# cbcpost technical report

***Release 2016.1.0***

**Martin Alnæs and Øyvind Evju**

November 07, 2016

Contents


**1 Introduction** **1**

**2 Installation** **3**
2.1 Dependencies . . . 3
2.2 Installing . . . 3

**3 Features** **5**

**4 Demos** **7**
4.1 A Basic Use Case . . . 7
4.2 Restart a Problem . . . 12
4.3 Replay a Problem . . . 13

**5 Functionality** **15**
5.1 The `Field`-class and subclasses . . . 15
5.2 The postprocessor . . . 19
5.3 Replay . . . 23
5.4 Batch running (cbcbatch) . . . 24
5.5 Dashboard view (cbcdashboard) . . . 25
5.6 Restart . . . 27
5.7 Utilities . . . 28

**6 Overview of available functionality** **31**
6.1 Postprocessor . . . 31
6.2 Replay . . . 32
6.3 Restart . . . 32
6.4 Fields . . . 32
6.5 Parameter system . . . 38
6.6 Other Classes . . . 38
6.7 Other Functions . . . 39
6.8 Utilities . . . 39

**7 Contributing** **43**
7.1 Pull requests . . . 43
7.2 Report problems . . . 43
7.3 Contact developers . . . 44

CHAPTER 1

# Introduction

cbcpost is developed to simplify the postprocessing of simulation results, produced by FEniCS solvers.

The framework is designed to take any given solution, and compute and save any derived data. Derived data can easily be made highly complex, due to the modular design and implementation of computations of quantities such as integrals, derivatives, magnitude etc, and the ability to chain these.

The interface is designed to be simple, with minimal cluttering of a typical solver code. This is illustrated by the following simple example:

```
# ... problem set up ...

# Set up postprocessor
solution = SolutionField("Displacement", dict(save=True))
postprocessor = PostProcessor(dict(casedir="Results/"))
postprocessor.add_field(solution)

# Add derived fields
postprocessor.add_fields([
    Maximum("Displacement", dict(save=True)),
    TimeAverage("Displacement", dict(save=True, start_time=1.0,
                                     end_time=2.0)),
])

t = 0.0
timestep = 0
while t < T:
    timestep += 1
    # ... solve equation ...

    # Update postprocessor
    postprocessor.update_all(dict("Displacement"=lambda: u), timestep, t)

    # continue
```

cbcpost is developed at the Center for Biomedical Computing, at Simula Research Laboratory by Øyvind Evju and Martin Sandve Alnæs.

This pdf-file is generated from rst-files with an incomplete programmers reference. For the updated documentation and programmers reference, see cbcpost.readthedocs.org.

CHAPTER 2

# Installation

## 2.1 Dependencies

The installation of cbcpost requires the following environment:

- Python 2.7
- Numpy
- Scipy
- Any dbm compatible database (dbhash, dbm or gdbm)
- FEniCS
- fenicstools (optional but highly recommended, tools to inspect parts of a solution)

To install FEniCS, please refer to the FEniCS download page.

To install fenicstools, please refer to the github page.

cbcpost and fenicstools follows the same version numbering as FEniCS, so make sure you install the matching versions. Backwards compatibility is not guaranteed (and quite unlikely).

In addition, to run the test suite

- pytest >2.4.0
- docutils

## 2.2 Installing

Get the software with git and install using pip:

```
git clone https://bitbucket.org/simula_cbc/cbcpost.git
cd cbcpost
pip install .
```

See the pip documentation for more installation options.

CHAPTER 3

# Features

The core concept in cbcpost is the `Field`, which represents something that can be computed from simulation solutions or other fields. The main features of cbcpost are

- Saving in 7 different save formats (xdmf, hdf5, xml, xml.gz, pvd, shelve, txt)
- Plotting using dolfin.plot or pyplot
- Automatic planning of field computations, saving and plotting
- Automatic dependency handling
- A range of predefined fields built in, including time integrals, point evaluations and norms
- Easily expandable with custom `Field`-subclasses
- Compute fields during simulation or replay from results on file
- Restart support
- Flexible parameter system
- Small footprint on solver code

CHAPTER 4

# Demos

To get started, we recommend starting with the demos. For explanation of generic FEniCS features, please refer to the official FEniCS documentation.

## 4.1 A Basic Use Case

To demonstrate the functionality of the postprocessor, consider the case of the 3D heat equation with variable diffusivity. The full demo can be found in `Basic.py`.

The general heat equation reads

$$\frac{\partial u}{\partial t} + \alpha(x)\Delta u = f$$

where u typically denotes the temperature and $\alpha$ denotes the material diffusivity.

Boundary conditions are in our example given as

$$u(x,t) = Asin(2\pi t x_0), x \in \partial\Omega$$

and initial condition

$$u(x,0) = 0.$$

We also use f=0, and solve the equations at the unit cube for $t \in (0,3]$.

### 4.1.1 Setting up the problem

We start by defining a set of parameters for our problem:

```
from cbcpost import *
from cbcpost.utils import cbc_print
from dolfin import *
set_log_level(WARNING)

# Create parameters for problem
params = ParamDict(
    T = 3.0,            # End time
    dt = 0.05,          # Time step
    theta = 0.5,        # Time stepping scheme (0.5=Crank-Nicolson)
    alpha0 = 10.0,      # Outer diffusivity
    alpha1 = 1e-3,      # Inner diffusivity
    amplitude = 3.0,    # Amplitude of boundary condition
)
```

The parameters are created using the utility class `ParamDict`, which extends the built-in python dict with dot notation to access values. We use the parameters to set up the problem using FEniCS:

```
# Create mesh
mesh = UnitCubeMesh(21,21,21)

# Function spaces
V = FunctionSpace(mesh, "CG", 1)
u,v = TrialFunction(V), TestFunction(V)

# Time and time-stepping
t = 0.0
timestep = 0
dt = Constant(params.dt)

# Initial condition
U = Function(V)

# Define inner domain
def inside(x):
    return (0.5 < x[0] < 0.8) and (0.3 < x[1] < 0.6) and (0.2 < x[2] < 0.7)

class Alpha(Expression):
    "Variable conductivity expression"
    def __init__(self, alpha0, alpha1):
        self.alpha0 = alpha0
        self.alpha1 = alpha1

    def eval(self, value, x):
        if inside(x):
            value[0] = self.alpha1
        else:
            value[0] = self.alpha0

# Conductivity
alpha = project(Alpha(params.alpha0, params.alpha1), V)

# Boundary condition
u0 = Expression("ampl*sin(x[0]*2*pi*t)", t=t, ampl=params.amplitude)
bc = DirichletBC(V, u0, "on_boundary")

# Source term
f = Constant(0)

# Bilinear form
a = (1.0/dt*inner(u,v)*dx()
    + Constant(params.theta)*alpha*inner(grad(u), grad(v))*dx)
L = (1.0/dt*inner(U,v)*dx()
    + Constant(1-params.theta)*alpha*inner(grad(U), grad(v))*dx()
    + inner(f,v)*dx())
A = assemble(a)
b = assemble(L)
bc.apply(A)
```

### 4.1.2 Setting up the PostProcessor

To set up the use case, we specify the case directory, and asks to clean out the case directory if there is any data remaining from a previous simulation:

```
pp = PostProcessor(dict(casedir="Results", clean_casedir=True))
```

Since we're solving for temperature, we add a SolutionField to the postprocessor:

```
pp.add_field(SolutionField("Temperature", dict(save=True,
                                save_as=["hdf5", "xdmf"],
                                plot=True,
                                plot_args=dict(range_min=-params.amplitude,
                                               range_max=params.amplitude),
                                )))
```

Note that we pass parameters, specifying that the field is to be saved in hdf5 and xdmf formats. These formats are default for dolfin.Function-type objects. We also ask for the Field to be plotted, with plot_args specifying the plot window. These arguments are passed directly to the dolfin.plot-command.

#### Time derivatives and time integrals

We can compute both integrals and derivatives of other Fields. Here, we add the integral of temperature from t=1.0 to t=2.0, the time-average from t=0.0 to t=5.0 as well as the derivative of the temperature field.

```
pp.add_fields([
    TimeIntegral("Temperature", dict(save=True, start_time=1.0,
                                     end_time=2.0)),
    TimeAverage("Temperature", dict(save=True, end_time=params.T)),
    TimeDerivative("Temperature", dict(save=True)),
    ])
```

Again, we ask the fields to be saved. The storage formats are determined by the datatype returned from the *compute*-functions.

#### Inspecting parts of a solution

We can also define fields to inspect parts of other fields. For this, we use some utilities from `cbcpost.utils`. For this problem, the domain of a different diffusivity lies entirely within the unit cube, and thus it may make sense to view some of the interior. We start by creating (sub)meshes of the domains we wish to inspect:

```
from cbcpost.utils import create_submesh, create_slice
celldomains = CellFunction("size_t", mesh)
celldomains.set_all(0)
AutoSubDomain(inside).mark(celldomains, 1)

slicemesh = create_slice(mesh, (0.7,0.5,0.5), (0.0,0.0,1.0))
submesh = create_submesh(mesh, celldomains, 1)
```

We then add instances of the fields `PointEval`, `SubFunction` and `Restrict` to the postprocessor:

```
pp.add_fields([
    PointEval("Temperature", [[0.7,0.5, 0.5]], dict(plot=True)),
    SubFunction("Temperature", slicemesh, dict(plot=True,
        plot_args=dict(range_min=-params.amplitude,
                       range_max=params.amplitude, mode="color"))),
    Restrict("Temperature", submesh, dict(plot=True, save=True)),
    ])
```

### Averages and norms

We can also compute scalars from other fields. `DomainAvg` compute the average of a specified domain (if not specified, the whole domain). Here, we compute the average temperature inside and outside the domain of different diffusivity, as specified by the variable *cell_domains*:

```
pp.add_fields([
    DomainAvg("Temperature", cell_domains=cell_domains,
              indicator=1, label="inner"),
    DomainAvg("Temperature", cell_domains=cell_domains,
              indicator=0, label="outer"),
])
```

The added parameter *label* does that these fields are now identified by *DomainAvg_Temperature-inner* and *DomainAvg_Temperature-inner*, respectively.

We can also compute the norm of any field:

```
pp.add_field(Norm("Temperature", dict(save=True)))
```

If no norm is specified, the L2-norm (or l2-norm) is computed.

### Custom fields

The user may also customize fields with custom computations. In this section we demonstrate two ways to compute the difference in average temperature between the two areas of different diffusivity at any given time. First, we take an approach based solely on accessing the *Temperature*-field:

```
class TempDiff1(Field):
    def __init__(self, domains, ind1, ind2, *args, **kwargs):
        Field.__init__(self, *args, **kwargs)
        self.domains = domains
        self.dx = Measure("dx", domain=self.domains.mesh(),
                          subdomain_data=self.domains)
        self.ind1 = ind1
        self.ind2 = ind2

    def before_first_compute(self, get):
        self.V1 = assemble(Constant(1)*self.dx(self.ind1))
        self.V2 = assemble(Constant(1)*self.dx(self.ind2))

    def compute(self, get):
        u = get("Temperature")
        T1 = 1.0/self.V1*assemble(u*self.dx(self.ind1))
        T2 = 1.0/self.V2*assemble(u*self.dx(self.ind2))
        return T1-T2
```

In this implementation we have to specify the domains, as well as compute the respective averages directly each time. However, since we already added fields to compute the averages in both domains, there is another, much less code-demanding way to do this:

```
class TempDiff2(Field):
    def compute(self, get):
        T1 = get("DomainAvg_Temperature-inner")
        T2 = get("DomainAvg_Temperature-outer")
        return T1-T2
```

Here, we use the provided *get*-function to access the fields named as above, and compute the difference. We add an instance of both to the potsprocessor:

```
pp.add_fields([
    TempDiff1(cell_domains, 1, 0, dict(plot=True)),
    TempDiff2(dict(plot=True)),
])
```

Since both these should be the same, we can check this with `ErrorNorm`:

```
pp.add_field(
    ErrorNorm("TempDiff1", "TempDiff2", dict(plot=True), name="error"),
)
```

We ask for the error to be plotted. Since this is a scalar, this will be done using matplotlibs *pyplot*-module. We also pass the keyword argument *name*, which overrides the default naming (which would have been ErrorNorm_TempDiff1_TempDiff2) with *error*.

#### Combining fields

Finally, we can also add combination of fields, provided all dependencies have already been added to the postprocessor. For example, we can compute the space average of a time-average of our field *Restrict_Temperature* the following way:

```
pp.add_fields([
    TimeAverage("Restrict_Temperature"),
    DomainAvg("TimeAverage_Restrict_Temperature", params=dict(save=True)),
])
```

If *TimeAverage("Restrict_Temperature")* is not added first, adding the `DomainAvg`-field would fail with a `DependencyException`, since the postprocessor would have no knowledge of the field *TimeAverage_Restrict_Temperature*.

#### Saving mesh and parameters

We choose to store the mesh, domains and parameters associated with the problem:

```
pp.store_mesh(mesh, cell_domains=cell_domains)
pp.store_params(params)
```

These will be stored to *mesh.hdf5*, *params.pickle* and *params.txt* in the case directory.

### 4.1.3 Solving the problem

Solving the problem is done very simply here using simple FEniCS-commands:

```
solver = KrylovSolver(A, "cg", "hypre_amg")
while t <= params.T+DOLFIN_EPS:
    cbc_print("Time: "+str(t))
    u0.t = float(t)

    assemble(L, tensor=b)
    bc.apply(b)
    solver.solve(U.vector(), b)

    # Update the postprocessor
    pp.update_all({"Temperature": lambda: U}, t, timestep)

    # Update time
    t += float(dt)
    timestep += 1
```

Note the single call to the postprocessor, *pp.update_all*, which will then execute the logic for the postprocessor. The solution *Temperature* is passed in a dict as a lambda-function. This lambda-function gives the user flexibility to process the solution in any way before it is used in the postprocessor. This can for example be a scaling to physical units or joining scalar functions to a vector function.

Finally, at the end of the time-loop we finalize the postprocessor through

```
pp.finalize_all()
```

This command will finalize and return values for fields such as for example time integrals.

## 4.2 Restart a Problem

Say we wish to run our simulation further than t=3.0, to see how it develops. To restart a problem, all you need is to use the computed solution as initial condition in a similar problem setup.

Restarting the heat equation solved as in *A Basic Use Case*, can be done really simple with cbcpost. Starting with the python file in *A Basic Use Case*, we only have to make a couple of minor changes.

We change the parameters T0 and T to look at the interval $t \in [3, 6]$:

```
params.T0 = 3.0
params.T = 6.0
```

and we replace the initial condition, using the `Restart`-class:

```
# Get restart data
restart = Restart(dict(casedir="../Basic/Results/"))
restart_data = restart.get_restart_conditions()

# Initial condition
U = restart_data.values()[0]["Temperature"]
```

Note that we point `Restart` to the case directory where the solution is stored. We could also choose to write our restart data to the same directory when setting up the postprocessor:

```
pp = PostProcessor(dict(casedir="../Basic/Results"))
```

## 4.3 Replay a Problem

Once a simulation is completed, one might want to compute other fields of the solution. This can be done with cbcposts `Replay`-functionality. The process can be done in very few lines of code.

In the following, we initialize a replay of the heat equation solved in *A Basic Use Case* and restarted in *Restart a Problem*. First, we set up a postprocessor with the fields we wish to compute:

```
from cbcpost import *
from dolfin import set_log_level, WARNING, interactive
set_log_level(WARNING)

pp = PostProcessor(dict(casedir="../Basic/Results"))

pp.add_fields([
    SolutionField("Temperature", dict(plot=True)),
    Norm("Temperature", dict(save=True, plot=True)),
    TimeIntegral("Norm_Temperature", dict(save=True, start_time=0.0,
                                                                    end_ti
])
```

To *replay* the simulation, we do:

```
replayer = Replay(pp)
replayer.replay()
interactive()
```

CHAPTER 5

# Functionality

The main functionality is handled with a `PostProcessor`-instance, populated with several `Field`-items.

The `Field`-items added to the `PostProcessor` can represent *meta* computations (`MetaField`, `MetaField2`) such as time integrals or time derivatives, restrictions or subfunction, or norms. They can also represent custom computations, such as stress, strain, stream functions etc. All subclasses of the `Field`-class inherits a set of parameters used to specify computation logic, and has a set of parameters related to saving, plotting, and computation intervals.

The `Planner`, instantiated by the PostProcessor, handles planning of computations based on Field-parameters. It also handles the dependency, and plans ahead for computations at a later time.

For saving purposes the PostProcessor also creates a `Saver`-instance. This will save Fields as specified by the Field-parameters and computed fields. It saves in a structured manner within a specified case directory.

In addition, there is support for plotting in the `Plotter`-class, also created within the PostProcessor. It uses either dolfin.plot or pyplot.plot to plot data, based on the data format.

## 5.1 The `Field`-class and subclasses

To understand how cbcpost works, one first needs to understand the role of *Fields*. All desired postprocessing must be added to the PostProcessor as subclasses of `Field`. The class itself is to be considered as an abstract base class, and must be subclassed to make sense.

All subclasses are expected to implement (at minimum) the `Field.compute()`-method. This takes a single argument which can be used to retrieve dependencies from other fields.

An important property of the `Field`-class, is the parameters. Through the `Parameterized`-interface, it implements a set of default parameters that is used by the PostProcessor when determining how to handle any given Field, with respect to computation frequency, saving and plotting.

### 5.1.1 Subclassing the `Field`-class

To compute any quantity of interest, one needs to either use one of the provided metafields or subclass `Field`. In the following, we will first demonstrate the simplicity of the interface, before demonstrating the flexibility of it.

### A viscous stress tensor

The viscous stress tensor for a Newtonian fluid is computed as

$$\sigma(\mathbf{u}, p) = -p\mathbb{I} + \mu(\nabla\mathbf{u} + \nabla\mathbf{u}^T)$$

where $\mu$ is the dynamic viscosity, $\mathbf{u}$ is the fluid velocity and $p$ is the pressure. A Field to compute this might be specified as the following:

```
from dolfin import *
from cbcpost import Field
from cbcpost.spacepool import get_grad_space
class Stress(Field):
    def __init__(self, mu, params=None, name="default", label=None):
        Field.__init__(self, params, name, label)
        self.mu = mu

    def before_first_compute(self, get):
        u = get("Velocity")

        # Create Function container on space of velocity gradient
        V = get_grad_space(u)
        self._function = Function(V, name=self.name)

    def compute(self, get):
        u = get("Velocity")
        p = get("Pressure")
        mu = self.mu

        expr = - p*Identity(u.cell().d) + mu*(grad(u)+grad(u)^T)

        return self.expr2function(expr, self._function)
```

Note that we have overridden three methods defined in `Field`:

- __init__
- before_first_compute
- compute

The __init__ method is only used to pass any additional arguments to our Field, in this case the viscosity. The keyword arguments *params*, *name* and *label* are passed directly to `Field.__init__()`.

before_first_compute is used to do any costly computations or allocations that are only required once. This is called from the postprocessor before any calls to compute is made. In this case we create a container (*_function*) that we can later use to store our computations. We use the *get*-argument to fetch the field named *Velocity*, and the helper function `get_grad_space()` to get the gradient space of the Velocity (a TensorFunctionSpace).

The compute method is responsible for computing our quantity. This is called from the postprocessor every time the `Planner` determines that this field needs to be computed. Here we use the *get*-argument to fetch the *Velocity* and *Pressure* required to compute the stress. We formulate the stress, and converts to a function using the helper function `Field.expr2function()`.

### Computing the maximum pressure drop

In this next section, we demonstrate some more functionality one can take advantage of when subclassing the `Field`-class. In a flow, the maximum pressure drop gives an indication of the forces involved in the flow. It can be written as

$$\tilde{p} := \max_{t\in[0,T]} \left( \max_{\mathbf{x}\in\Omega} p(\mathbf{x},t) - \min_{\mathbf{x}\in\Omega} p(\mathbf{x},t) \right)$$

A `Field`-class to compute this can be implemented as

```
from dolfin import *
from cbcpost import Field
from cbcpost.spacepool import get_grad_space
class PTilde(Field):
    def add_fields(self):
        return [ Maximum("Pressure"), Minimum("Pressure") ]

    def before_first_compute(self, get):
        self._ptilde = 0.0
        self._tmax = 0.0

    def compute(self, get):
        pmax = get("Maximum_Pressure")
        pmin = get("Minimum_Pressure")
        t = get("t")

        if pmax-pmin > self._ptilde:
            self._ptilde = pmax-pmin
            self._tmax = t

        return None

    def after_last_compute(self, get):
        return (self._ptilde, self._tmax)
```

Here, we implement two more `Field`-methods:

- add_fields
- after_last_compute

The add_fields method is a convenience function to make sure that dependent Fields are added to the postprocessor. This can also be handled manually, but this makes for a cleaner code. Here we add two fields to compute the (spatial) `Maximum` and `Minimum` of the pressure.

The method after_last_compute is called when the compution is finished. This is determined by the time parameters (see *Parameters*), and handled within the postprocessors `Planner`-instance.

## 5.1.2 Field names

The internal communication of fields is based on the name of the `Field`-instances. The default name is

```
[class name]-[optional label]
```

The label can be specified in the *__init__*-method (through the *label*-keyword), or a specific name can be set using the *name*-keyword.

When subclassing the `Field`-class, the default naming convention can overloaded in the `Field.name`-property.

### 5.1.3 The *get*-argument

In the three methods *before_first_compute*, *compute* and *after_last_compute* a single argument (in addition to *self*) is passed from the postprocessor, namely the *get*-argument. This argument is used to fetch the computed value from other fields, through the postprocessor. The argument itself points to the `PostProcessor.get()`-method, and is typically used with these two arguments:

- Field name
- Relative timestep

A call using the *get*-function will trigger a computation of the field with the given name, and cache it in the postprocessor. Therefore, a second call with the same arguments, will return the cached value and not trigger a new computation.

The calls to the *get*-function also determines the dependencies of a Field (see *Dependency handling*).

### 5.1.4 Parameters

The logic of the postprocessor relies on a set of parameters defined on each Field. For explanation of the common parameters and their default, see `Field.default_params()`.

### 5.1.5 SolutionField

The `SolutionField`-class is a convenience class, for specifying Field(s) that will be provded as solution variables. It requires a single argument as the name of the Field. Since it is a solution field, it does not implement it does not implement a *compute*-method, but relies on data passed to the `PostProcessor.update_all()` for its associatied data. It is used to be able to build dependencies in the postprocessor.

### 5.1.6 MetaField and MetaField2

Two additional base classes are also available. These are designed to allow for computations that are not specific (such as PTilde or Stress), but where you need to specify the Field(s) to compute on.

Subclasses of the `MetaField`-class include for example `Maximum`, `Norm` and `TimeIntegral`, and takes a single name (or Field) argument to specify which Field to do the computation on.

Subclasses of the `MetaField2` include `ErrorNorm`, and takes two name (or Field) arguments to specify which Fields to compute with.

### 5.1.7 Provided fields

Several meta fields are provided in cbcpost, for general computations. These are summarized in the following table:

| Time dependent | Spatially restricted | Norms and averages | Other |
|---|---|---|---|
| TimeDerivative | SubFunction | DomainAvg | Magnitude |
| TimeIntegral | Restrict | Norm | |
| TimeAverage | Boundary PointEval | ErrorNorm Maximum Minimum | |

For more details of each field, refer to metafields.

## 5.2 The postprocessor

The `PostProcessor`-class is responsible for all the logic behind the scenes. This includes logic related to:

- Dependency handling
- Planning and caching of computation
- Saving
- Plotting

The planning, saving and plotting is delegated to dedicated classes (`Planner`, `Saver` and `Plotter`), but is called from within a `PostProcessor`-instance.

### 5.2.1 The *update_all*-function

The main interface to the user is through the `PostProcessor.update_all()`-method. This takes three arguments: a *dict* representing the solution, the solution time and the solution timestep.

The time and timestep is used for saving logic, and stored in a *play log* and metadata of the saved data. This is necessary for the replay and restart functionality, as well as order both the saved and plotted fields.

The solution argument should be of the format:

```
solution = dict(
    "Velocity": lambda: u
    "Pressure": lambda: p
)
```

Note that we pass a lambda function as values in the dict. This is done to give the user the flexibility for special solvers, and can be replaced with any callable to do for example a conversion. This can be useful when there are discrepancies between the solver solution, and the desired *physical* solution. This could be for example a simple scaling, or it could be that a mixed or segregated approach is used in the solver.

Because this function might be non-negligible in cost, it will be treated in the same manner as the `Field.compute()`-method, and not called unless required.

### 5.2.2 Dependency handling

When a field is added to the postprocessor, a dependency tree is built. These dependencies represent the required fields (or time parameters) required to succesfully execute the *compute*-method.

The source code of the *compute*-function is inspected with the *inspect*-module, by looking for calls through the *get*-argument, and build a dependency tree from that.

Assume that the following code is executed:

```
pp = PostProcessor()
pp.add_field(SolutionField("F"))
pp.add_field(TimeDerivative("F"))
```

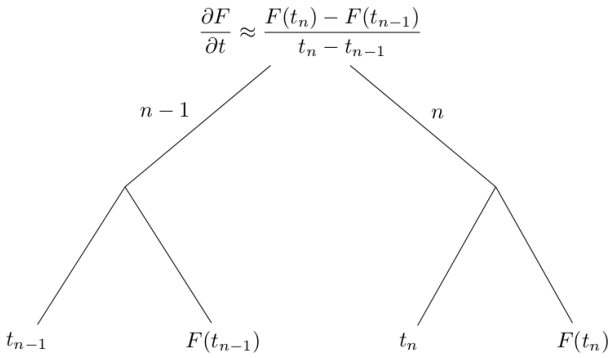
In that case, when the `TimeDerivative`-field is added to the postprocessor, the following code is inspected:

```
class TimeDerivative(MetaField):
    def compute(self, get):
        u1 = get(self.valuename)
        u0 = get(self.valuename, -1)

        t1 = get("t")
        t0 = get("t", -1)

        # ... [snip] ...
```

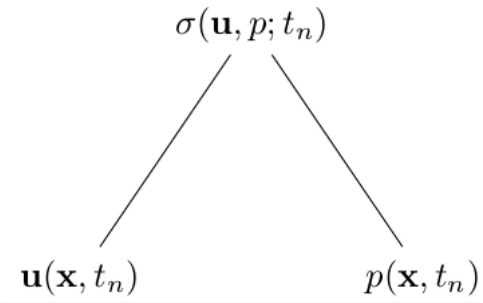
By evaluating the *get*-calls here, we are able to build the following dependency tree:



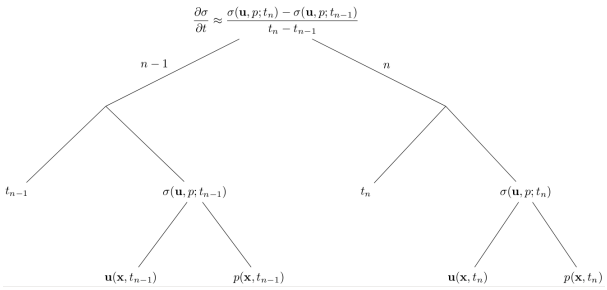

If we extend the above example to add the time derivative of the viscous stress tensor (see *A viscous stress tensor*) like the following:

```
pp = PostProcessor()
pp.add_fields([SolutionField("Velocity"), SolutionField("Pressure")])
pp.add_field(Stress())
pp.add_field(TimeDerivative("Stress"))
```

The first emphasized line will trigger building of the dependency tree for the stress:




while the second emphasized line will use this dependency tree, and trigger the building of the larger dependency tree




### 5.2.3 Planner

The `Planner`-class will set up a plan of the computations for the coming timesteps. This algorithm will inspect the dependencies of each field, and compute the necessary fields at the required time.

In addition, it determines how long each computation should be kept in cache.

**Note:** This does not yet support variable timestepping.

### 5.2.4 Saver

The `Saver`-class handles all the saving operations in cbcpost. It will determine if and how to save based on Field-parameters. In addition, there are helper methods in `PostProcessor` for saving mesh and parameters.

For fields, several saveformats are available:

| Replay/restart-compatible | Visualization | Plain text |
|---|---|---|
| hdf5 | xdmf | txt |
| xml | pvd | |
| xml.gz | | |
| shelve | | |

The default save formats are:

- *hdf5* and *xdmf* if data is dolfin.Function
- *txt* and *shelve* if data is float, int, list, tuple or dict

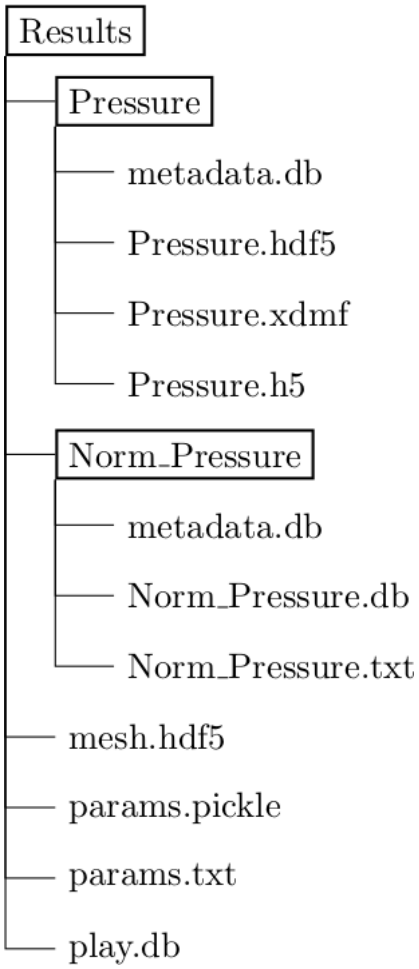
The saving is done in a structured manner below the postprocessors case director. Consider the following example:

```
pp = PostProcessor(dict(casedir="Results/"))
pp.add_fields([
    SolutionField("Pressure", save=True),
    Norm("Pressure", save=True),
])
pp.store_mesh(mesh, facet_domains=my_facet_domains,
              cell_domains=my_cell_domains)
pp.store_params(
    ParamDict(
        mu = 1.5,
        case = "A",
        bc = "p(0)=1",
```

```
    )
)
```

Here, we ask the postprocessor to save the Pressure and the (L2-)norm of the pressure, we store the mesh with associated cell- and facet domains, and we save some (arbitrary) parameters. (Note the use of `ParamDict`).

This will result in the following structure of the *Results*-folder:




### 5.2.5 Plotter

Two types of data are supported for plotting:

- dolfin.Function-type objects
- Scalars (int, float, etc)

The `Plotter`-class plots using dolfin.plot or pyplot.plot depending on the input data. The plotting is updated each timestep the Field is directly triggered for recomputation, and rescaled if necessary. For dolfin plotting, arguments can be passed to the dolfin.plot-command through the parameter *plot_args*.

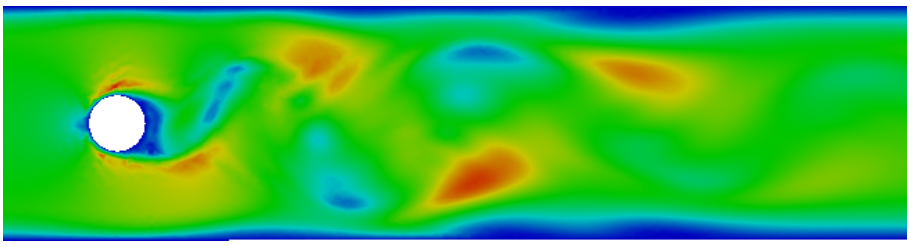


Fig. 5.1: dolfin.Function objects are plotted with dolfin.plot

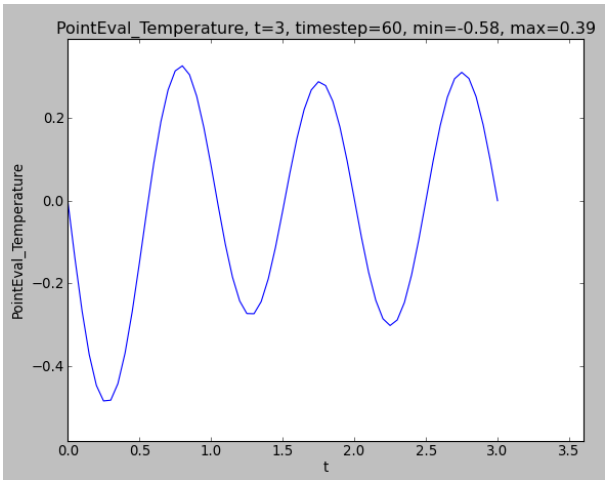



Fig. 5.2: Simple scalars are plotted with pyplot

## 5.3 Replay

One of the key functionalities of the cbcpost framework is the ability to replay problem. Consider the case where one wants to extract additional information from a simulation. Simulations are typically costly, and redoing simulations are not generally desired (or even feasible). This motivates the functionality to *replay* the simulation by loading the computed solution back into memory and compute additional fields.

This has several major benefits:

- Compute additional quantities
- Limit memory consumption of initial computation
- Compute quantities unsupported in parallel
- Compute costly, conditional quantities (e.g. not to be performed if simulation was unable to complete)
- Create visualization data

The interface to the replay module is minimal:

```
from cbcpost import PostProcessor, Replay

pp = PostProcessor(dict(casedir="ExistingResults/"))
pp.add_field(MyCustomField(), dict(save=True))

replayer = Replay(pp)
```

```
replayer.replay()
```

In the replay module, all fields that are stored in a reloadable format will be treated as a solution. They will be passed to a postprocessor as instances of the `Loadable`-class. This makes sure that no unnecessary I/O-operations occur, as the stored data are only loaded when they are triggered in the postprocessor.

## 5.4 Batch running (cbcbatch)

When you've set up a simulation in a python file, you can investigate a range of parameters through the shell script *cbcbatch*. This allows you to easily run simulations required to for example compute convergence rates or parameter sensitivity with respect to some compute Field.

Based on the parameters of your solver or problem, you can set up parameter ranges with command line arguments to cbcbatch. Say for example that you wish to investigate the effects of refinement level *N* and timestep *dt* over a given range. Then you can launch cbcbatch by invoking

```
cbcbatch run.py N=[8,16,32,64,128] dt=[0.1,0.05,0.025,0.0125] \
    casedir=BatchResults
```

where run.py is the python file to launch the simulation. This will then add all combinations of N and dt (5*4=20) to a queue, and launch simulations when the resources are available. We call *dt* and *N* the *batch parameters*.

By default, cbcbatch runs on a single core, but this can be modified by setting the *num_cores* argument:

```
cbcbatch run.py N=[8,16,32,64,128] dt=[0.1,0.05,0.025,0.0125] \
    casedir=BatchResults num_cores=8
```

This will cause 8 simulations to be run at a time, and new ones started as soon as one core becomes available. Since there may be a large variations in computational cost between parameters, it is also supported to tie one of the batch parameters to run in parallel with mpirun:

```
cbcbatch run.py N=[8,16,32,64,128] dt=[0.1,0.05,0.025,0.0125] \
    casedir=BatchResults num_cores=8 mpirun=[1,1,2,4,8] \
    mpirun_parameter=N
```

This command will run all simulations with N=1 and N=2 on a single core, N=32 on 2 cores, N=64 on 4 cores and N=128 on 8 cores.

---

**Important:** The runnable python file must set *set_parse_command_line_arguments(True)* to be run in batch mode.

---

---

**Important:** The command line parameters *casedir*, *mpirun*, *mpirun_parameter* and *num_cores* are reserved for cbcbatch and can thus not be used as batch parameters.

---

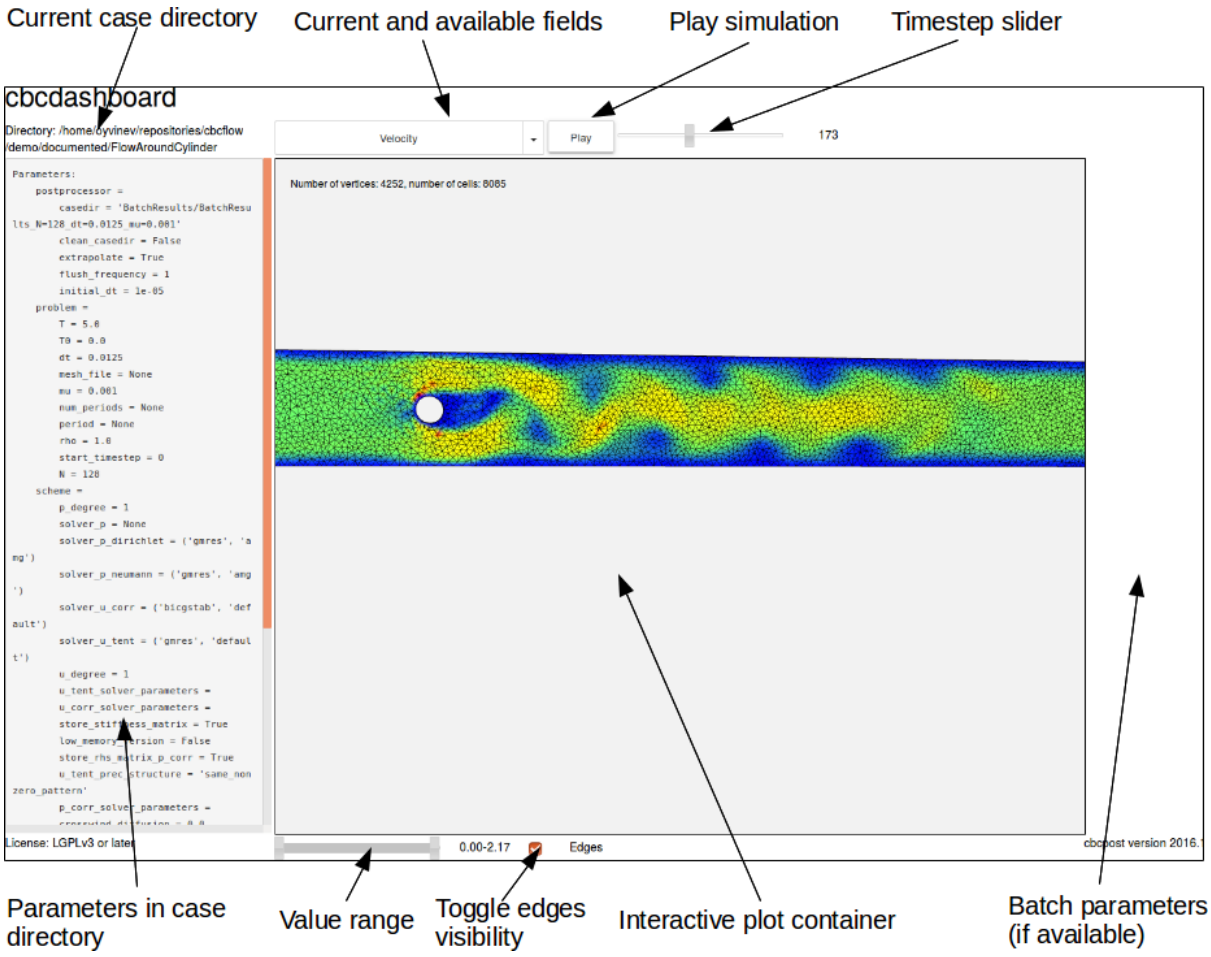
## 5.5 Dashboard view (cbcdashboard)

To easily view the results of a simulation postprocessed with cbcpost, you can use *cbcdashboard*. This is a command line scripts that launch a jupyter notebook server, and opens a notebook containing a GUI to investigate the provided case directory. The case directory can either be a directory for a single simulation, or containing results from a *cbcbatch*-run.

To view results launch the dashboad like this:

```
cbcdashboard casedir
```

execute the first cell of the notebook, the GUI will launch and show you the available simulation results, here with denoted widget areas:



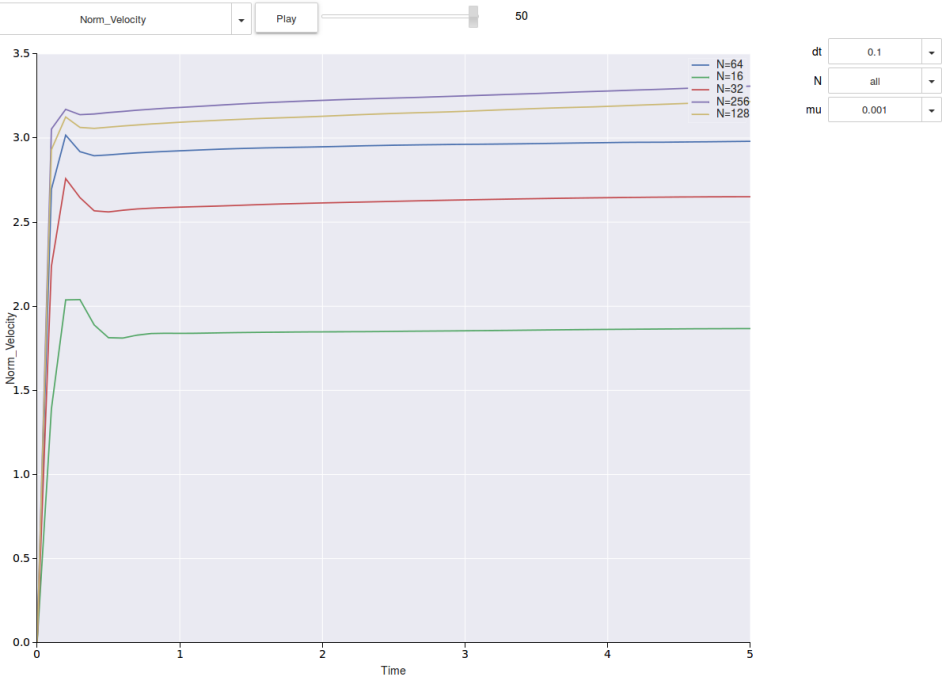

The interactive plot is an HTML-container that change based on the selected value, and can show both dolfin Functions (through X3DOM), time-dependent scalars and vectors (through matplotlib and mpld3.figure_to_html), constant scalars and, in the case of batch directories, tables of constant scalars (through pandas.DataFrame.to_html).

Showing batch results can be done by pointing cbcdashboard to a directory created with *cbcbatch* and containing the results of that batch simulations. This will launch a slightly different GUI, where you have the ability to select single batch parameters or set one or two batch parameter to *all*. When a batch parameter is set to *all*, it will show the fields available for comparison between the simulations, allowing for detailed inspection of parameter sensitivity:

## cbcdashboard

Directory: /home/oyvinev/repositories/cbcflow
/demo/documented/FlowAroundCylinder

```
Parameters:
    BatchParams =
        casedir = 'BatchResults'
        num_cores = 4
    Params =
        dt = (0.1, 0.05, 0.025, 0.0125)
        N = (16, 32, 64, 128, 256)
        mu = (0.001, 0.01, 0.1, 1.0)
```




License: LGPLv3 or later

cbcpost version 2016.1

## cbcdashboard

Directory: /home/oyvinev/repositories/cbcflow
/demo/documented/FlowAroundCylinder

```
Parameters:
    BatchParams =
        casedir = 'BatchResults'
        num_cores = 4
    Params =
        dt = (0.1, 0.05, 0.025, 0.0125)
        N = (16, 32, 64, 128, 256)
        mu = (0.001, 0.01, 0.1, 1.0)
```


| N\mu | 0.001 | 0.01 | 0.1 | 1.0 |
|---|---|---|---|---|
| 16 | 0.026 | 0.019 | 0.013 | 0.013 |
| 32 | 0.121 | 0.132 | 0.136 | 0.136 |
| 64 | 0.120 | 0.157 | 0.172 | 0.173 |
| 128 | 0.156 | 0.169 | 0.188 | 0.190 |
| 256 | 0.155 | 0.174 | 0.196 | 0.197 |

License: LGPLv3 or later

cbcpost version 2016.1

By specifying all parametersm you can investigate the solutions directly, similar to what you can if specifying a single case directory.

Currently, you can only change the browser to open the notebook in from the command line, by passing *browser=my-favourite-browser* to cbcdashboard.

## 5.6 Restart

The restart functionality lets the user set up a problem for restart. This functionality is based on the idea that a restart of a simulation is nothing more than changing the initial conditions of the problem in question. Therefore, the `Restart`-class is used to extract the solution at any given time(s) in a format that may be used as intiial conditions.

If we want to restart any problem, where a solution has been stored by cbcpost, we can simply point to the case directory:

```
from cbcpost import *
restart = Restart(dict(casedir='Results/'))
restart_data = restart.get_restart_conditions()
```

If you for instance try to restart the simple case of the heat equation, *restart_data* will be a *dict* of the format {t0: {"Temperature": U0}}. If you try to restart for example a (Navier-)Stokes-problem, it will take a format of {t0: {"Velocity": U0, "Pressure": P0}}.

There are several options for fetching the restart conditions.

### 5.6.1 Specify restart time

You can easily specify the restart time to fetch the solution from:

```
t0 = 2.5
restart = Restart(dict(casedir='Results/', restart_times=t0))
restart_data = restart.get_restart_conditions()
```

If the restart time does not match a solution time, it will do a linear interpolation between the closest existing solution times.

### 5.6.2 Fetch multiple restart times

For many problems, initial conditions are required at several time points prior to the desired restart time. This can be handled through:

```
dt = 0.01
t1 = 2.5
t0 = t1-dt
restart = Restart(dict(casedir='Results/', restart_times=[t0,t1]))
restart_data = restart.get_restart_conditions()
```

### 5.6.3 Rollback case directory for restart

If you wish to write the restarted solution to the same case directory, you will need to clean up the case directory to avoid write errors. This is done by setting the parameter *rollback_casedir*:

```
t0 = 2.5
restart = Restart(dict(casedir='Results/', restart_times=t0,
                       rollback_casedir=True))
restart_data = restart.get_restart_conditions()
```

### 5.6.4 Specifying solution names to fetch

By default, the Restart-module will search through the case directory for all data stored as a `SolutionField`. However, you can also specify other fields to fetch as restart data:

```
solution_names = ["MyField", "MyField2"]
restart = Restart(dict(casedir='Results/', solution_names=solution_names))
restart_data = restart.get_restart_conditions()
```

In this case, all `SolutionField`-names will be ignored, and only restart conditions from fields named *MyField* and *MyField2* will be returned.

### 5.6.5 Changing function spaces

If you wish to restart the simulation using different function spaces, you can pass the function spaces to *get_restart_conditions*:

```
V = FunctionSpace(mesh, "CG", 3)
restart = Restart(dict(casedir='Results/'))
restart_data = restart.get_restart_conditions(spaces={"Temperature": V})
```

**Note:** This does not currently work for function spaces defined on a different mesh.

## 5.7 Utilities

A set of utilities are provided with cbcpost. Below are just a few of them. For a more complete set of utilities, refer to the programmers-reference.

### 5.7.1 The `ParamDict`-class

The `ParamDict`-class extends to the stadard python *dict*. It supports dot-notation (*mydict["key"] == mydict.key*), and nested parameters.

**Todo**

Extend this documentation.

### 5.7.2 The `Parameterized`-class

The `Parameterized`-class is sued for classes that are associated with a set of parameters. All subclasses must implement the method `Parameterized.default_params()`, which return a ParamDict/dict with default values for the parameters.

When initialized, it takes a *params*-option where specific parameters are set, and overwriting the associated parameters returned from *default_params*. This is then stored in an attribute *params* attached to the object.

When initializing a Parameterized-object, no new keys are allowed. This means that all parameters of a Parameterized-instance must be defined with default values in *default_params*.

The class is subclasses several places within cbcpost:

- `Field`
- `PostProcessor`
- `Restart`
- `Replay`

### 5.7.3 Pooling of function spaces

When using many different functions across a large function, it may be useful to reuse FunctionSpace definitions. This has two basic advantages:

- Reduced memory consumption
- Reduced computational cost

Space pools are grouped according to mesh, with *Mesh.id()* used as keys in a *weakref.WeakValueDictionary*. Once a mesh is out of focus in the program, the related SpacePool is removed.

### 5.7.4 Submesh creation

The `SubMesh`-class in dolfin is not currently supported in dolfin. In cbcpost, the function `create_submesh()` is the equivalent functionality, but with parallel support.

This allows for arbitrary submeshes in parallel based by providing a MeshFunction and marker.

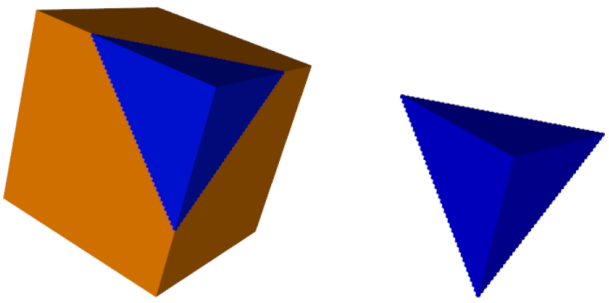


Fig. 5.3: Submesh created with *create_submesh* in cbcpost.

### 5.7.5 Mesh slicing

Three-dimensional meshes can be sliced in cbcpost with the Slice-class. The `Slice`-class takes basemesh, togetther with a point and normal defining the slicing plane, to create a slicemesh.

The `Slice`-class is a subclass of dolfin.Mesh.

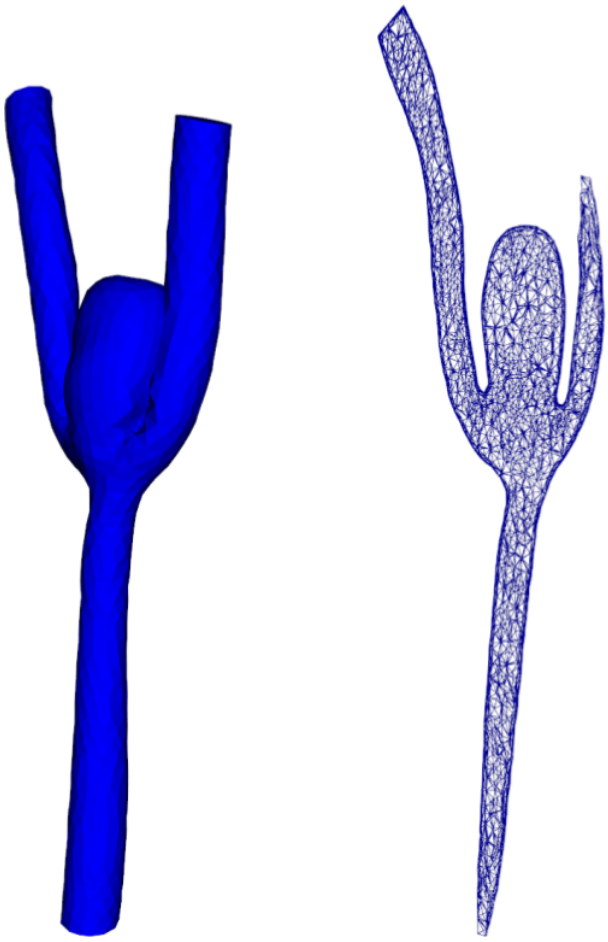


Fig. 5.4: A complex 3D-mesh, with an associated slicemesh.

**Warning:** Slice-instances are intended for visualization only, and may produce erronous results if used for computations.

CHAPTER 6

# Overview of available functionality

## 6.1 Postprocessor

### 6.1.1 PostProcessor

All basic user interface is gathered here.

Default parameters are:

| Key | Default value | Description |
|---|---|---|
| casedir | '.' | Case directory - relative path to use for saving |
| extrapolate | True | Constant extrapolation of fields prior to first update call |
| initial_dt | 1e-5 | Initial timestep. Only used in planning algorithm at first update call. |
| clean_casedir | False | Clean out case directory prior to update. |
| flush_frequency | 1 | Frequency to flush shelve and txt files (playlog, metadata and data) |

### 6.1.2 Planner

Planner class to plan for all computations.

### 6.1.3 Saver

Class to handle all saving in cbcpost.

### 6.1.4 Plotter

Class to handle plotting of objects.

Plotting is done using pylab or dolfin, depending on object type.

## 6.2 Replay

### 6.2.1 Replay

Replay class for postprocessing exisiting solution data.

Default parameters are:

| Key | Default value | Description |
|---|---|---|
| check_memory_frequency | 0 | Frequency to report memory usage |

## 6.3 Restart

### 6.3.1 Restart

Class to fetch restart conditions through.

Default parameters are:

| Key | Default value | Description |
|---|---|---|
| casedir | '.' | Case directory - relative path to read solutions from |
| restart_times | -1 | float or list of floats to find restart times from. If -1, restart from last available time. |
| solution_names | 'default' | Solution names to look for. If 'default', will fetch all fields stored as SolutionField. |
| rollback_casedir | False | Rollback case directory by removing all items stored after largest restart time. This allows for saving data from a restarted simulation in the same case directory. |

## 6.4 Fields

### 6.4.1 Field bases

#### ConstantField

Class for setting constant values. Helpful in dependency inspection.

#### Field

Base class for all fields.

Arguments:

**name** Specify name for field. If default, a name will be created based on class-name.

**label** Specify a label. The label will be added to the name, if name is default.

Default params are:

| Key | Default value | Description |
|---|---|---|
| start_timestep | -1e16 | Timestep to start computation |
| end_timestep | 1e16 | Timestep to end computation |
| stride_timestep | 1 | Number of steps between each computation |
| start_time | -1e16 | Time to start computation |
| end_time | 1e16 | Time to end computation |
| stride_time | 1e-16 | Time between each computation |
| plot | False | Plot Field after a directly triggered computation |
| plot_args | {} | Keyword arguments to pass to dolfin.plot. |
| safe | True | Trigger safe computation. This allows get-calls to this field outside postprocessor. Set to False to rely on postprocessor and improve efficiency. |
| save | False | Save Field after a directly triggered computation |
| save_as | 'determined by data' | Format(s) to save in. Allowed save formats:<br>The default values are:<br>• ['hdf5', 'xdmf'] if data is dolfin.Function<br>• ['txt', 'shelve'] if data is float, int, list, tuple or dict |
| expr2function | 'assemble' | How to convert Expression to Function. Allowed values:<br>• 'assemble'<br>• 'project'<br>• 'interpolate' |
| finalize | False | Switch whether to finalize if Field. This is especially useful when a costly computation is only interesting at the end time. |

### MetaField

Base class for all Fields that operate on a different Field.

Arguments:

**value** Field or fieldname to operate on

### MetaField2

Base class for all Fields that operate on two different Fields.

Arguments:

**value1** First Field or fieldname to operate on

**value2** Second Field or fieldname to operate on

#### SolutionField

Helper class to specify solution variables to the postprocessor.

Arguments:

**name** Name of the solution field

This field can be added to the postprocessor, although it does not implement a *compute*-method. A solution with the same name is expected to be passed to the *PostProcessor.update_all*-method.

#### Time

Compute the time spent between before_first_compute and after_last_compute calls.

Useful for crude time measuring.

### 6.4.2 Operators

#### Add

Add two fields

#### Divide

Divide two fields

#### Multiply

Multiply two fields

#### OperatorField

Base class for all operators on fields

#### Subtract

Subtract two fields

### 6.4.3 MetaFields

#### Boundary

Extracts the boundary values of a Function and returns a Function object living on the equivalent FunctionSpace on boundary mesh.

Only CG1 and DG0 spaces currently functioning.

#### DomainAvg

Compute the domain average for a specified domain. Default to computing the average over the entire domain.

Parameters used to describe the domain are:

Arguments:

**measure** Measure describing the domain (default: dx())

**cell_domains** A CellFunction describing the domains

**facet_domains** A FacetFunction describing the domains

**indicator** Domain id corresponding to cell_domains or facet_domains

If cell_domains/facet_domains and indicator given, this overrides given measure.

#### DomainSD

Compute the domain standard deviation for a specified domain. Default to computing the standard devitation over the entire domain.

Parameters used to describe the domain are:

Arguments:

**measure** Measure describing the domain (default: dx())

**cell_domains** A CellFunction describing the domains

**facet_domains** A FacetFunction describing the domains

**indicator** Domain id corresponding to cell_domains or facet_domains

If cell_domains/facet_domains and indicator given, this overrides given measure.

#### Dot

Compute the dot product between two fields

### ErrorNorm

Computes the error norm of two Fields. If the Fields Function-objects, the computation is forwarded to the dolfin function *errornorm*. Otherwise two float list-type object is expected, and the $l^p$-norm is computed as

$$||\mathbf{x}-\mathbf{y}||_p := \left(\sum_i = 1^n |x_i - y_i|^p\right)^{1/p}.$$

The $\infty$-norm is computed as

$$||\mathbf{x}-\mathbf{y}||_\infty := max(|x_1 - y_1|, |x_2 - y_2|, ..., |x_n - y_n|)$$

Default parameters are:

| Key | Default value | Description |
|---|---|---|
| norm_type | 'default' | The norm type to choose. For dolfin.Function or dolfin.Vector, refer to dolfin.norm for valid norm types. Otherwise, p-norm is supported. Invoke using 'l2', 'l3' etc, or 'linf' for max-norm. |
| degree_rise | 3 | Parameter to be passed to dolfin.errornorm |
| relative | False | Divide norm by the norm of first field |

### Magnitude

Compute the magnitude of a Function-evaluated Field.

Supports function spaces where all subspaces are equal.

### Maximum

Computes the maximum of a Field.

### Minimum

Computes the minimum of a Field.

### Norm

Computes a norm of a Field. If the Field returns a Vector or Function, the computation is forwarded to the dolfin function *norm*. Otherwise a float list-type object is expected, and the $l^p$-norm is computed as

$$||\mathbf{x}||_p := \left(\sum_i = 1^n |x_i|^p\right)^{1/p}.$$

The $\infty$-norm is computed as

$$||\mathbf{x}||_\infty := max(|x_1|, |x_2|, ..., |x_n|)$$

Default parameters are:

| Key | Default value | Description |
|---|---|---|
| norm_type | 'default' | The norm type to choose. For dolfin.Function or dolfin.Vector, refer to dolfin.norm for valid norm types. Otherwise, p-norm is supported. Invoke using 'l2', 'l3' etc, or 'linf' for max-norm. |

### PointEval

Evaluate a Field in points.

Arguments:

**points** List of Points or tuples

This field requires fenicstools.

Default parameters are:

| Key | Default value | Description |
|---|---|---|
| broadcast_results | True | Broadcast results from compute to all processes. If False, result is ony returned on process 0 |

### Restrict

Restrict is used to restrict a Field to a submesh of the mesh associated with the Field.

This has only been tested for CG spaces and DG spaces of degree 0.

### SubFunction

SubFunction is used to interpolate a Field on a non-matching mesh.

This field requires fenicstools.

### Threshold

Compute a new Function based on input function and input threshold. Returned Function is 1 where above/below threshold, and 0 otherwise.

Default parameters are:

| Key | Default value | Description |
|---|---|---|
| threshold_by | "below" | Set the function to threshold "above" or "below" threshold function |

#### TimeAverage

Compute the time average of a field $F$ as

$$\frac{1}{T1 - T0} \int_{T0}^{T1} F dt$$

Computes a `TimeIntegral`, and scales it.

#### TimeDerivative

Compute the time derivative of a Field $F$ through an explicit difference formula:

F'(t_n) approx frac{F(t_n)-F(t_{n-1})}{t_n-t_{n-1}}

#### TimeIntegral

Compute a time integral of a field $F$ by the backward trapezoidal method:

$$\int_{T0}^{T1} F dt \approx \sum_{n=1}^{n=N} \frac{F(t_{n-1}) + F(t_n)}{2} (t_{n-1} - t_n)$$

where $t_0 = T0$ and $t_N = T1$.

## 6.5 Parameter system

### 6.5.1 ParamDict

The base class extending the standard python dict.

### 6.5.2 Parameterized

Core functionality for parameterized subclassable components.

Merges base and user params into one ParamDict.

## 6.6 Other Classes

### 6.6.1 MeshPool

A mesh pool to reuse meshes across a program. FIXME: Mesh doesn't support weakref. id refers to the shared_ptr, not the actual object.

### 6.6.2 SpacePool

A function space pool to reuse spaces across a program.

## 6.7 Other Functions

### 6.7.1 get_grad_space

Get gradient space of Function.

This is experimental and currently only designed to work with CG-spaces.

### 6.7.2 get_parse_command_line_arguments

Return whether to parse command line arguments

### 6.7.3 set_parse_command_line_arguments

Switch on/off command line argument parsing

## 6.8 Utilities

### 6.8.1 Functions

#### boundarymesh_to_mesh_dofmap

Find the mapping from dofs on boundary FS to dofs on full mesh FS

#### cbc_log

Log on master process.

#### cbc_print

Print on master process.

#### cbc_warning

Raise warning on master process.

#### compute_connectivity

Compute connected regions of mesh. Regions are considered connected if they share a vertex through an edge.

#### create_function_from_metadata

Create a function from metadata

### create_slice

Create a slicemesh from a basemesh.

Arguments:

**basemesh** Mesh to slice

**point** Point in slicing plane

**normal** Normal to slicing plane

**closest_region** Set to True to extract disjoint region closest to specified point

**crinkle_clip** Set to True to return mesh of same topological dimension as basemesh

Only 3D-meshes currently supported for slicing.

Slice-instances are intended for visualization only, and may produce erronous results if used for computations.

### create_submesh

This function allows for a SubMesh-equivalent to be created in parallel

### get_memory_usage

Return memory usage in MB

### get_set_vector

Equivalent of setvector[set_indices] = getvector[get_indices] for global indices (MPI-blocking). Pass temp_array to avoid initiation of array on call.

### import_fenicstools

Import fenicstools helper function.

### in_serial

Return True if running in serial.

### mesh_to_boundarymesh_dofmap

Find the mapping from dofs on full mesh FS to dofs on boundarymesh FS

### on_master_process

Return True if on process number 0.

#### restriction_map

Return a map between dofs in Vb to dofs in V. Vb's mesh should be a submesh of V's Mesh.

#### safe_mkdir

Create directory without exceptions in parallel.

#### strip_code

Strips code of unnecessary spaces, comments etc.

#### time_to_string

Format time in seconds as a human readable string.

#### timeit

Simple timer

### 6.8.2 Classes

#### Loadable

Create an instance that reads a Field from file as specified by the parameters. Requires that the file is written in cbcpost (or in the same format).

Arguments:

**filename** Filename where function is stored

**fieldname** Name of Field

**timestep** Timestep to load

**time** Time

**saveformat** Saveformat of field

**s function** Function to load Field into

This class is used internally from :class:'.Replay' and :class:'Restart', and made to be passed to *PostProcessor.update_all*.

#### Slice

Deprecated Slice-class

### Timer

Class to perform timing.

Arguments:

**frequency** Frequency which to report timings.

CHAPTER 7

# Contributing

cbcpost is located at

https://bitbucket.org/simula_cbc/cbcpost

A *git* workflow is used, and the language is python. There are no strict guidelines followed, but as a rule of thumb, consider the following:

- No '*' imports
- No unused imports
- Whitespace instead of tabs
- All modules, public functions and classes should have reasonable docstrings
- 4 space indentation levels
- Unit tests should cover the majority of the code
- Look at existing code, and use common sense

cbcpost will follow the development of FEniCS, and will likely require development versions of FEniCS-components between releases.

## 7.1 Pull requests

If you wish to fix issues or add features, please check out the code using git, make your changes in a clean branch and create a pull request.

Before making a pull request, make sure that all unit tests pass, and that you add sufficient unit tests for new code.

To avoid unnecessary work, please contact the developers in advance.

## 7.2 Report problems

Please report any bugs or feature requests to the issue tracker on Bitbucket.

## 7.3 Contact developers

You can contact the developers directly:

- Øyvind Evju
- Martin Sandve Alnæs.