo_%s_%s" % (problem.shortname(), scheme.shortname())
postprocessor = NSPostProcessor({"casedir": casedir})
```

The *casedir* parameter points the postprocessor to the directory where it should save the data it is being asked to save. By default, it stores the mesh, all parameters and a *play log* in that directory.

Then, we have to choose what we want to compute from the solution. The command

```
show_fields()
```

lists all available `PPField` to compute from the solution.

In this case, we are interested in the velocity, pressure and stream function, and we wish to both plot and save these at every timestep:

```
plot_and_save = dict(plot=True, save=True)
fields = [
    Pressure(plot_and_save),
```

```
    Velocity(plot_and_save),
    StreamFunction(plot_and_save),
    ]
```

With no saveformat prescribed, the postprocessor will choose default saveformats based on the type of data. You can use

```
print PPField.default_parameters()
```

to see common parameters of these fields.

Finally, we need to add these fields to the postprocessor:

```
postprocessor.add_fields(fields)
```

Solving the problem

We now have instances of the classes `NSProblem`, `NSScheme`, and `NSPostProcessor`.

These can be combined in a general class to handle the logic between the classes, namely a `NSSolver` instance:

```
solver = NSSolver(problem, scheme, postprocessor)
```

This class has functionality to pass the solution from scheme on to the postprocessor, report progress to screen and so on. To solve the problem, simply execute

```
solver.solve()
```

2.2 Womersley flow in 3D

In this demo it is demonstrated how to handle problems with time-dependent boundary conditions and known analytical solution/reference solution. The problem is transient Womersley flow in a cylindrical pipe.

The source code can be found in `Womersley3D.py`.

We start by importing cbcflow and dolfin:

```
from cbcflow import *
from dolfin import *
```

2.2.1 Specifying the domain

Our domain is a cylinder of length 10.0 and radius 0.5:

```
LENGTH = 10.0
RADIUS = 0.5
```

The meshes for this has been pregenerated and is available in the demo data, see *Demos*.

```
files = [path.join(path.dirname(path.realpath(__file__)),"../../../cbcflow-data/pipe_1k.xml.gz"),
         path.join(path.dirname(path.realpath(__file__)),"../../../cbcflow-data/pipe_3k.xml.gz"),
         path.join(path.dirname(path.realpath(__file__)),"../../../cbcflow-data/pipe_24k.xml.gz"),
         path.join(path.dirname(path.realpath(__file__)),"../../../cbcflow-data/pipe_203k.xml.gz"),
         path.join(path.dirname(path.realpath(__file__)),"../../../cbcflow-data/pipe_1611k.xml.gz"),
        ]
```

We define *SubDomain* classes for inflow and outflow:

```
class Inflow(SubDomain):
    def inside(self, x, on_boundary):
        return x[0] < 1e-6 and on_boundary

class Outflow(SubDomain):
    def inside(self, x, on_boundary):
        return x[0] > LENGTH-1e-6 and on_boundary
```

We could also add a *SubDomain*-class for the remaining wall, but this will be handled later.

2.2.2 Defining the NSProblem

We first define a problem class inheriting from `NSProblem`:

```
class Womersley3D(NSProblem):
```

The parameters of the problem are defined to give a Reynolds number of about 30 and Womersley number of about 60.

```
@classmethod
def default_params(cls):
    params = NSProblem.default_params()
    params.replace(
        # Time parameters
        T=None,
        dt=1e-3,
        period=0.8,
        num_periods=1.0,
        # Physical parameters
        rho=1.0,
        mu=1.0/30.0,
        )
    params.update(
        # Spatial parameters
        refinement_level=0,
        # Analytical solution parameters
        Q=1.0,
        )
    return params
```

In the constructor, we load the mesh from file and mark the boundary domains relating to inflow, outflow and wall in a *FacetFunction*:

```
def __init__(self, params=None):
    NSProblem.__init__(self, params)

    # Load mesh
    mesh = Mesh(files[self.params.refinement_level])

    # We know that the mesh contains markers with these id values
    self.wall_boundary_id = 0
    self.left_boundary_id = 1
    self.right_boundary_id = 2

    facet_domains = FacetFunction("size_t", mesh)
    facet_domains.set_all(3)
```

```
    DomainBoundary().mark(facet_domains, self.wall_boundary_id)
    Inflow().mark(facet_domains, self.left_boundary_id)
    Outflow().mark(facet_domains, self.right_boundary_id)
```

We then define a transient profile for the flow rate, for use later:

```
 # Setup analytical solution constants
Q = self.params.Q
self.nu = self.params.mu / self.params.rho

# Beta is the Poiseuille pressure drop if the flow rate is stationary Q
self.beta = 4.0 * self.nu * Q / (pi * RADIUS**4)

# Setup transient flow rate coefficients
print "Using transient bcs."
P = self.params.period
tvalues = np.linspace(0.0, P)
Qfloor, Qpeak = -0.2, 1.0
Qvalues = Q * (Qfloor + (Qpeak-Qfloor)*np.sin(pi*((P-tvalues)/P)**2)**2)
self.Q_coeffs = zip(tvalues, Qvalues)
```

Finally, we store the mesh and facet domains to *self*:

```
# Store mesh and markers
self.initialize_geometry(mesh, facet_domains=facet_domains)
```

2.2.3 The analytical solution

The Womersley profile can be obtained by using the helper function `make_womersley_bcs()`. This function returns a list of scalar *Expression* instances defining the Womersley profile:

```
def analytical_solution(self, spaces, t):
    # Create womersley objects
    ua = make_womersley_bcs(self.Q_coeffs, self.mesh, self.left_boundary_id, self.nu, None, self.face
    for uc in ua:
        uc.set_t(t)
    pa = Expression("-beta * x[0]", beta=1.0)
    pa.beta = self.beta # TODO: This is not correct unless stationary...
    return (ua, pa)
```

Note that the pressure solution defined here is not correct in the transient case.

2.2.4 Using an analytical/reference solution

If one for example wants to validate a scheme, it is required to define the following functions:

```
def test_fields(self):
    return [Velocity(), Pressure()]

def test_references(self, spaces, t):
    return self.analytical_solution(spaces, t)
```

The `test_fields()` function tells that the fields `Velocity` and `Pressure` should be compared to the results from `test_references()`, namely the analytical solution.

These functions are used in the regression/validation test suite to check and record errors.

2.2.5 Initial conditions

As initial conditions we simply use the analytical solution at t=0.0:

```
def initial_conditions(self, spaces, controls):
    return self.analytical_solution(spaces, 0.0)
```

2.2.6 Boundary conditions

At the boundaries, we also take advantage of the analytical solution, and we set no-slip conditions at the cylinder walls:

def boundary_conditions(self, spaces, u, p, t, controls): # Create no-slip bcs d = len(u) u0 = [Constant(0.0)] * d noslip = (u0, self.wall_boundary_id)

Get other bcs from analytical solution functions ua, pa = self.analytical_solution(spaces, t)

Create inflow boundary conditions for velocity inflow = (ua, self.left_boundary_id)

Create outflow boundary conditions for pressure p_outflow = (pa, self.right_boundary_id)

Return bcs in two lists bcu = [noslip, inflow] bcp = [p_outflow]

return (bcu, bcp)

Now, since these boundary conditions are transient, we need to use the `update()` function. The `boundary_conditions()` function is called at the start of solve step, and a call-back is done to the `update()` function to do any updates to for example the boundary conditions. In here, we update the time in the inlet boundary condition:

```
def update(self, spaces, u, p, t, timestep, bcs, observations, controls):
    bcu, bcp = bcs
    uin = bcu[1][0]
    for ucomp in uin:
        ucomp.set_t(t)
```

2.2.7 Solving the problem

Finally, we initate the problem, a scheme and postprocessor

```
def main():
    problem = Womersley3D({"refinement_level": 2})
    scheme = IPCS_Stable()

    casedir = "results_demo_%s_%s" % (problem.shortname(), scheme.shortname())
    plot_and_save = dict(plot=True, save=True)
    fields = [
        Pressure(plot_and_save),
        Velocity(plot_and_save),
        ]
    postproc = NSPostProcessor({"casedir": casedir})
    postproc.add_fields(fields)
```

and solves the problem

```
solver = NSSolver(problem, scheme, postproc)
solver.solve()
```

2.3 Replay a problem

This demo is based on the *Flow Around A Cylinder* demo, and demonstrates how one can compute any `PPField` from a stored solution, given that the dependencies of the field are either saved to disk or computable from the fields saved to disk.

This can be very useful when one needs additional information to the one specified at the time of the solve.

The source code for this demo can be found in `Replay.py`.

We start by importing from *Flow Around A Cylinder*:

```
import sys
# Add FlowAroundCylinder problem as example problem
sys.path.insert(0, path.join(path.dirname(path.realpath(__file__)),'../../../demo/documented/FlowArou
from FlowAroundCylinder import FlowAroundCylinder
```

We then import cbcflow

```
from cbcflow import *
```

Note that we for this demo does not need to import from dolfin.

2.3.1 Play

We start by defining the *play*-routine, that is, a normal solve. We use the `IPCS_Stable` scheme and save the results to *Results*.

```
def play():
    # First solve the problem
    problem = FlowAroundCylinder({"refinement_level": 3})
    scheme = IPCS_Stable()
```

For this we simply store the velocity at every second timestep, and the pressure at every third timestep. We plot the velocity, and we send keyword *mode=color* so the plot will show the velocity magnitude:

```
postprocessor = NSPostProcessor({"casedir": "Results"})

postprocessor.add_fields([
    Velocity({"save": True, "stride_timestep": 2, "plot": True, "plot_args": {"mode": "color"}}),
    Pressure({"save": True, "stride_timestep": 3}),
])
```

We then solve the problem:

```
solver = NSSolver(problem, scheme, postprocessor)
solver.solve()
```

2.3.2 Replay

When the solve has been performed, we might want to compute some more derived fields. We start by creating a new postprocessor instance. This instance must point to the same case directory as the original solve:

```
def replay():
    # Create postprocessor pointing to the same casedir
    postprocessor = NSPostProcessor({"casedir": "Results"})
```

We then define some new fields to store, and adds them to the postprocessor

```
# Add new fields to compute
postprocessor.add_fields([
    Stress({"save": True}),
    StreamFunction({"save": True, "plot": True}),
    L2norm("Velocity", {"save": True, "plot": True}),
])
```

The `StreamFunction` and `L2norm` of `Velocity` only depend on the velocity, and we expect this to be computed at the same timesteps we computed the velocity in the original solve.

The `Stress`, however, depends on both the `Velocity` and the `Pressure`. Since the velocity was saved at every second timestep, and the pressure was only saved every third timestep, we expect this to be computed at timesteps 0, 6, 12, ...

We then initiate a `NSReplay` instance with the postprocessor-instance as argument, and call its *replay*-function to execute the replay routine:

```
# Replay
replayer = NSReplay(postprocessor)
replayer.replay()
```

2.4 Restart a problem

This demo demonstrates the functionality of restarting a problem. This could be useful for example if one wants to run the simulation for longer than anticipated on initial solve, if one wants parts of the solution to have a different temporal resolution or if one wants to completely change the boundary conditions of the problem at a certain point.

It is based on the Beltrami problem, a 3D problem with a known analytical solution.

The source code for this demo can be found in `Restart.py`.

We start by importing the problem,

```
from os import path
import sys
# Add Beltrami problem as example problem
sys.path.insert(0, path.join(path.dirname(path.realpath(__file__)), '../../../demo/undocumented/Belt
from Beltrami import Beltrami
```

and cbcflow:

```
from cbcflow import *
```

2.4.1 Play

We define the normal solve routine, by first initiating a problem and scheme:

```
def play():
    problem = Beltrami(dict(dt=1e-2, T=1.0))
    scheme = IPCS()
```

Note that we here use the `IPCS` scheme.

We then creates some fields to save for this solve:

```
fields = [
    Velocity(dict(save=True, stride_timestep=5)),
    Pressure(dict(save=True, stride_timestep=10)),
    L2norm("Velocity", dict(save=True, stride_timestep=2))
]
```

Note that we *must* save velocity and pressure for restart to work.

We then add the fields to a `NSPostProcessor` instance,

```
postprocessor = NSPostProcessor(dict(casedir='results'))
postprocessor.add_fields(fields)
```

and solves the equation:

```
solver = NSSolver(problem, scheme, postprocessor)
solver.solve()
```

2.4.2 Restart

When we restart the problem, we wish to reuse most of the parameters of the original solve. These are save by the postprocessor to the case directory, and can be *unpickled*:

```
def restart():
    # Load params, to reuse
    import pickle
    params = pickle.load(open('results/params.pickle', 'r'))
```

If we don't change any of the parameters, we would basically be solving the exact same problem. Thus, we change the end time and time step of the problem:

```
problem = Beltrami(params.problem)
problem.params.T = 2.0
problem.params.dt = 5e-3
```

We are also free to change the scheme, so we change the scheme to `IPCS_Stable`:

```
scheme = IPCS_Stable(params.scheme)
```

On the restart, we also set up a different set of fields

```
# Set up postprocessor with new fields
fields = [
    Velocity(dict(save=True)),
    Pressure(dict(save=True)),
    WSS(dict(save=True)),
]
```

and a new `NSPostProcessor` instance:

```
postprocessor = NSPostProcessor(dict(casedir='results'))
postprocessor.add_fields(fields)
```

We then need to define our new solver, and set some restart-specific parameters:

```
solver = NSSolver(problem, scheme, postprocessor)
solver.params["restart"] = True
solver.params["restart_time"] = 0.5
```

The solver will try to search for a solution in the postprocessors case directory at time 0.5, and replace the the method `initial_conditions()` in the `NSProblem` instance to reflect the solution at t=0.5.

Our call to solve will then restart this problem from the specified parameters, and solve the problem:

```
solver.solve()
```

CHAPTER 3

Functionality (TODO)

3.1 Solve

TODO: Write about solve functionality.

3.2 Postprocessing

TODO: Write about postprocessing functionality

3.3 Restart

TODO: Write about restart functionality.

3.4 Replay

TODO: Write about replay functionality.

3.5 Solve

TODO: Write about solve functionality.

3.6 Postprocessing

TODO: Write about postprocessing functionality

3.7 Restart

TODO: Write about restart functionality.

3.8 Replay

TODO: Write about replay functionality.

CHAPTER 4

Design (TODO)

4.1 General ideas

TODO: Write about the general ideas of cbcflow, such as a modular design and communcation through NSSolver.

4.2 Schemes

TODO: Wrote about schemes, how they can be developed, used and validated.

4.3 Problems

TODO: Write about how problems are defined and used.

4.4 Postprocessing

TODO: Write about the postprocessing framework

4.4.1 PPFields

Something about PPFields

4.4.2 Plot

How plot is handles

4.4.3 Save

How save is handled, saveformat etc.

4.5 Special objects

TODO: Write about special objects

4.5.1 ParamDict

Write about ParamDict here

4.5.2 NSSpacePool

Write about NSSpacePool

4.5.3 Helper functions

Write about helper functions here, such as list_schemes() etc.

4.6 General ideas

TODO: Write about the general ideas of cbcflow, such as a modular design and communcation through NSSolver.

4.7 Schemes

TODO: Wrote about schemes, how they can be developed, used and validated.

4.8 Problems

TODO: Write about how problems are defined and used.

4.9 Postprocessing

TODO: Write about the postprocessing framework

4.9.1 PPFields

Something about PPFields

4.9.2 Plot

How plot is handles

4.9.3 Save

How save is handled, saveformat etc.

4.10 Special objects

TODO: Write about special objects

4.10.1 ParamDict

Write about ParamDict here

4.10.2 NSSpacePool

Write about NSSpacePool

4.10.3 Helper functions

Write about helper functions here, such as list_schemes() etc.

CHAPTER 5

Programmer's reference

This is the base module of cbcflow.

To use cbcflow, do:

```
from cbcflow import *
```

Modules:

5.1 cbcflow.bcs module

Helper modules for specifying inlet and outlet boundary conditions.

Modules:

5.1.1 cbcflow.bcs.Poiseuille module

Classes

class cbcflow.bcs.Poiseuille.**PoiseuilleComponent**(*args*)

Bases: Expression

eval(*value*, *x*)

set_t(*t*)

class cbcflow.bcs.Poiseuille.**Poiseuille**(*coeffs*, *mesh*, *indicator*, *scale_to=None*, *facet_domains=None*)

Bases: list

Functions

cbcflow.bcs.Poiseuille.**make_poiseuille_bcs**(*coeffs*, *mesh*, *indicator*, *scale_to=None*, *facet_domains=None*)

Generate a list of expressions for the components of a Poiseuille profile.

5.1.2 cbcflow.bcs.Resistance module

Classes

class cbcflow.bcs.Resistance.**Resistance**(*C*, *u*, *ind*, *facet_domains*)
Bases: Constant

Functions

cbcflow.bcs.Resistance.**compute_resistance_value**(*C*, *u*, *ind*, *facet_domains*)

5.1.3 cbcflow.bcs.UniformShear module

Classes

class cbcflow.bcs.UniformShear.**UniformShear**(*u*, *ind*, *facet_domains*, *C=10000*)
Bases: Constant

Functions

cbcflow.bcs.UniformShear.**compute_uniform_shear_value**(*u*, *ind*, *facet_domains*, *C=10000*)

5.1.4 cbcflow.bcs.Womersley module

Classes

class cbcflow.bcs.Womersley.**Womersley**(*coeffs*, *mesh*, *indicator*, *nu*, *scale_to=None*, *facet_domains=None*)
Bases: list

class cbcflow.bcs.Womersley.**WomersleyComponent1**(*args*)
Bases: Expression

eval(*value*, *x*)

set_t(*t*)

class cbcflow.bcs.Womersley.**WomersleyComponent2**(*args*)
Bases: Expression

eval(*value*, *x*)

set_t(*t*)

Functions

cbcflow.bcs.Womersley.**fourier_coefficients**(*x*, *y*, *T*, *N=25*)
From x-array and y-spline and period T, calculate N complex Fourier coefficients.

cbcflow.bcs.Womersley.**make_womersley_bcs**(*coeffs*, *mesh*, *indicator*, *nu*, *scale_to=None*, *facet_domains=None*, *coeffstype='Q'*, *num_fourier_coefficients=25*)
Generate a list of expressions for the components of a Womersley profile.

5.2 cbcflow.core module

Core modules of cbcflow.

Modules:

5.2.1 cbcflow.core.nspostprocessor module

Classes

class `cbcflow.core.nspostprocessor.`**`NSPostProcessor`**(*params=None*)

Bases: `cbcflow.core.parameterized.Parameterized`

`add_field`(*field*)
Add field to postprocessor. Recursively adds basic dependencies.

`add_fields`(*fields*)
Add several fields at once.

classmethod `default_params`()

`finalize_all`(*spaces*, *problem*)
Finalize all PPFields after last timestep has been computed.

`find_dependencies`(*field*)
Read dependencies from source code in field.compute function

`get`(*name*, *timestep=0*)
Get the value of a named field at a particular.

The timestep is relative to now. Values are computed at first request and cached.

`get_casedir`()

`get_savedir`(*field_name*)
Returns savedir for given fieldname

`store_mesh`(*mesh*)
Store mesh in casedir to mesh.hdf5 (dataset Mesh) in casedir.

`store_params`(*params*)
Store parameters in casedir as params.pickle and params.txt.

`update_all`(*solution*, *t*, *timestep*, *spaces*, *problem*)
Updates cache, plan, play log and executes plan.

class `cbcflow.core.nspostprocessor.`**`DependencyException`**(*fieldname=None*, *dependency=None*, *timestep=None*, *original_exception_msg=None*)

Bases: `exceptions.Exception`

Functions

`cbcflow.core.nspostprocessor.`**`import_pylab`**()
Set up pylab if available.

`cbcflow.core.nspostprocessor.`**`disable_plotting`**()
Disable all plotting if we run in parallell.

5.2.2 cbcflow.core.nsproblem module

Classes

class `cbcflow.core.nsproblem.NSProblem`(*params*)

Bases: `cbcflow.core.parameterized.Parameterized`

Base class for all Navier-Stokes problems.

`analytical_solution`(*spaces*, *t*)

Return analytical solution.

Can be ignored when no such solution exists, this is only used in the validation frameworks to validate schemes and test grid convergence etc.

TODO: Document expected analytical_solution behaviour here.

Returns: u, p

`body_force`(*spaces*, *t*)

Return body force, defaults to 0.

If not overridden by subclass this function will return zero.

Returns: list of scalars.

`boundary_conditions`(*spaces*, *u*, *p*, *t*, *controls*)

Return boundary conditions in raw format.

Boundary conditions should . The boundary conditions can be specified as follows:

```
# Specify u=(0,0,0) on mesh domain 0 and u=(x,y,z) on mesh domain 1
bcu = [
    ([Constant(0), Constant(0), Constant(0)], 0),
    ([Expression("x[0]"), Expression("x[1]"), Expression("x[2]")], 1)
    ]

# Specify p=x^2+y^2 on mesh domain 2 and p=0 on mesh domain 3
bcp = [
    (Expression("x[0]*x[0]+x[1]*x[1]"), 2),
    (Constant(0), 3)
]

return bcu, bcp
```

Note that the velocity is specified as a list of scalars instead of vector expressions.

For schemes applying Dirichlet boundary conditions, the domain argument(s) are parsed to DirichletBC and can be specified in a matter that matches the signature of this class.

This function must be overridden py subclass.

Returns: a tuple with boundary conditions for velocity and pressure

`controls`(*spaces*)

Return controls for optimization problem.

Optimization problem support is currently experimental. Can be ignored for non-control problems.

TODO: Document expected controls behaviour here.

classmethod `default_params`()

Returns the default parameters for a problem.

Explanation of parameters:

Time parameters:

- start_timestep: int, initial time step number
- dt: float, time discretization value
- T0: float, initial time
- T: float, end time
- period: float, length of period
- num_periods: float, number of periods to run

Either T or period and num_period must be set. If T is not set, T=T0+period*num_periods is used.

Physical parameters:

- mu: float, kinematic viscosity
- rho: float, mass density

Space discretization parameters:

- mesh_file: str, filename to load mesh from (if any)

initial_conditions (*spaces*, *controls*)

Return initial conditions.

The initial conditions should be specified as follows: :: # Return u=(x,y,0) and p=0 as initial conditions u0 = [Expression("x[0]"), Expression("x[1]"), Constant(0)] p0 = Constant(0) return u0, p0

Note that the velocity is specified as a list of scalars instead of vector expressions.

This function must be overridden py subclass.

Returns: u, p

initialize_geometry (*mesh*, *facet_domains=None*, *cell_domains=None*)

Stores mesh, domains and related quantities in a canonical member naming.

Creates attributes on self:

- mesh
- facet_domains
- cell_domains
- ds
- dS
- dx

observations (*spaces*, *t*)

Return observations of velocity for optimization problem.

Optimization problem support is currently experimental. Can be ignored for non-control problems.

TODO: Document expected observations behaviour here.

test_functionals (*spaces*)

Return fields to be used by regression tests.

Can be ignored when no such solution exists, this is only used in the validation frameworks to validate schemes and test grid convergence etc.

Returns: list of fields.

test_references()

Return reference values corresponding to test_functionals to be used by regression tests.

Can be ignored when no such solution exists, this is only used in the validation frameworks to validate schemes and test grid convergence etc.

Returns: list of reference values.

update(*spaces*, *u*, *p*, *t*, *timestep*, *boundary_conditions*, *observations=None*, *controls=None*)

Update functions previously returned to new timestep.

This function is called before computing the solution at a new timestep.

The arguments boundary_conditions, observations, controls should be the exact lists of objects returned by boundary_conditions, observations, controls.

Typical usage of this function would be to update time-dependent boundary conditions:

```
bcu, bcp = boundary_conditions
for bc, _ in bcu:
    bc.t = t

for bc, _ in bcp:
    bc.t = t
```

returns None

5.2.3 cbcflow.core.nsreplay module

Classes

class cbcflow.core.nsreplay.**NSReplay**(*postprocessor*, *params=None*)

Bases: cbcflow.core.parameterized.Parameterized

Replay class for postprocessing exisiting solution data.

classmethod default_params()

replay()

Replay problem with given postprocessor.

Functions

cbcflow.core.nsreplay.**print_replay_plan**(*plan*)

cbcflow.core.nsreplay.**have_necessary_deps**(*solution*, *pp*, *field*)

5.2.4 cbcflow.core.nsscheme module

Classes

class cbcflow.core.nsscheme.**NSScheme**(*params=None*)

Bases: cbcflow.core.parameterized.Parameterized

Base class for all Navier-Stokes schemes.

TODO: Clean up and document new interface.

classmethod default_params()

solve(*problem*, *update*)
Solve Navier-Stokes problem by executing scheme.

5.2.5 cbcflow.core.nssolver module

Classes

class cbcflow.core.nssolver.**NSSolver**(*problem*, *scheme=None*, *postprocessor=None*, *params=None*)
Bases: cbcflow.core.parameterized.Parameterized

High level Navier-Stokes solver. This handles all logic between the cbcflow components.

For full functionality, the user should instantiate this class with a NSProblem instance, NSScheme instance and NSPostProcessor instance.

classmethod default_params()
Returns the default parameters for a problem.

Explanation of parameters:

- debug: bool, debug mode
- check_mem_frequency: int, timestep frequency to check memory consumption
- restart: bool, turn restart mode on or off
- restart_time: float, time to search for restart data
- restart_timestep: int, timestep to search for restart data

If restart=True, maximum one of restart_time and restart_timestep can be set.

solve()
Handles top level logic related to solve.

Cleans casedir or loads restart data, stores parameters and mesh in casedir, calls scheme.solve, and lets postprocessor finalize all fields.

Returns: namespace dict returned from scheme.solve

update(*u*, *p*, *t*, *timestep*, *spaces*)
Callback from scheme.solve after each timestep to handle update of postprocessor, timings, memory etc.

5.2.6 cbcflow.core.paramdict module

Classes

class cbcflow.core.paramdict.**ParamDict**(**args*, ***kwargs*)
Bases: dict

arg_assign(*name*, *value*)

copy_recursive()
Copy ParamDict hierarchy recursively, using copy.deepcopy() to copy values.

items()

iterdeep()
Iterate recursively over all parameter items.

iteritems()

iterkeys()

keys()

parse_args(*args*)

pop(*name*, *default=None*)
Returns Paramdict[name] if the key exists. If the key does not exist the default value is returned.

render_args()

replace(*params=None*, ***kwparams*)
Perform a recursive update where no new keys are allowed.

replace_recursive(*params=None*, ***kwparams*)
Perform a recursive update where no new keys are allowed.

replace_shallow(*params=None*, ***kwparams*)
Perform a shallow update where no new keys are allowed.

update(*params=None*, ***kwparams*)
Perform a recursive update, allowing new keys to be introduced.

update_recursive(*params=None*, ***kwparams*)
Perform a recursive update, allowing new keys to be introduced.

update_shallow(*params=None*, ***kwparams*)
Perform a shallow update, allowing new keys to be introduced.

5.2.7 cbcflow.core.parameterized module

Classes

class cbcflow.core.parameterized.**Parameterized**(*params*)
Bases: object

Core functionality for parameterized subclassable components.

classmethod default_params()
Merges base and user params into one ParamDict.

classmethod description()
Get a one-sentence description of what the class represents.

By default uses first line of class docstring.

classmethod shortname()
Get a one-word description of what the class represents.

By default uses class name.

5.2.8 cbcflow.core.restart module

Classes

class cbcflow.core.restart.**Restart**(*problem*, *postprocessor*, *restart_time*, *restart_timestep*)
Bases: object

Class to specify restart data. Searches within the given postprocessors casedir for solution data to find a suitable timestep to restart from, based on the parameters restart_time or restart_timestep.

Overwrites problem.initial_conditions with solution loaded from file and sets problem.params.T0. Handles postprocessing to avoid overwrite conflicts with existing data.

Functions

`cbcflow.core.restart.`**`find_common_savetimesteps`**(*play_log*, *fields*)

5.3 cbcflow.fields module

A collection of postprocessing fields (PPFields) to be used by a NSPostProcessor object.

Modules:

5.3.1 cbcflow.fields.bases module

Base classes for all postprocessing fields.

Classes

class `cbcflow.fields.bases.`**`MetaPPField2`**(*value1*, *value2*, *params=None*, *label=None*)

Bases: `cbcflow.fields.bases.PPField.PPField`

`name`

class `cbcflow.fields.bases.`**`PPField`**(*params=None*, *label=None*)

Bases: `cbcflow.core.parameterized.Parameterized`

`after_last_compute`(*pp*, *spaces*, *problem*)

Called after the simulation timeloop.

`before_first_compute`(*pp*, *spaces*, *problem*)

Called prior to the simulation timeloop.

`compute`(*pp*, *spaces*, *problem*)

Called each time the quantity should be computed.

`convert`(*pp*, *spaces*, *problem*)

Called if quantity is input to NSPostProcessor.update_all

classmethod `default_params`()

classmethod `default_save_as`()

`expr2function`(*expr*, *function*)

`name`

Return name of field, by default the classname but can be overloaded in subclass.

class `cbcflow.fields.bases.`**`MetaPPField`**(*value*, *params=None*, *label=None*)

Bases: `cbcflow.fields.bases.PPField.PPField`

`name`

5.3.2 cbcflow.fields.basic module

Basic postprocessing fields.

These fields can all be created from the postprocessor from name only. This is useful when handling dependencies for a postprocessing field:

```
class DummyField(PPField):
    def __init__(self, field_dep):
        self.field_dep = field_dep

    def compute(self, pp, spaces, problem):
        val = pp.get(field_dep)
        return val/2.0
```

If a postprocessing field depends only on basic fields to be calculated, the dependencies will be implicitly added to the postprocessor "on the fly" from the name alone:

```
field = DummyField("ABasicField")
pp = NSPostProcessor()
pp.add_field(field) # Implicitly adds ABasicField object
```

For non-basic dependencies, the dependencies have to be explicitly added *before* the field depending on it:

```
dependency = ANonBasicField("ABasicField")
field = DummyField(dependency.name)
pp.add_field(dependency) # Added before field
pp.add_field(field) # pp now knows about dependency
```

Modules:

cbcflow.fields.basic.AnalyticalPressure module

Classes

class `cbcflow.fields.basic.AnalyticalPressure.`**`AnalyticalPressure`**(*params=None*, *label=None*)

Bases: `cbcflow.fields.bases.PPField.PPField`

`before_first_compute`(*pp*, *spaces*, *problem*)

`compute`(*pp*, *spaces*, *problem*)

classmethod `default_params`()

cbcflow.fields.basic.AnalyticalVelocity module

Classes

class `cbcflow.fields.basic.AnalyticalVelocity.`**`AnalyticalVelocity`**(*params=None*, *label=None*)

Bases: `cbcflow.fields.bases.PPField.PPField`

`before_first_compute`(*pp*, *spaces*, *problem*)

`compute`(*pp*, *spaces*, *problem*)

classmethod `default_params`()

cbcflow.fields.basic.Delta module

Classes

class `cbcflow.fields.basic.Delta.`**`Delta`**(*params=None*, *label=None*)

Bases: `cbcflow.fields.bases.PPField.PPField`

`before_first_compute`(*pp*, *spaces*, *problem*)

`compute`(*pp*, *spaces*, *problem*)

classmethod **`default_params`**()

cbcflow.fields.basic.KineticEnergy module

Classes

class `cbcflow.fields.basic.KineticEnergy.`**`KineticEnergy`**(*params=None*, *label=None*)

Bases: `cbcflow.fields.bases.PPField.PPField`

`compute`(*pp*, *spaces*, *problem*)

cbcflow.fields.basic.Lambda2 module

Classes

class `cbcflow.fields.basic.Lambda2.`**`Lambda2`**(*params=None*, *label=None*)

Bases: `cbcflow.fields.bases.PPField.PPField`

`before_first_compute`(*pp*, *spaces*, *problem*)

`compute`(*pp*, *spaces*, *problem*)

classmethod **`default_params`**()

cbcflow.fields.basic.LocalCfl module

Classes

class `cbcflow.fields.basic.LocalCfl.`**`LocalCfl`**(*params=None*, *label=None*)

Bases: `cbcflow.fields.bases.PPField.PPField`

`before_first_compute`(*pp*, *spaces*, *problem*)

`compute`(*pp*, *spaces*, *problem*)

cbcflow.fields.basic.PhysicalPressure module

Classes

class `cbcflow.fields.basic.PhysicalPressure.`**`PhysicalPressure`**(*params=None*, *label=None*)

Bases: `cbcflow.fields.bases.PPField.PPField`

The physical pressure is the solver pressure scaled by density.

compute (*pp*, *spaces*, *problem*)

cbcflow.fields.basic.Pressure module

Classes

class cbcflow.fields.basic.Pressure.**Pressure** (*params=None*, *label=None*)
Bases: cbcflow.fields.bases.PPField.PPField

convert (*pp*, *spaces*, *problem*)

cbcflow.fields.basic.Pressure.**SolverPressure**
alias of Pressure

cbcflow.fields.basic.PressureError module

Classes

class cbcflow.fields.basic.PressureError.**PressureError** (*params=None*, *label=None*)
Bases: cbcflow.fields.bases.PPField.PPField

before_first_compute (*pp*, *spaces*, *problem*)

compute (*pp*, *spaces*, *problem*)

classmethod default_params ()

cbcflow.fields.basic.PressureGradient module

Classes

class cbcflow.fields.basic.PressureGradient.**PressureGradient** (*params=None*, *label=None*)
Bases: cbcflow.fields.bases.PPField.PPField

before_first_compute (*pp*, *spaces*, *problem*)

compute (*pp*, *spaces*, *problem*)

cbcflow.fields.basic.Q module

Classes

class cbcflow.fields.basic.Q.**Q** (*params=None*, *label=None*)
Bases: cbcflow.fields.bases.PPField.PPField

before_first_compute (*pp*, *spaces*, *problem*)

compute (*pp*, *spaces*, *problem*)

classmethod default_params ()

cbcflow.fields.basic.Strain module

Classes

class `cbcflow.fields.basic.Strain.`**`Strain`**(*params=None*, *label=None*)
Bases: `cbcflow.fields.bases.PPField.PPField`

`before_first_compute`(*pp*, *spaces*, *problem*)

`compute`(*pp*, *spaces*, *problem*)

cbcflow.fields.basic.StreamFunction module

Classes

class `cbcflow.fields.basic.StreamFunction.`**`StreamFunction`**(*params=None*, *label=None*)
Bases: `cbcflow.fields.bases.PPField.PPField`

`before_first_compute`(*pp*, *spaces*, *problem*)

`compute`(*pp*, *spaces*, *problem*)

cbcflow.fields.basic.Stress module

Classes

class `cbcflow.fields.basic.Stress.`**`Stress`**(*params=None*, *label=None*)
Bases: `cbcflow.fields.bases.PPField.PPField`

`before_first_compute`(*pp*, *spaces*, *problem*)

`compute`(*pp*, *spaces*, *problem*)

cbcflow.fields.basic.Velocity module

Classes

class `cbcflow.fields.basic.Velocity.`**`Velocity`**(*params=None*, *label=None*)
Bases: `cbcflow.fields.bases.PPField.PPField`

`convert`(*pp*, *spaces*, *problem*)

cbcflow.fields.basic.VelocityCurl module

Classes

class `cbcflow.fields.basic.VelocityCurl.`**`VelocityCurl`**(*params=None*, *label=None*)
Bases: `cbcflow.fields.bases.PPField.PPField`

`compute`(*pp*, *spaces*, *problem*)

cbcflow.fields.basic.VelocityDivergence module

Classes

class cbcflow.fields.basic.VelocityDivergence.**VelocityDivergence**(*params=None*, *label=None*)

Bases: cbcflow.fields.bases.PPField.PPField

compute(*pp*, *spaces*, *problem*)

cbcflow.fields.basic.VelocityError module

Classes

class cbcflow.fields.basic.VelocityError.**VelocityError**(*params=None*, *label=None*)

Bases: cbcflow.fields.bases.PPField.PPField

before_first_compute(*pp*, *spaces*, *problem*)

compute(*pp*, *spaces*, *problem*)

classmethod default_params()

cbcflow.fields.basic.VelocityGradient module

Classes

class cbcflow.fields.basic.VelocityGradient.**VelocityGradient**(*params=None*, *label=None*)

Bases: cbcflow.fields.bases.PPField.PPField

before_first_compute(*pp*, *spaces*, *problem*)

compute(*pp*, *spaces*, *problem*)

cbcflow.fields.basic.WSS module

Classes

class cbcflow.fields.basic.WSS.**WSS**(*params=None*, *label=None*)

Bases: cbcflow.fields.bases.PPField.PPField

before_first_compute(*pp*, *spaces*, *problem*)

compute(*pp*, *spaces*, *problem*)

Functions

cbcflow.fields.basic.WSS.**local_mesh_to_boundary_dofmap**(*boundary*, *V*, *Vb*)

5.3.3 cbcflow.fields.meta module

Fields that require input parameters. This is typically fields that can be used on different fields, for example time derivatives, averages, parts of fields etc.

Modules:

cbcflow.fields.meta.BoundaryAvg module

Classes

class `cbcflow.fields.meta.BoundaryAvg.`**`BoundaryAvg`**(*value*, *params=None*, *label=None*)

Bases: `cbcflow.fields.bases.MetaPPField.MetaPPField`

`compute`(*pp*, *spaces*, *problem*)

cbcflow.fields.meta.DiffH1norm module

Classes

class `cbcflow.fields.meta.DiffH1norm.`**`DiffH1norm`**(*value1*, *value2*, *params=None*, *label=None*)

Bases: `cbcflow.fields.bases.MetaPPField2.MetaPPField2`

Compute the full H1 norm of the difference between uh and u relative to u.

`compute`(*pp*, *spaces*, *problem*)

cbcflow.fields.meta.DiffH1seminorm module

Classes

class `cbcflow.fields.meta.DiffH1seminorm.`**`DiffH1seminorm`**(*value1*, *value2*, *params=None*, *label=None*)

Bases: `cbcflow.fields.bases.MetaPPField2.MetaPPField2`

Compute the H1 semi norm of the difference between uh and u relative to u.

`compute`(*pp*, *spaces*, *problem*)

cbcflow.fields.meta.DiffL2norm module

Classes

class `cbcflow.fields.meta.DiffL2norm.`**`DiffL2norm`**(*value1*, *value2*, *params=None*, *label=None*)

Bases: `cbcflow.fields.bases.MetaPPField2.MetaPPField2`

Compute the L2 norm of the difference between uh and u relative to u.

`compute`(*pp*, *spaces*, *problem*)

cbcflow.fields.meta.DomainAvg module

Classes

class cbcflow.fields.meta.DomainAvg.**DomainAvg**(*value*, *params=None*, *label=None*)
Bases: cbcflow.fields.bases.MetaPPField.MetaPPField

compute(*pp*, *spaces*, *problem*)

cbcflow.fields.meta.FlowRate module

Classes

class cbcflow.fields.meta.FlowRate.**FlowRate**(*boundary_id*, *params=None*, *label=None*)
Bases: cbcflow.fields.bases.PPField.PPField

compute(*pp*, *spaces*, *problem*)

name

cbcflow.fields.meta.H1norm module

Classes

class cbcflow.fields.meta.H1norm.**H1norm**(*value*, *params=None*, *label=None*)
Bases: cbcflow.fields.bases.MetaPPField.MetaPPField

compute(*pp*, *spaces*, *problem*)

cbcflow.fields.meta.H1seminorm module

Classes

class cbcflow.fields.meta.H1seminorm.**H1seminorm**(*value*, *params=None*, *label=None*)
Bases: cbcflow.fields.bases.MetaPPField.MetaPPField

compute(*pp*, *spaces*, *problem*)

cbcflow.fields.meta.L2norm module

Classes

class cbcflow.fields.meta.L2norm.**L2norm**(*value*, *params=None*, *label=None*)
Bases: cbcflow.fields.bases.MetaPPField.MetaPPField

compute(*pp*, *spaces*, *problem*)

cbcflow.fields.meta.Linfnorm module

Classes

class cbcflow.fields.meta.Linfnorm.**Linfnorm**(*value*, *params=None*, *label=None*)
Bases: cbcflow.fields.bases.MetaPPField.MetaPPField

compute(*pp*, *spaces*, *problem*)

cbcflow.fields.meta.Magnitude module

Classes

class cbcflow.fields.meta.Magnitude.**Magnitude**(*value*, *params=None*, *label=None*)
Bases: cbcflow.fields.bases.MetaPPField.MetaPPField

compute(*pp*, *spaces*, *problem*)

cbcflow.fields.meta.Maximum module

Classes

class cbcflow.fields.meta.Maximum.**Maximum**(*value*, *params=None*, *label=None*)
Bases: cbcflow.fields.bases.MetaPPField.MetaPPField

compute(*pp*, *spaces*, *problem*)

cbcflow.fields.meta.Minimum module

Classes

class cbcflow.fields.meta.Minimum.**Minimum**(*value*, *params=None*, *label=None*)
Bases: cbcflow.fields.bases.MetaPPField.MetaPPField

compute(*pp*, *spaces*, *problem*)

cbcflow.fields.meta.PointEval module

Classes

class cbcflow.fields.meta.PointEval.**PointEval**(*value*, *points*, *params=None*, *label=None*)
Bases: cbcflow.fields.bases.MetaPPField.MetaPPField

after_last_compute(*pp*, *spaces*, *problem*)

before_first_compute(*pp*, *spaces*, *problem*)

compute(*pp*, *spaces*, *problem*)

name

Functions

cbcflow.fields.meta.PointEval.**points_in_ball**(*center*, *radius*, *resolution*)

cbcflow.fields.meta.PointEval.**points_in_circle**(*center*, *radius*, *resolution*)

cbcflow.fields.meta.PointEval.**points_in_square**(*center*, *radius*, *resolution*)

cbcflow.fields.meta.PointEval.**points_in_box**(*center*, *radius*, *resolution*)

cbcflow.fields.meta.PointEval.**import_fenicstools**()

cbcflow.fields.meta.RunningAvg module

Classes

class cbcflow.fields.meta.RunningAvg.**RunningAvg**(*value*, *params=None*, *label=None*)

Bases: cbcflow.fields.bases.MetaPPField.MetaPPField

after_last_compute(*pp*, *spaces*, *problem*)

before_first_compute(*pp*, *spaces*, *problem*)

compute(*pp*, *spaces*, *problem*)

cbcflow.fields.meta.RunningL2norm module

Classes

class cbcflow.fields.meta.RunningL2norm.**RunningL2norm**(*value*, *params=None*, *label=None*)

Bases: cbcflow.fields.bases.MetaPPField.MetaPPField

before_first_compute(*pp*, *spaces*, *problem*)

compute(*pp*, *spaces*, *problem*)

cbcflow.fields.meta.RunningMax module

Classes

class cbcflow.fields.meta.RunningMax.**RunningMax**(*value*, *params=None*, *label=None*)

Bases: cbcflow.fields.bases.MetaPPField.MetaPPField

after_last_compute(*pp*, *spaces*, *problem*)

before_first_compute(*pp*, *spaces*, *problem*)

compute(*pp*, *spaces*, *problem*)

cbcflow.fields.meta.RunningMin module

Classes

class cbcflow.fields.meta.RunningMin.**RunningMin**(*value*, *params=None*, *label=None*)

Bases: cbcflow.fields.bases.MetaPPField.MetaPPField

after_last_compute(*pp*, *spaces*, *problem*)

before_first_compute(*pp*, *spaces*, *problem*)

compute(*pp*, *spaces*, *problem*)

cbcflow.fields.meta.SecondTimeDerivative module

Classes

class cbcflow.fields.meta.SecondTimeDerivative.**SecondTimeDerivative**(*value*, *params=None*, *label=None*)

Bases: cbcflow.fields.bases.MetaPPField.MetaPPField

compute(*pp*, *spaces*, *problem*)

cbcflow.fields.meta.SubFunction module

Classes

class cbcflow.fields.meta.SubFunction.**SubFunction**(*field*, *submesh*, *params=None*, *label=None*)

Bases: cbcflow.fields.bases.PPField.PPField

before_first_compute(*pp*, *spaces*, *problem*)

compute(*pp*, *spaces*, *problem*)

name

Functions

cbcflow.fields.meta.SubFunction.**import_fenicstools**()

cbcflow.fields.meta.TimeDerivative module

Classes

class cbcflow.fields.meta.TimeDerivative.**TimeDerivative**(*value*, *params=None*, *label=None*)

Bases: cbcflow.fields.bases.MetaPPField.MetaPPField

compute(*pp*, *spaces*, *problem*)

cbcflow.fields.meta.TimeIntegral module

Classes

class cbcflow.fields.meta.TimeIntegral.**TimeIntegral**(*value*, *params=None*, *label=None*)

Bases: cbcflow.fields.bases.MetaPPField.MetaPPField

compute(*pp*, *spaces*, *problem*)

5.3.4 Functions

`cbcflow.fields.show_fields()`
Lists which fields are available.

5.4 cbcflow.schemes module

A collection of Navier-Stokes schemes.

Modules:

5.4.1 cbcflow.schemes.experimental module

These schemes are considered experimental, and one can not generally expect correct simulations results with these.

They implement a variety of ideas for solving the Navier-Stokes equations, but lack the testing and validation of the *official* schemes.

Modules:

cbcflow.schemes.experimental.bottipietro module

Classes

class `cbcflow.schemes.experimental.bottipietro.BottiPietro(params=None)`
Bases: `cbcflow.core.nsscheme.NSScheme`

TODO: Describe.

classmethod `default_params()`

`solve(problem, update)`

cbcflow.schemes.experimental.coupled_picard module

Classes

class `cbcflow.schemes.experimental.coupled_picard.CoupledPicard(params=None)`
Bases: `cbcflow.core.nsscheme.NSScheme`

Coupled scheme using a fixed point (Picard) nonlinear solver.

classmethod `default_params()`

`solve(problem, update)`

cbcflow.schemes.experimental.couplednonlinear module

Classes

class `cbcflow.schemes.experimental.couplednonlinear.CoupledNonLinear(params=None)`
Bases: `cbcflow.core.nsscheme.NSScheme`

Coupled scheme with fixed-point iterations on the convection term. NB: Direct solver!

classmethod default_params()

solve(*problem*, *update*, *restart=None*)

cbcflow.schemes.experimental.coupledpreconditioned module

Classes

class cbcflow.schemes.experimental.coupledpreconditioned.**CoupledPreconditioned**(*params=None*)
Bases: cbcflow.core.nsscheme.NSScheme

Coupled scheme with block preconditioning using cbc.block

classmethod default_params()

solve(*problem*, *update*)

cbcflow.schemes.experimental.coupledpreconditioned_kam module

Classes

class cbcflow.schemes.experimental.coupledpreconditioned_kam.**CoupledPreconditionedKAM**(*params=N*
Bases: cbcflow.core.nsscheme.NSScheme

Coupled scheme with block preconditioning using cbc.block

classmethod default_params()

solve(*problem*, *update*)

cbcflow.schemes.experimental.ipcs_opt_seg module

Classes

class cbcflow.schemes.experimental.ipcs_opt_seg.**SegregatedIPCS_Optimized**(*params=None*)
Bases: cbcflow.core.nsscheme.NSScheme

Incremental pressure-correction scheme, optimized version.

classmethod default_params()

solve(*problem*, *update*, *restart=None*)

cbcflow.schemes.experimental.ipcs_penalty module

Classes

class cbcflow.schemes.experimental.ipcs_penalty.**PenaltyIPCS**(*params=None*)
Bases: cbcflow.core.nsscheme.NSScheme

Incremental pressure-correction scheme with penalty terms for boundary conditions.

classmethod default_params()

solve(*problem*, *update*)

cbcflow.schemes.experimental.ipcs_penalty_segregated module

Classes

class `cbcflow.schemes.experimental.ipcs_penalty_segregated.`**`SegregatedPenaltyIPCS`**(*params=None*)
Bases: `cbcflow.core.nsscheme.NSScheme`

Segregated incremental pressure-correction scheme with penalty terms for pressure BCs.

classmethod **`default_params`**()

`solve`(*problem*, *update*, *restart=None*)

cbcflow.schemes.experimental.ipcs_segregated module

Classes

class `cbcflow.schemes.experimental.ipcs_segregated.`**`SegregatedIPCS`**(*params=None*)
Bases: `cbcflow.core.nsscheme.NSScheme`

Segregated incremental pressure-correction scheme.

classmethod **`default_params`**()

`solve`(*problem*, *update*, *restart=None*)

cbcflow.schemes.experimental.ipcs_stabilized module

Classes

class `cbcflow.schemes.experimental.ipcs_stabilized.`**`IPCS_Stabilized`**(*params=None*)
Bases: `cbcflow.core.nsscheme.NSScheme`

Incremental pressure-correction scheme stabilized with SUPG.

classmethod **`default_params`**()

`solve`(*problem*, *update*)

cbcflow.schemes.experimental.karper module

Classes

class `cbcflow.schemes.experimental.karper.`**`Karper`**(*params=None*)
Bases: `cbcflow.core.nsscheme.NSScheme`

TODO: Describe...

classmethod **`default_params`**()

`solve`(*problem*, *update*)

cbcflow.schemes.experimental.piso module

Classes

class `cbcflow.schemes.experimental.piso.`**`PISO`**(*params=None*)
Bases: `cbcflow.core.nsscheme.NSScheme`

PISO, Issa 1985, implemented according to algorithm in Versteeg and Malalasekera

classmethod **`default_params`**()

`solve`(*problem*, *update*)

cbcflow.schemes.experimental.stokes module

Classes

class `cbcflow.schemes.experimental.stokes.`**`Stokes`**(*params=None*)
Bases: `cbcflow.core.nsscheme.NSScheme`

Coupled solver for the transient Stokes problem.

classmethod **`default_params`**()

`solve`(*problem*, *update*)

cbcflow.schemes.experimental.yosida module

Classes

class `cbcflow.schemes.experimental.yosida.`**`Yosida`**(*params=None*)
Bases: `cbcflow.core.nsscheme.NSScheme`

Yosida scheme with lumped mass matrix in the Schur complement

classmethod **`default_params`**()

`solve`(*problem*, *update*)

5.4.2 cbcflow.schemes.official module

These *official* schemes have been validated against reference solutions.

Modules:

cbcflow.schemes.official.ipcs module

This incremental pressure correction scheme (IPCS) is an operator splitting scheme that follows the idea of Goda [1]. This scheme preserves the exact same stability properties as Navier-Stokes and hence does not introduce additional dissipation in the flow.

The idea is to replace the unknown pressure with an approximation. This is chosen as the pressure solution from the previous solution.

[1] Goda, Katuhiko. *A multistep technique with implicit difference schemes for calculating two-or three-dimensional cavity flows.* Journal of Computational Physics 30.1 (1979): 76-95.

The time discretization is done using backward Euler, the diffusion term is handled with Crank-Nicholson, and the convection is handled explicitly, making the equations completely linear. Thus, we have a discretized version of the Navier-Stokes equations as

$$\frac{1}{\Delta t}\left(u^{n+1}-u^n\right)-\nabla\cdot\nu\nabla u^{n+\frac{1}{2}}+u^n\cdot\nabla u^n+\frac{1}{\rho}\nabla p^{n+1}=f^{n+1},$$
$$\nabla\cdot u^{n+1}=0,$$

where $\tilde{u}^{n+\frac{1}{2}}=\frac{1}{2}\tilde{u}^{n+1}+\frac{1}{2}u^n$.

For the operator splitting, we use the pressure solution from the previous timestep as an estimation, giving an equation for a tentative velocity, $\tilde{u}^{n+1}$:

$$\frac{1}{\Delta t}\left(\tilde{u}^{n+1}-u^n\right)-\nabla\cdot\nu\nabla\tilde{u}^{n+\frac{1}{2}}+u^n\cdot\nabla u^n+\frac{1}{\rho}\nabla p^n=f^{n+1}.$$

This tenative velocity is not divergence free, and thus we define a velocity correction $u^c=u^{n+1}-\tilde{u}^{n+1}$. Substracting the second equation from the first, we see that

$$\frac{1}{\Delta t}u^c-\nabla\cdot\nu\nabla u^c+\frac{1}{\rho}\nabla\left(p^{n+1}-p^n\right)=0,$$
$$\nabla\cdot u^c=-\nabla\cdot\tilde{u}^{n+1}.$$

The operator splitting is a first order approximation, $O(\Delta t)$, so we can, without reducing the order of the approximation simplify the above to

$$\frac{1}{\Delta t}u^c+\frac{1}{\rho}\nabla\left(p^{n+1}-p^n\right)=0,$$
$$\nabla\cdot u^c=-\nabla\cdot\tilde{u}^{n+1},$$

which is reducible to a Poisson problem:

$$\Delta p^{n+1}=\Delta p^n+\frac{\rho}{\Delta t}\nabla\cdot\tilde{u}^{n+1}.$$

The corrected velocity is then easily calculated from

$$u^{n+1}=\tilde{u}^{n+1}-\frac{\Delta t}{\rho}\left(p^{n+1}-p^n\right)$$

The scheme can be summarized in the following steps:

1. Replace the pressure with a known approximation and solve for a tenative velocity u^{n+1}.
2. Solve a Poisson equation for the pressure, p^{n+1}
3. Use the corrected pressure to find the velocity correction and calculate u^{n+1}
4. Update t, and repeat.

Classes

class `cbcflow.schemes.official.ipcs.`**`IPCS`**(*params=None*)

Bases: `cbcflow.core.nsscheme.NSScheme`

Incremental pressure-correction scheme.

classmethod **`default_params`**()

`solve`(*problem*, *update*)

cbcflow.schemes.official.ipcs_stable module

This scheme follows the same logic as in `IPCS`, but with a few notable exceptions.

A parameter θ is added to the diffusion and convection terms, allowing for different evaluation of these, and the convection is handled semi-implicitly:

$$\frac{1}{\Delta t}\left(\tilde{u}^{n+1} - u^n\right) - \nabla \cdot \nu \nabla \tilde{u}^{n+\theta} + u^* \cdot \nabla \tilde{u}^{n+\theta} + \nabla p^n = f^{n+1},$$

where

$$u^* = \frac{3}{2}u^n - \frac{1}{2}u^{n-1},$$
$$\tilde{u}^{n+\theta} = \theta \tilde{u}^{n+1} + (1-\theta)\, u^n.$$

This convection term is unconditionally stable, and with $\theta = 0.5$, this equation is second order in time and space [2].

In addition, the solution process is significantly faster by solving for each of the velocity components separately, making for D number of smaller linear systems compared to a large system D times the size.

Classes

class `cbcflow.schemes.official.ipcs_stable.`**`IPCS_Stable`**(*params=None*)

Bases: `cbcflow.core.nsscheme.NSScheme`

Incremental pressure-correction scheme, fast and stable version.

classmethod **`default_params`**()

`solve`(*problem*, *update*, *restart=None*)

5.4.3 Functions

`cbcflow.schemes.`**`show_schemes`**()

Lists which schemes are available.

[2] Simo, J. C., and F. Armero. *Unconditional stability and long-term behavior of transient algorithms for the incompressible Navier-Stokes and Euler equations.* Computer Methods in Applied Mechanics and Engineering 111.1 (1994): 111-154.

5.5 cbcflow.utils module

A collection of internal utilities.

Modules:

5.5.1 cbcflow.utils.bcs module

Utility functions used by the boundary condition modules.

Functions

`cbcflow.utils.bcs.`**`compute_transient_scale_value`**(*bc*, *period*, *mesh*, *facet_domains*, *ind*, *scale_value*)

`cbcflow.utils.bcs.`**`x_to_r2`**(*x*, *c*, *n*)

`cbcflow.utils.bcs.`**`compute_boundary_geometry_acrn`**(*mesh*, *ind*, *facet_domains*)

`cbcflow.utils.bcs.`**`compute_radius`**(*mesh*, *facet_domains*, *ind*, *center*)

`cbcflow.utils.bcs.`**`compute_area`**(*mesh*, *ind*, *facet_domains*)

5.5.2 cbcflow.utils.common module

Utility functions common across cbcflow.

Classes

class `cbcflow.utils.common.`**`Timer`**(*enabled*)

`completed`(*msg*)

Functions

`cbcflow.utils.common.`**`time_to_string`**(*t*)

`cbcflow.utils.common.`**`timeit`**(*t0=None*, *msg=None*)

`cbcflow.utils.common.`**`cbcflow_print`**(*msg*)

`cbcflow.utils.common.`**`has_converged`**(*r*, *iter*, *method*, *maxiter=200*, *tolerance=0.0001*)
Check if solution has converged.

`cbcflow.utils.common.`**`on_master_process`**()

`cbcflow.utils.common.`**`epsilon`**(*u*)
Return symmetric gradient.

`cbcflow.utils.common.`**`as_scalar_spaces`**(*V*)
Return a list of scalar (sub-)spaces consistent with V that can be used to apply DirichletBCs to components.

`cbcflow.utils.common.`**`get_memory_usage`**()

`cbcflow.utils.common.`**`hdf5_link`**(*hdf5filename*, *link_from*, *link_to*)
Create internal link in hdf5 file

`cbcflow.utils.common.`**`as_object`**(*u*)
Return a single object if possible, else a list.

`cbcflow.utils.common.`**`cbcflow_warning`**(*msg*)

`cbcflow.utils.common.`**`in_serial`**()

`cbcflow.utils.common.`**`cbcflow_log`**(*level*, *msg*)

`cbcflow.utils.common.`**`as_scalar_space`**(*V*)
Return a scalar (sub-)space consistent with V that can be used to construct a new Function.

`cbcflow.utils.common.`**`as_list`**(*u*)
Return a list of objects.

`cbcflow.utils.common.`**`parallel_eval`**(*func*, *point*, *gather=True*)
Parallel-safe function evaluation

`cbcflow.utils.common.`**`sigma`**(*u*, *p*, *mu*)
Return stress tensor.

`cbcflow.utils.common.`**`safe_mkdir`**(*dir*)
Create directory without exceptions in parallel.

`cbcflow.utils.common.`**`is_periodic`**(*bcs*)
Check if boundary conditions are periodic.

5.5.3 cbcflow.utils.core module

Utility functions used by the core modules.

Modules:

cbcflow.utils.core.show module

Functions

`cbcflow.utils.core.show.`**`animate_functions`**(*functions*, *name*, *V*)

`cbcflow.utils.core.show.`**`animate_expression`**(*f*, *name*, *V*, *t*, *timesteps*)

cbcflow.utils.core.spaces module

Functions

`cbcflow.utils.core.spaces.`**`galerkin_family`**(*degree*)

`cbcflow.utils.core.spaces.`**`decide_family`**(*family*, *degree*)

Classes

class `cbcflow.utils.core.`**`NSSpacePoolMixed`**(*mesh*, *u_degree*, *p_degree*, *u_family='auto'*, *p_family='auto'*)
Bases: `cbcflow.utils.core.spaces.NSSpacePool`

A function space pool with custom named spaces for use with mixed Navier-Stokes schemes.

Qbc
Scalar valued space for setting pressure BCs.

Ubc
List of scalar valued spaces for setting velocity BCs.

class `cbcflow.utils.core.`**`NSSpacePoolSplit`**(*mesh*, *u_degree*, *p_degree*, *u_family='auto'*, *p_family='auto'*)

Bases: `cbcflow.utils.core.spaces.NSSpacePool`

A function space pool with custom named spaces for use with split Navier-Stokes schemes.

Qbc
Scalar valued space for setting pressure BCs.

Ubc
List of scalar valued spaces for setting velocity BCs.

class `cbcflow.utils.core.`**`NSSpacePool`**(*mesh*, *u_degree*, *p_degree*, *u_family='auto'*, *p_family='auto'*)

Bases: `cbcflow.utils.core.spaces.SpacePool`

A function space pool with custom named spaces for use with Navier-Stokes schemes.

DQ
Vector valued space for pressure gradient.

DQ0
Scalar valued space for pressure gradient component.

DU
Vector valued space for gradients of single velocity components.

DU0
Scalar valued space for gradient component of single velocity component.

DV
Tensor valued space for gradients of velocity vector.

Q
Scalar valued space for pressure.

U
Scalar valued space for velocity components.

U_CG1

V
Vector valued space for velocity vector.

V_CG1

W
Mixed velocity-pressure space.

class `cbcflow.utils.core.`**`NSSpacePoolSegregated`**(*mesh*, *u_degree*, *p_degree*, *u_family='auto'*, *p_family='auto'*)

Bases: `cbcflow.utils.core.spaces.NSSpacePool`

A function space pool with custom named spaces for use with segregated Navier-Stokes schemes.

Qbc
Scalar valued space for setting pressure BCs.

Ubc
List of scalar valued spaces for setting velocity BCs.

class cbcflow.utils.core.**SpacePool**(*mesh*)
Bases: object

A function space pool to reuse spaces across a program.

get_custom_space(*family*, *degree*, *shape*)

get_space(*degree*, *rank*, *family='auto'*)

Functions

cbcflow.utils.core.**strip_code**(*code*)
Strips code of unnecessary spaces, comments etc.

cbcflow.utils.core.**show_problem**(*problem*, *interactive=True*, *bc_snapshots=4*)
Display properties of the problem.

Intended for inspecting and debugging the problem setup. This functions runs through most of the interface

5.5.4 cbcflow.utils.fields module

Utility functions used by postprocessing fields.

Classes

class cbcflow.utils.fields.**Slice**(*basemesh*, *point*, *normal*)
Bases: Mesh

5.5.5 cbcflow.utils.schemes module

Utility functions used some or all of the different schemes.

Classes

class cbcflow.utils.schemes.**RhsGenerator**(*space*)
Bases: object

Class for storing the instructions to create the RHS vector b. The two main purposes of this class are:

- make it easy to define the LHS and RHS in the same place
- make it easy to generate RHS from matrix-XXX products, where XXX may be either * a Constant (which can be projected to a vector at once) * an Expression (which must be projected each time, because its parameters may change) * a Function

Functions

cbcflow.utils.schemes.**make_rhs_pressure_bcs**(*problem*, *spaces*, *bcs*, *v*)

cbcflow.utils.schemes.**assign_ics_mixed**(*up0*, *spaces*, *ics*)

cbcflow.utils.schemes.**assign_ics_split**(*u0*, *p0*, *spaces*, *ics*)

`cbcflow.utils.schemes.`**`compute_regular_timesteps`**(*problem*)

Compute fixed timesteps for problem.

Returns (dt, t0, timesteps), where timesteps does not include t0.

`cbcflow.utils.schemes.`**`make_segregated_velocity_bcs`**(*problem*, *spaces*, *bcs*)

`cbcflow.utils.schemes.`**`make_velocity_bcs`**(*problem*, *spaces*, *bcs*)

`cbcflow.utils.schemes.`**`assign_ics_segregated`**(*u0*, *p0*, *spaces*, *ics*)

`cbcflow.utils.schemes.`**`make_pressure_bcs`**(*problem*, *spaces*, *bcs*)

`cbcflow.utils.schemes.`**`iround`**(*x*)

`cbcflow.utils.schemes.`**`make_penalty_pressure_bcs`**(*problem*, *spaces*, *bcs*, *gamma*, *test*, *trial*)

CHAPTER 6

Indices and tables

- *genindex*
- *modindex*
- *search*

b

c

f

s

U