Oasis - User Manual

Mikael Mortensen
University of Oslo, Department of Mathematics, Mechanics Division

January 29, 2015

Oasis (Optimized And StrIpped Solver) is a high-performance Navier-Stokes solver written entirely in the Python interface to FEniCS. The solver is unstructured, runs with MPI and interfaces, through FEniCS, to state-of-the-art linear algebra backends like PETSc and Trilinos.

To be able to work with *Oasis*, a user should posess basic skills in how to solve PDEs through FEniCS, that in turn requires basic programming skills in Python. A good introduction to the FEniCS and *dolfin*, which is the problem solving environment of FEniCS, can be obtained by following the tutorial: `http://fenicsproject.org/documentation/tutorial`.

1 Installation

Oasis may be installed using the *git* version control system or by downloading released packages. *Oasis* depends on FEniCS (fenicsproject.org) and the development version of *Oasis* will at all times attempt to stay compatible with the development version of FEniCS. When a release is made for FEniCS, then a release will be made for *Oasis* as well. *Oasis* will not run with an incompatible version of FEniCS.

1.1 Dependensies (FEniCS)

FEniCS is installed by following instructions from the page `http://fenicsproject.org/download/`. There are binary packages for all FEniCS releases that can be installed easily for most operating systems (except Windows). If you want to use the development version of *Oasis*, then you need to install development version of FEniCS from source. This is quite a demanding task, unless it is performed through dorsal (see link on FEniCS download page).

1.2 Installation of ***Oasis*** through ***git***

Oasis can be installed simply by cloning the github repository to your own computer

```
$ git clone https://github.com/mikaem/Oasis.git
$ cd Oasis
```

No further installation is necessary and the clone will leave you on the *master* branch of *Oasis*. You may now choose to switch branch or to check out a previous release. A list of remote tags is obtained with

```
$ git ls-remote --tags
```

which at the time of writing results in two tags:

```
From https://github.com/mikaem/Oasis.git
7a77f9eae84a8bf6d0b601ed4a86f4c6eb57aa07        refs/tags/1.3.0
8738385dc82e602c03b7781557e7c5cc0426d0be        refs/tags/1.4.0
7512f61f0d799bbde2cbe43cbc95a2bfc79e3b68        refs/tags/1.5.0
575b8c826405d34494885ebf5085d5e5597217ab        refs/tags/v1.3
```

The *1.5.0* tag represents a release of *Oasis* that is compatible with version 1.5.0 of FEniCS. It is obtained though

```
$ git checkout 1.5.0
```

Note that you will not have write access to the remote repository at `https://github.com/mikaem/Oasis.git` and as such you cannot push the changes you make to this repository. If you intend to develop and commit code yourselves, you probably want to create a fork of the *Oasis* repository by pressing the **Fork** button on github (you need to be a registered user on github). Proceed by cloning this fork to your own computer and start working on a new branch. For example, to start a new branch using the tag of *Oasis* that is compatible with fenics 1.5.0, you can execute the following commands after forking (with user name on github 'github-username')

```
$ git clone https://github.com/github-username/Oasis.git
$ git checkout -b newbranch 1.5.0
$ ... make changes
$ git commit -a -m "Some changes"
$ git push --set-upstream origin newbranch
```

1.3 Installation of *Oasis* by downloading released version

Oasis may also be installed without git by downloading and extracting a released version from `https://github.com/mikaem/Oasis/releases`. To install a version of *Oasis* that is compatible with fenics version 1.5.0, you can execute the following commands

```
$ wget https://github.com/mikaem/Oasis/archive/1.5.0.tar.gz
$ tar -xvf 1.5.0.tar.gz
$ cd Oasis-1.5.0
```

This version of the solver will be detached from the main repository, though, and may not be kept up to date with the development version.

```
Oasis
├── __init__.py
├── NSfracStep.py
├── NSCoupled.py
├── common/
│   ├── __init__.py
│   ├── io.py
│   └── utilities.py
├── solvers/
│   ├── __init__.py
│   ├── NSfracStep/
│   │   ├── __init__.py
│   │   ├── Chorin.py
│   │   ├── IPCS.py
│   │   ├── IPCS_ABCN.py
│   │   ├── IPCS_ABE.py
│   │   ├── BDFPC.py
│   │   ├── BDFPC_Fast.py
│   │   └── LES/
│   │       ├── __init__.py
│   │       └── Wale.py
│   └── NSCoupled/
│       ├── __init__.py
│       ├── default.py
│       ├── naive.py
│       └── cylindrical.py
└── problems/
    ├── __init__.py
    ├── DrivenCavity.py
    ├── Nozzle2D.py
    ├── Cylinder.py
    ├── ...
    ├── NSfracStep/
    │   ├── __init__.py
    │   ├── DrivenCavity.py
    │   ├── Nozzle2D.py
    │   ├── Cylinder.py
    │   ├── Channel.py
    │   └── ...
    └── NSCoupled/
        ├── __init__.py
        ├── DrivenCavity.py
        ├── Nozzle2D.py
        ├── Cylinder.py
        └── ...
```

Figure 1: Directory tree structure of *Oasis*.

2 Files and folders

Oasis is designed as a Python package with a main executable module (`NSfracStep.py`) and three submodules (`common, solvers, problems`). *Oasis* consists of all files and folders shown in Figure 1.

2.1 NSfracStep.py

The main modules `NSfracStep.py` and `NSCoupled.py` are executable and requires only that the problem to be solved is added after the keyword problem, e.g.,

```
$ python NSfracStep.py problem=DrivenCavity solver=IPCS_ABCN
$ python NSCoupled.py problem=DrivenCavity element=TaylorHood
```

or the preferred way through Ipython

```
[1] run NSfracStep problem=DrivenCavity
[1] run NSCoupled problem=DrivenCavity element=TaylorHood
```

The solver keyword is optional since `IPCS_ABCN` is the default fractional step solver (IPCS is short for incremental pressure correction). The main solver module (`NSfracStep/NSCoupled`) pulls in required information from one solver module in the `solvers` subfolder and one problem in the `problems` subfolder. The required information from the problem module is a computational mesh, initial and boundary conditions, control parameters plus any other user defined action. A user should not need to do modify anything in `common` or `solvers` submodules and is only required to create a problem module.

The `NSfracStep` module contains a high-level implementation for a generic fractional step algorithm for the Navier-Stokes equations. The most significant part of the implementation is shown in Figure 2. All functions have default implementations in the `common`, `solvers` or `problems` submodules. Most interesting for users are functions ending in `hook`, that may be overloaded in the problem module. For example, the function `start_timestep_hook` is called at the start of every new time step and may as such be used to update, e.g., a time dependent boundary condition.

The `NSfracStep` module is responsible for creating function spaces, test and trial spaces as well as allocating functions to hold the actual finite element solution. The most important variables declared in the module are shown in Fig. 3.

2.2 problems

The `problems` submodule contains modules for implemented problems. There is one folder targeting the coupled solver and one for the fractional step solver. Code that is common for both solvers can be placed in the top level of the problems folder, as shown for `DrivenCavity, Nozzle2D` and `Cylinder` in Fig. 1.

The default version of all generic `hook` functions are implemented in module `problems/__init__.py` and the more solver specific hook functions are implemented in `problems/NSfracStep/__init__.py` and `problems/NSCoupled/__init__.py`. The most important are shown in Figure 4. Any function may be overloaded by reimplementing it in the user defined problem module.

```python
# Preassemble and prepare solver
vars().update(setup(**vars()))

# Anything problem specific
vars().update(pre_solve_hook(**vars()))

# Enter loop for time advancement
while t < T and not stop:
  t += dt
  inner_iter = 0
  # Do something at start of timestep
  start_timestep_hook(**vars())

  # Enter velocity/pressure inner loop
  for inner_iter < max_iters:
    inner_iter += 1
    if inner_iter == 1:
      assemble_first_inner_iter(**vars())

    # Solve tentative velocity
    for i, ui in enumerate(u_components):
      velocity_tentative_assemble(**vars())
      velocity_tentative_hook    (**vars())
      velocity_tentative_solve   (**vars())

    # Solve pressure correction
    pressure_assemble(**vars())
    pressure_hook    (**vars())
    pressure_solve   (**vars())

  # Solve velocity update
  velocity_update(**vars())

  # Solve for all scalar components
  if len(scalar_components) > 0:
    scalar_assemble(**vars())
    for ci in scalar_components:
      scalar_hook (**vars())
      scalar_solve(**vars())

  # Do something at end of timestep
  temporal_hook(**vars())

  # Save and update to next timestep
  stop = save_solution(**vars())

# Finalize solver
theend_hook(**vars())
```

Figure 2: The fractional step time integration algorithm in `NSfracStep.py`

```
# Declare function spaces and trial and test functions
V = FunctionSpace(mesh, "Lagrange", velocity_degree)
Q = FunctionSpace(mesh, "Lagrange", pressure_degree)
u, v = TrialFunction(V), TestFunction(V)
p, q = TrialFunction(Q), TestFunction(Q)

# Get dimension of problem
dim = mesh.geometry().dim()

# Create list of components we are solving for
u_components = map(lambda x: "u"+str(x), range(dim))# velocity components
uc_comp  =  u_components + scalar_components        # velocity + scalars
sys_comp = u_components + ["p"] + scalar_components # velocity +
    pressure + scalars

# Create dictionaries for the solutions at three timesteps
q_  = {ui: Function(V) for ui in uc_comp}
q_1 = {ui: Function(V) for ui in uc_comp}
q_2 = {ui: Function(V) for ui in u_components} # Note only velocity

# Allocate solution for pressure field and correction
p_ = q_["p"] = Function(Q)
phi_ = Function(Q)

# Create vector views of the segregated velocity components
u_  = as_vector([q_ [ui] for ui in u_components])# Velocity vector t
u_1 = as_vector([q_1[ui] for ui in u_components])# Velocity vector t-dt
u_2 = as_vector([q_2[ui] for ui in u_components])# Velocity vector t-2*dt
```

Figure 3: The declaration section of `NSfracStep.py`. Allocation of necessary storage and parameters for solving the momentum equation through its segregated components. `FunctionSpace`, `Function`, `TrialFunction`, `TestFunction`, `as_vector` are all classes or functions imported from the dolfin module.

Note the special calling sequence for the functions used in Fig. 2 and declared in Fig. 4. The functions are called with the entire NSfracStep namespace `**vars()` as argument and in the declaration of the function (Fig. 4) any variable required may be unpacked in the list of arguments and used by reference inside the function. There is no copying involved and the overhead in calling functions this way is very small, yet extremely flexible.

Control parameters The `problems` module contains a range of control parameters that may be overloaded by the user. The control parameters are kept in a dictionary called `NS_parameters`, which is declared in `problems/__init__.py` as shown in Fig. 5. Solver specific parameters are declared in `problems/NSfracStep/__init__.py` and `problems/NSCoupled/__init__.py`. The most significant for the fractinal step solver are shown in Fig. 7 and for the coupled solver in Fig. 6. The purpose of each parameter is also hinted at in Figs. 7, 6. Output of results to file is controlled by parameters `checkpoint`, `save_step`, `folder` and output_timeseries_as_vector. The folder variable is a string giving the name of the main folder used for storage. Using default value folder = "results", the results will be stored in folder Oasis/results/data/ in a directory tree similar to that shown in Fig. 8. To avoid accidentally overwriting old results, each new execution of the program creates a new folder with a unique integer under the data folder unless the solver is restarted from a previous solution. In that case, the same folder is used and new results are appended to those already existing. The Checkpoint folder contains all parameters (NS_parameters) and solution vectors stored in HDF5 format, that are continuously overwritten every checkpoint time step. The solution vectors in the Checkpoint folder contain the velocity at two previous time steps and the pressure from the latest, thus making it possible to stop and restart the solver from this solution at no loss of accuracy whatsoever. The TimeSeries folder contains solution files in dolfin's XDMF-HDF5 format, viewable by, e.g., the external visualisation tool Paraview (paraview.org). Results are stored every save_step time step and the velocity is stored as a vector and not three (or two) scalars. This exact behaviour may be altered by setting NS_parameters["output_timeseries_as_vector"] = False. The remaining folders in Fig. 8 are empty unless explicitly coded in the problem module, see, e.g., problems/Channel.py. The Stats folder contains turbulence statistics, whereas the Voluviz folder contains fields possible to view with Voluviz, an in-house visualisation tool developed at the Norwegian Defence Research Establishment (FFI), available upon request from the authors. Note that computing turbulence statistics and Voluviz-fields require the Python package fenicstools (https://github.com/mikaem/fenicstools).

2.3 solvers

The solvers submodule contains specific implementations of steady coupled and transient fractional step algorithms. The fractional step folder contains the following summodules:

- NSfracStep/__init__.py - Default (empty) implementation of most functions seen in Fig. 2 (serves as a base module).
- NSfracStep/Chorin.py - Naive implementation of Chorin's projection method. Adams-Bashforth advecting velocity and Crank-Nicolson for diffusion and advected velocity.
- NSfracStep/IPCS.py - A naive implementation of the incremental pressure correction scheme using Adams-Bashforth advecting velocity and Crank-Nicolson for diffusion and advected velocity. The scheme is second order in time.
- NSfracStep/IPCS_ABCN.py - Optimized implementation of IPCS.
- NSfracStep/IPCS_ABE.py - Optimized implementation using explicit Adams-Bashforth convection.
- NSfracStep/BDFPC.py - A naive implementation of a second order backwards differencing scheme.

- `NSfracStep/BDFPC_Fast.py` - Optimized version of `BDFPC`.

There are three different implementations of steady coupled solvers found under the NSCoupled folder.

- `NSCoupled/__init__.py` Default (empty) implementations of functions required by the coupled solver.
- `NSCoupled/naive.py` Naive coupled solver. No optimization.
- `NSCoupled/default.py` Fairly optimized solver preassembling forms that are not changing.
- `NSCoupled/Cylindrical.py` Coupled solver in cylindrical coordiantes.

The functions implemented in the `solvers` submodule cannot be overloaded by the user in the problem module.

2.4 common

The `common` folder contains two files:

- `__init__.py` contains one single function, `parse_command_line` that is used to parse any parameters supplied through the command line.
- `io.py` contains functions for storing and retrieving solutions.
  - `save_solution` - used for storing intermediate or checkpoint solutions.
  - `create_initial_folders` - used for creating the folders storing results (see Fig.8).
  - `check_if_kill` - checks if a file named `killoasis` has been placed in the `NS_parameters['folder']` directory. If found, then the solver is stopped cleanly at the end of the time step after saving the solution to the `Checkpoint` folder.
  - `init_from_restart` - used for restarting the solver from a previously stored solution in `Checkpoint` folder.
- `utilities.py` Contains special classes that overload Fenics' `Function` class by appending the classes with efficient methods for doing projection and fast assembly of linear and bilinear forms associated with the projection.

3 Implementing a new problem

Oasis is a programmable solver and the user is required to implement new problems by creating a new Python module (a file ending in .py) located in the *problems* folder. A problem module must implement at least

- A dolfin `Mesh`.

- Boundary conditions.
- Control parameters, e.g., viscosity, time step and end time.

And for most problems you will probably also want to set

- Initialisation of the solution. (dolfin `Functions` are automatically initialised to zero).
- Post-processing.

Lid driven cavity We will now illustrate by implementing the common lid driven cavity test problem. The lid driven cavity is computed on a two-dimensional domain $\Omega = [0, 1] \times [0, 1]$, where the velocity of the top lid at $y = 1$ is $\boldsymbol{u} = (1, 0)$ and there is no-slip on the remaining three surrounding walls. The kinematic viscosity is 0.01 and the flow is laminar. The driven cavity may be either steady or transient and as such the case may be solved by both coupled and fractional step solvers. For this reason we create common code in the top level problems folder `problems/DrivenCavity.py`

Start by creating a new empty file in the problems folder

```
bash>> cd Oasis/problems; touch DrivenCavity.py
```

Next on the todo list is to create a computational mesh. The mesh can either be created using dolfin's built in meshing capabilities or be read in from file. Furthermore, `mesh` may be created as a callable function or a variable name. When using builtins we usually choose to create a callable function, since this allows us to set the size of the mesh through the commandline interface. The mesh is here created as a callable function using the `UnitSquareMesh` function from the dolfin namespace. The mesh is skewed towards the walls since it is important to capture the large gradients of the flow near the corners. The coordinates of the mesh are collected in the variable `x`, which is a numpy array of shape $((Nx + 1) \cdot (Ny + 1), 2)$ with view into the `UnitSquareMesh` object. Thus, modifying `x` simultaneously modifies the coordinates of the mesh. We also create some strings marking the locations of the walls, that later will be used to create boundary conditions.

```
from dolfin import UnitSquareMesh
from numpy import cos, pi

# Create a mesh
def mesh(Nx=50, Ny=50, **params):
    m = UnitSquareMesh(Nx, Ny)
    x = m.coordinates()
    x[:] = (x - 0.5) * 2
    x[:] = 0.5*(cos(pi*(x-1.) / 2.) + 1.)
    return m

noslip = "std::abs(x[0]*x[1]*(1-x[0]))<1e-8"
top    = "std::abs(x[1]-1) < 1e-8"
bottom = "std::abs(x[1]) < 1e-8"
```

The entire namespace of `problems/__init__.py` can now be imported into either `problems/NSfracStep/DrivenCavit`
or `problems/NSCoupled/DrivenCavity.py` and code is thus reused efficiently.

We now have a mesh and are left with control parameters and boundary conditions. Control parameters may all be set through the commandline or by updating the `NS_parameters` dictionary (see Fig. 7 or 6). An implementation for the fractional step solver is

```python
from ..NSfracStep import *
from ..DrivenCavity import *

# Override some problem specific parameters
NS_parameters.update(
    nu = 0.001,
    T  = 1.0,
    dt = 0.001,
    plot_interval = 20,
    print_intermediate_info = 100,
    use_krylov_solvers = True)

# Specify boundary conditions
def create_bcs(V, **NS_namespace):
    bc0  = DirichletBC(V, 0, noslip)
    bc00 = DirichletBC(V, 1, top)
    bc01 = DirichletBC(V, 0, top)
    return dict(u0 = [bc00, bc0],
                u1 = [bc01, bc0],
                p  = [])
```

Note that `from ..DrivenCavity import *` imports the common namespace from `problems/DrivenCavity.py` and `from ..NSfracStep import *` imports the default parameters and subroutines from `problems/NSfracStep/__init__.py`. The `plot_interval` parameter represents a variable that does nothing in itself. However, we will make use of it in the `temporal_hook` function, called at the end of each time step.

Dirichlet boundary conditions are created like for any other FEniCS application by using dolfin's `DirichletBC` class:

```python
# Specify boundary conditions
noslip = "std::abs(x[0]*x[1]*(1-x[0]))<1e-8"
top = "std::abs(x[1]-1) < 1e-8"
def create_bcs(V, **NS_namespace):
    bc0 = DirichletBC(V, 0, noslip)
    bc00 = DirichletBC(V, 1, top)
    bc01 = DirichletBC(V, 0, top)
    return dict(u0 = [bc00, bc0],
                u1 = [bc01, bc0],
                p  = [])
```

The `noslip` and `top` strings are used to identify parts of the boundary using corresponding coordinates `(x, y) = (x[0], x[1])`. `V` is the `FunctionSpace` of the velocity components. Note that the boundary condition for the top lid is placed first in the list for `u0` (velocity component in x-direction). This has implications for the velocities in the two corners located at $(x, y) = (0, 1)$ and $(x, y) = (1, 1)$, that are now set to zero. If the boundary condition for the top (i.e., `bc00`) is placed last in the list, then we obtain $\boldsymbol{u} = (1, 0)$ in both corners. There are no boundary conditions for the pressure.

At this point we have already specified enough to make the solver run. Still, we want to make a few more modifications by initialising the solution and by writing some routines for visualising the solution as it evolves. The solution is by default initialised to zero. Here we simply apply the boundary conditions to the solutions, such that the solution at $t = 0$ has a top lid with velocity (1,0):

```
def initialize(x_1, x_2, bcs, **NS_namespace):
    for ui in x_2:
        [bc.apply(x_1[ui]) for bc in bcs[ui]]
        [bc.apply(x_2[ui]) for bc in bcs[ui]]
```

Note that `x_`, `x_1`, `x_2` are dictionaries with velocity components or pressure as keys and solution vectors as values. The vectors are views into the solution `Function`'s `q_`, `q_1`, `q_2`, declared in the `NSfracStep.py` module (see Fig. 3).

We want to visualise the velocity as a vector and we want the figure to be updated in the same frame as the solution progresses (i.e., do not create a new figure for each time step). To this end we need a `VectorFunctionSpace` and a velocity vector `Function uv` defined and returned to the `NSfracStep` namespace using the `pre_solve_hook` function, called prior to entering the time integration loop (see Fig. 2)

```
def pre_solve_hook(mesh, velocity_degree, **NS_namespace):
    Vv = VectorFunctionSpace(mesh, 'CG', velocity_degree)
    return dict(uv=Function(Vv))
```

The `Function uv` may now be unpacked, updated with the new solution and plotted in the `temporal_hook` function. Note also the use of the `plot_interval` parameter that we set in the `NS_parameters` dictionary.

```
def temporal_hook(tstep, u_, uv, p_, plot_interval, **NS_namespace):
    if tstep % plot_interval == 0:
        assign(uv.sub(0), u_[0])
        assign(uv.sub(1), u_[1])
        plot(uv, title='Velocity')
        plot(p_, title='Pressure')
```

Running the problem The problem runs with, e.g.,

```
[1] run NSfracStep problem=DrivenCavity Nx=100 Ny=100
```

Any parameter may be overloaded on the commandline. The expected outcome of the program is two plots, velocity vectors and pressure, that evolve during simulations. The velocity vectors are shown in Fig. 9 for time `T=1.0`. Furthermore, the solution will be stored to the `TimeSeries` folder under `drivencavity_results/data/1` each 10'th time step. Hence, for 1000 time steps, there should be 100 snapshots of both velocity and pressure. These can be post-processed using ParaView, e.g., by producing a movie from the time series.

Running with MPI The solver runs with MPI at no additional effort as long as FEniCS has been installed with support for MPI (the default packages of FEniCS are compiled with MPI support). Run the program in a bash-shell using 4 CPUs with command

```
bash>> mpirun -np 4 python NSfracStep.py problem=DrivenCavity T=10000.0
```

The mesh will then automatically be distributed across the 4 processors. The only visible difference is that dolfin's plots will look weird because each processor plots its own mesh and solution. Visualisation with MPI is thus best performed with a post-processing tool like Paraview, using the solution stored to `TimeSeries`. You may suppress intermediate plots by setting `plot_interval` to a really large number.

Stopping the solver Since the end time has been set to `T=10000.0` through the command-line, the solver will run for quite a long time. To stop it cleanly you can create an empty file called `killoasis` in the result folder:

```
bash>> touch drivencavity_results/killoasis
```

This will make sure that the solver stops only after storing the solution on the current time step to the `Checkpoint` folder. The solver may, as such, be restarted from this solution at a later convenience. See `problems/NSfracStep/Channel.py` for a turbulence simulation, where such behaviour has been implemented.

Adding scalars Any number of scalars may be added to the solver. To add a scalar, simply add the `scalar_components` list to the problem namespace, like

```
scalar_compnents = ["alfa", "beta"]
```

for adding two scalars named `alfa` and `beta`. The diffusivities are specified like, e.g.,

```
Schmidt["alfa"] = 1.
Schmidt["beta"] = 10.
```

where the Schmidt number is the momentum diffusivity divided by the scalar mass diffusivity.

Naturally, you also need to define the boundary and initial conditions. For illustration, we choose `alfa`=1 for $y = 1$ and `beta`=1 for $y = 0$. The `create_bcs` and `initialize` functions may to this end be extended like

```python
def create_bcs(V, **NS_namespace):
    bc0 = DirichletBC(V, 0, noslip)
    bc00 = DirichletBC(V, 1, top)
    bc01 = DirichletBC(V, 0, top)
    return dict(u0 = [bc00, bc0],
                u1 = [bc01, bc0],
                p  = [],
                alfa = [bc00],
                beta = [DirichletBC(V, 1, bottom)])

def initialize(x_1, x_2, bcs, **NS_namespace):
    for ui in x_1:
        [bc.apply(x_1[ui]) for bc in bcs[ui]]
    for ui in x_2:
        [bc.apply(x_2[ui]) for bc in bcs[ui]]
```

Note that the scalars use the same function space as the velocity components. However, the scalars are only stored on two time steps and as such the scalars are not found in the dictionary `x_2`, only `x_` and `x_1`.

```
problems/__init__.py:
  def body_force(mesh, **NS_namespace):
      """Specify body force"""
      return Constant((0,)*mesh.geometry().dim())

  def initialize(**NS_namespace):
      """Initialize solution."""
      pass

  def create_bcs(sys_comp, **NS_namespace):
      """Return dictionary of Dirichlet boundary conditions."""
      return dict((ui, []) for ui in sys_comp)

  def theend_hook(**NS_namespace):
      """Called at the very end."""
      pass

problems/NSfracStep/__init__.py:
  def velocity_tentative_hook(**NS_namespace):
      """Called just prior to solving for tentative velocity."""
      pass

  def pressure_hook(**NS_namespace):
      """Called prior to pressure solve."""
      pass

  def start_timestep_hook(**NS_namespace):
      """Called at start of new timestep"""
      pass

  def temporal_hook(**NS_namespace):
      """Called at end of a timestep."""
      pass

  def pre_solve_hook(**NS_namespace):
      """Called just prior to entering time-loop. Must return a
    dictionary."""
      return {}

problems/NSCoupled/__init__.py:
  def start_iter_hook(**NS_namespace):
      """Called at the start of a new iteration."""
      pass

  def NS_hook(**NS_namespace):
      """Called between assemble and solve."""
      pass

  def end_iter_hook(**NS_namespace):
      """Called at the end of an iteration."""
      pass
```

Figure 4: Most of the problem specific functions defined in `problems/__init__.py`, `problems/NSfracStep/__init__.py` and `problems/NSCoupled/__init__.py` that may be overloaded in the implemented problem module. The dictionary `NS_namespace` is the namespace of the main `NSfracStep/NSCoupled` module.

```
# Default parameters for all solvers
NS_parameters = dict(
  nu = 0.01,                # Kinematic viscosity
  folder = 'results',       # Relative folder for storing results
  velocity_degree = 2,      # default velocity degree
  pressure_degree = 1       # default pressure degree
  )
```

Figure 5: Common control parameters for both the coupled and the fractional step solvers.

```
# Default parameters NSCoupled solver
NS_parameters.update(
  # Solver parameters
  omega = 1.0,              # Underrelaxation factor

  # Some discretization options
  solver = "default",       # "default", "naive", "cylindrical"

  # Parameters used to tweek solver
  max_iter = 10,            # Maximum number of iterations
  max_error = 1e-8,         # Tolerance for absolute error
  print_velocity_pressure_convergence = False,

  # Parameters used to tweek output
  plot_interval = 10,
  output_timeseries_as_vector = True, # Store velocity as vector
)
```

Figure 6: The most common control parameters for the steady state coupled solver.

```
NS_parameters.update(
  # Physical constants and solver parameters
  t = 0.0,                # Time
  tstep = 0,              # Timestep
  T = 1.0,                # End time
  dt = 0.01,              # Time interval on each timestep

  # Some discretization options
  AB_projection_pressure = False,  # Use Adams Bashforth projection
  solver = "IPCS_ABCN",  # "IPCS_ABCN", "IPCS_ABE", "IPCS", "Chorin",
    "BDFPC", "BDFPC_Fast"

  # Parameters used to tweek solver
  max_iter = 1,           # Number of inner pressure velocity iterations
  max_error = 1e-6,       # Tolerance for inner iterations
  iters_on_first_timestep = 2, # Number of iterations on first timestep
  use_krylov_solvers = False,  # Otherwise use LU-solver
  print_intermediate_info = 10,
  print_velocity_pressure_convergence = False,

  # Parameters used to tweek output
  plot_interval = 10,
  checkpoint = 10,        # Overwrite solution in Checkpoint folder every.
  save_step = 10,         # Store solution each save_step
  restart_folder = None, # For restarting solution
  output_timeseries_as_vector = True, # Store velocity as vector

  # Choose LES model and set default parameters
  les_model = None,        # None, Wale

  # Solver parameters to go in parameters['krylov_solver']
  krylov_solvers = dict(
    monitor_convergence = False, report = False,
    error_on_nonconvergence = False,
    nonzero_initial_guess = True,
    maximum_iterations = 200,
    relative_tolerance = 1e-8,
    absolute_tolerance = 1e-8),

  # Velocity update
  velocity_update_solver = dict(
    method = 'default', #"lumping", "gradient_matrix"
    solver_type = 'cg',
    preconditioner_type = 'jacobi',
    low_memory_version = False),

  velocity_krylov_solver = dict(
    solver_type = 'bicgstab',
    preconditioner_type = 'jacobi'),

  pressure_krylov_solver = dict(
    solver_type = 'gmres',
    preconditioner_type = 'hypre_amg'),

)
# For periodic domains recreate constrained_domain in problem module
constrained_domain = None

# To solve for scalars provide a list like ['scalar1', 'scalar2']
scalar_components = []

# Diffusivities: Schmidt = nu/D (momentum diffusivity / mass diffusivity)
Schmidt = defaultdict(lambda: 1.)
```

Figure 7: The most common control parameters for the fractional step solver.

```
results
└── data
    └── 1
        ├── Checkpoint
        │   ├── p.h5
        │   ├── params.dat
        │   ├── u0.h5
        │   ├── u1.h5
        │   └── u2.h5
        └── TimeSeries
            ├── params.dat
            ├── p_from_tstep_0.h5
            ├── p_from_tstep_0.xdmf
            ├── u_from_tstep_0.h5
            └── u_from_tstep_0.xdmf
```

Figure 8: Directory tree structure of results stored by *Oasis*.

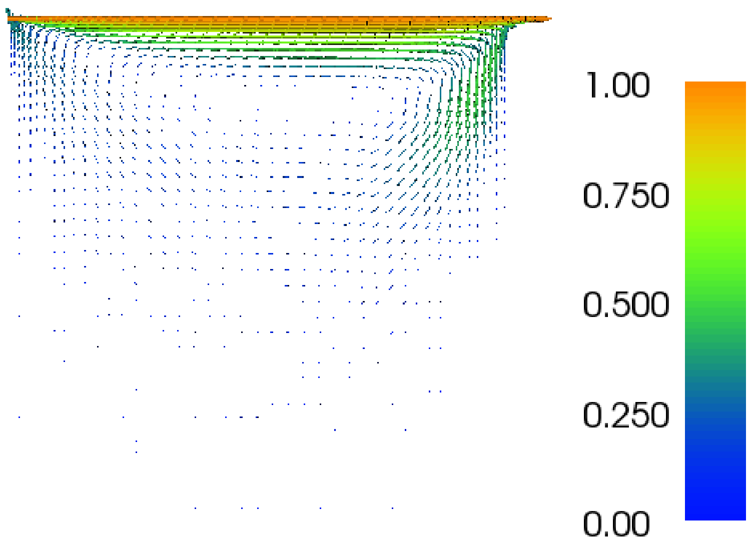

Figure 9: Driven cavity flow. Velocity vectors at T=1.0.