

Oasis: a high-level/high-performance open source Navier-Stokes solver

Mikael Mortensen^{a,b,*}, Kristian Valen-Sendstad^{b,c}

^aUniversity of Oslo, Moltke Moes vei 35, 0851 Oslo, Norway

^bCenter for Biomedical Computing at Simula Research Laboratory, P.O.Box 134, N-1325 Lysaker, Norway

^cUniversity of Toronto, 5 Kings College Road, Toronto, ON, Canada

Abstract

Oasis is a high-level/high-performance finite element Navier-Stokes solver written from scratch in Python using building blocks from the FEniCS project (fenicsproject.org). The solver is unstructured and targets large-scale applications in complex geometries on massively parallel clusters. *Oasis* utilizes MPI and interfaces, through FEniCS, to the linear algebra backend PETSc. *Oasis* advocates a high-level, programmable user interface through the creation of highly flexible Python modules for new problems. Through the high-level Python interface the user is placed in complete control of every aspect of the solver. A version of the solver, that is using piecewise linear elements for both velocity and pressure, is shown reproduce very well the classical, spectral, turbulent channel simulations of Moser, Kim and Mansour at $Re_\tau = 180$ [Phys. Fluids, vol 11(4), p. 964]. The computational speed is strongly dominated by the iterative solvers provided by the linear algebra backend, which is arguably the best performance any similar implicit solver using PETSc may hope for. Higher order accuracy is also demonstrated and new solvers may be easily added within the same framework.

Keywords: CFD; FEniCS; Python; Navier-Stokes

PROGRAM SUMMARY

Manuscript Title: Oasis: a high-level/high-performance open source Navier Stokes solver

Authors: Mikael Mortensen, Kristian Valen-Sendstad

Program Title: Oasis

Journal Reference:

Catalogue identifier:

Licensing provisions: GNU Lesser GPL version 3 or any later version

Programming language: Python/C++

Computer: Any single laptop computer or cluster.

Operating system: Any(Linux, OSX, Windows)

RAM: a few Megabytes to several hundred Gigabytes.

Number of processors used: 1 - 1000

Keywords: FEniCS, Python, MPI, C++, finite element, fractional step

Classification: 12

External routines/libraries: FEniCS

(www.fenicsproject.org, that in turn depends on a number of external libraries like MPI, PETSc, Epetra, Boost and ParMetis)

Nature of problem:

Incompressible, Newtonian fluid flow.

Solution method:

The finite element method.

Unusual features:

FEniCS automatically generates and compiles low-level C++ code based on high-level Python code.

1. Introduction

The Navier-Stokes equations describe the flow of incompressible, Newtonian fluids. The equations are transient, nonlinear and velocity is non-trivially coupled with pressure. A lot of research has been devoted to finding efficient ways of linearizing, coupling and solving these equations. Many commercial solvers for Computational Fluid Dynamics (CFD) are available, and, due to the complexity of the high-level implementations (usually Fortran or C), users are often operating these solvers through a Graphical User Interface (GUI). To implement a generic, unstructured Navier-Stokes solver from scratch in a low-level language like C or Fortran is a considerable and time-consuming task involving

*Corresponding author.

E-mail address: mikaem@math.uio.no

tens of thousands of lines of error prone code that require much maintenance. Nowadays, as will be shown in this paper, the use of new and modern high-level software tools enables developers to cut the size of programs down to a few hundred lines and development times to hours.

The implementation of any unstructured (Eulerian) CFD-solver requires a computational mesh. For most CFD software packages today the mesh is generated by a third-party software like, e.g., the open source projects VMTK [1], Gmsh [2] or Cubit [3]. To solve the governing equations on this computational mesh, the equations must be linearized and discretized such that a solution can be found for a certain (large) set of degrees of freedom. Large systems of linear equations need to be assembled and subsequently solved by appropriate direct or iterative methods. Like for mesh generation, basic linear algebra, with matrix/vector storage and operations, is nowadays most commonly outsourced to third-party software packages like PETSc [4] and Trilinos [5] (see, e.g., [6, 7, 8]).

With both mesh generation and linear algebra outsourced, the main job of CFD solvers boils down to linearization, discretization and assembly of the linear system of equations. This is by no means a trivial task as it requires, e.g., maps from computational cells to global degrees of freedom and connectivity of cells, facets and vertices. For parallel performance it is also necessary to distribute the mesh between processors and set up for inter-communication between compute nodes. Fortunately, much of the Message Passing Interface (MPI) is already handled by the providers of basic linear algebra. When it comes down to the actual discretization, the most common approaches are probably the finite volume method, which is very popular for fluid flow, finite differences or the finite element method.

FEniCS [9] is a generic open source software framework that aims at automating the discretization of differential equations through the finite element method. FEniCS takes full advantage of specialized, reliable and robust third-party providers of computational software and interfaces to both PETSc and Trilinos for linear algebra and several third-party mesh generators. FEniCS utilizes the Unified Form Language (UFL, [10]) and the FEniCS Form Compiler (FFC, [11]) to automatically generate low-level C++ code that efficiently evaluates any equation formulated as a finite element variational form. The FEniCS user has to provide

the high-level variational form that is to be solved, but does not need to actually perform any coding on the level of the computational cell, or element. A choice is made of finite element basis functions, and code is then generated for the form accordingly. There is a large library of possible finite elements to choose from and they may be combined both implicitly in a coupled manner or explicitly in a segregated manner - all at the same level of complexity to the user. The user never has to see the generated low-level code, but, this being an open source project, the code is wide open for inspection and even manual fine-tuning and optimization is possible.

In this paper we will describe the Navier-Stokes solver *Oasis*, that is written from scratch in Python, using building blocks from FEniCS and the PETSc backend. Our goal with this paper is to describe a code that is (i) short and easily understood, (ii) easily configured and (iii) as fast and accurate as state-of-the-art Navier-Stokes solvers developed entirely in low-level languages.

We assume that the reader has some basic knowledge of how to write simple solvers for partial differential equations using the FEniCS framework. Otherwise, reference is given to the online FEniCS tutorial [12].

2. Fractional step algorithm

In *Oasis* we are solving the incompressible Navier-Stokes equations, optionally complemented with any number of passive or reactive scalars. The governing equations are thus

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} = \nu \nabla^2 \mathbf{u} - \nabla p + \mathbf{f}, \quad (1)$$

$$\nabla \cdot \mathbf{u} = 0, \quad (2)$$

$$\frac{\partial c_\alpha}{\partial t} + \mathbf{u} \cdot \nabla c_\alpha = D_\alpha \nabla^2 c_\alpha + f_\alpha, \quad (3)$$

where $\mathbf{u}(\mathbf{x}, t)$ is the velocity vector, ν the kinematic viscosity, $p(\mathbf{x}, t)$ the fluid pressure, $c_\alpha(\mathbf{x}, t)$ is the concentration of species α and D_α its diffusivity. Any volumetric forces (like buoyancy) are denoted by $\mathbf{f}(\mathbf{x}, t)$ and chemical reaction rates (or other scalar sources) by $f_\alpha(\mathbf{c})$, where $\mathbf{c}(\mathbf{x}, t)$ is the vector of all species concentrations. The constant fluid density is incorporated into the pressure. Note that through the volumetric forces there is a possible feedback to the Navier-Stokes equations from the species, and, as such, a Boussinesq formulation

for natural convection (see, e.g., [13]) is possible within the current framework.

We will now outline a generic fractional step method, where the velocity and pressure are solved for in a segregated manner. Since it is important for the efficiency of the constructed solver, the velocity vector \mathbf{u} will be split up into its individual components u_k .¹ Time is split up into uniform intervals² using a constant time step $\Delta t = t^n - t^{n-1}$, where superscript n is an integer and $t^n \in \mathcal{R}^+$. The governing equations are discretized in both space and time. Discretization in space is performed using finite elements, whereas discretization in time is performed with finite differences. Following Simo and Armero [14] the generic fractional step algorithm can be written as

$$\frac{u_k^I - u_k^{n-1}}{\Delta t} + B_k^{n-1/2} = \nu \nabla^2 \tilde{u}_k - \nabla_k p^* + f_k^{n-1/2} \quad \text{for } k = 1, \dots, d, \quad (4)$$

$$\nabla^2 \varphi = -\frac{1}{\Delta t} \nabla \cdot \mathbf{u}^I, \quad (5)$$

$$\frac{u_k^n - u_k^I}{\Delta t} = -\nabla_k \varphi \quad \text{for } k = 1, \dots, d, \quad (6)$$

$$\frac{c_\alpha^n - c_\alpha^{n-1}}{\Delta t} + B_\alpha^{n-1/2} = D_\alpha \nabla^2 \tilde{c}_\alpha + f_\alpha^{n-1/2}, \quad (7)$$

where u_k^n is component k of the velocity vector at time t^n , d is the dimension of the problem, $\varphi = p^{n-1/2} - p^*$ is a pressure correction and p^* is a tentative pressure. We are solving for the velocity and pressure on the next time step, i.e., u_k^n for $k = 1, \dots, d$ and $p^{n-1/2}$. However, the tentative velocity equation (4) is solved with the tentative velocity component u_k^I as unknown. To avoid strict time step restrictions, the viscous term is discretized using a semi-implicit Crank-Nicolson interpolated velocity component $\tilde{u}_k = 0.5(u_k^I + u_k^{n-1})$. The nonlinear convection term is denoted by $B_k^{n-1/2}$, indicating that it should be evaluated at the midpoint between time steps n and $n-1$. Two different discretizations of convection are currently used by *Oasis*

$$B_k^{n-1/2} = \frac{3}{2} \mathbf{u}^{n-1} \cdot \nabla u_k^{n-1} - \frac{1}{2} \mathbf{u}^{n-2} \cdot \nabla u_k^{n-2}, \quad (8)$$

$$B_k^{n-1/2} = \bar{\mathbf{u}} \cdot \nabla \tilde{u}_k, \quad (9)$$

¹FEniCS can alternatively solve vector equations where all components are coupled.

²It is trivial to use nonuniform intervals, but uniform is used here for convenience.

where the first is a fully explicit Adams-Bashforth discretization and the second is implicit, with an Adams-Bashforth projected convecting velocity vector $\bar{\mathbf{u}} = 1.5 \mathbf{u}^{n-1} - 0.5 \mathbf{u}^{n-2}$ and Crank-Nicolson for the convected velocity. Both discretizations are second order accurate in time, and, since the convecting velocity is known, there is no implicit coupling between the (possibly) three velocity components solved for.

Convection of the scalar is denoted by $B_\alpha^{n-1/2}$. The term must be at most linear in c_α^n and otherwise any known velocity and scalar may be used in the discretization. Note that solving for c_α^n the velocity \mathbf{u}^n will be known and may be used to discretize $B_\alpha^{n-1/2}$. The discretization used in *Oasis* is

$$B_\alpha^{n-1/2} = \bar{\mathbf{u}} \cdot \nabla \tilde{c}_\alpha$$

where $\tilde{c}_\alpha = 0.5(c_\alpha^n + c_\alpha^{n-1})$.

An iterative fractional step method involves solving Eq. (4) for all tentative velocity components and (5) for a pressure correction. The procedure is repeated a desired number of times before finally a velocity correction (6) is solved to ensure conservation of mass before moving on to the next time step. The fractional step method can thus be outlined as shown in Algorithm 1. Note that if the momentum equation depends on the scalar (e.g., when using a Boussinesq model), then there may also be a second iterative loop over Navier-Stokes and temperature. The iterative scheme shown in Algorithm 1 is based on the observation that the tentative velocity computed in Eq. (4) only depends on previous known solutions $\mathbf{u}^{n-1}, \mathbf{u}^{n-2}$ and not \mathbf{u}^n . As such, the velocity update can be placed outside the inner iteration. In case of an iterative scheme where the convection depends on \mathbf{u}^n (e.g., $\mathbf{u}^n \cdot \nabla \tilde{u}_k$) the update would have to be moved inside the inner loop.

We now have an algorithm that can be used to integrate the solution forward in time, and it is clear that the fractional step algorithm allows us to solve for the coupled velocity and pressure fields in a computationally efficient segregated manner. The efficiency and long term stability (see [14]) are the main motivations for our choice of algorithm. However, we should mention here that there are plenty of similar, alternative algorithms for time stepping of segregated solvers. The most common algorithm is perhaps Pressure Implicit with Splitting of Operators (PISO) [15], which is used by both Ansys-Fluent [16], Star-CD [17] and OpenFOAM [18]. A completely different strategy would

```

Set time and initial conditions
t = 0
for time steps n = 0, 1, 2, ... do
  t = t + dt
  for inner iterations i = 0, 1, ... do
     $\varphi = p^* = p^{n-1/2}$ 
    solve (4) for  $u_k^I, k = 1, \dots, d$ 
    solve (5) for  $p^{n-1/2}$ 
     $\varphi = p^{n-1/2} - \varphi$ 
  end
  solve (6) for  $u_k^n, k = 1, \dots, d$ 
  solve (7) for  $c_\alpha^n$ 
  update to next timestep
end

```

Algorithm 1: Generic fractional step algorithm for the Navier-Stokes equations.

be to solve for velocity and pressure simultaneously (coupled solvers). Using FEniCS such a coupled approach is straightforward to implement, and, in fact, it requires less coding than the segregated one. However, since the coupled approach requires more memory than a segregated, and since there are more issues with the efficiency of linear algebra solvers, the segregated approach is favoured here.

We are still left with the spatial discretization and the actual implementation. To this end we will first show how the implementation can be performed naively, using very few lines of Python code. We will then, finally, describe the implementation of the high-performance solver.

3. Variational formulations for the fractional step solver

The governing PDEs (4), (5), (6) and (7) are discretized with the finite element method in space on a bounded domain $\Omega \subset R^d$, with $2 \leq d \leq 3$, and the boundary $\partial\Omega$. Trial and test spaces for the velocity components are defined as

$$\begin{aligned} V &= \{v \in H^1(\Omega) : v = u_0 \text{ on } \partial\Omega\}, \\ \hat{V} &= \{v \in H^1(\Omega) : v = 0 \text{ on } \partial\Omega\}, \end{aligned} \quad (10)$$

where u_0 is a prescribed velocity component on part $\partial\Omega$ of the boundary and $H^1(\Omega)$ is the Sobolev space containing functions v such that v^2 and $|\nabla v|^2$ have finite integrals over Ω . Both the scalars and pressure use the same $H^1(\Omega)$ space without the restricted boundary part. The test functions for velocity component and pressure are denoted as v and

q , respectively, whereas the scalar simply uses the same test function as the velocity component.

To obtain a variational form for component k of the tentative velocity vector, we multiply equation (4) by v and then integrate over the entire domain using integration by parts on the Laplacian

$$\begin{aligned} \int_{\Omega} \left(\frac{u_k^I - u_k^{n-1}}{\Delta t} + B_k^{n-1/2} \right) v + \nu \nabla \tilde{u}_k \cdot \nabla v \, dx = \\ \int_{\Omega} \left(-\nabla_k p^* + f_k^{n-1/2} \right) v \, dx + \int_{\partial\Omega} \nu \nabla_n \tilde{u}_k v \, ds. \end{aligned} \quad (11)$$

Here ∇_n represents the gradient in the direction of the outward normal on the boundary. Note that the trial function u_k^I enters also through the Crank-Nicolson velocity component $\tilde{u}_k = 0.5(u_k^I + u_k^{n-1})$. The boundary term is only important for some boundaries and is neglected for the rest of this paper.

The variational form for the pressure correction is obtained by multiplying Eq. (5) by q and then integrating over the domain, using again integration by parts

$$\int_{\Omega} \nabla \varphi \cdot \nabla q \, dx - \int_{\partial\Omega} \nabla_n \varphi q \, ds = \int_{\Omega} \frac{\nabla \cdot \mathbf{u}^I}{\Delta t} q \, dx. \quad (12)$$

The boundary integral can be neglected for all parts of the domain where the velocity is prescribed.

A variational form for the velocity update of component k is obtained by multiplying (6) by v and integrating over the domain

$$\int_{\Omega} \frac{u_k^n - u_k^I}{\Delta t} v \, dx = - \int_{\Omega} \nabla_k \varphi v \, dx. \quad (13)$$

Finally, a variational form for the scalar component α is obtained by multiplying Eq. (7) by v , and then integrating over the domain using integration by parts on the diffusion term

$$\begin{aligned} \int_{\Omega} \left(\frac{c_\alpha^n - c_\alpha^{n-1}}{\Delta t} + B_\alpha^{n-1/2} \right) v + D_\alpha \nabla \tilde{c}_\alpha \cdot \nabla v \, dx = \\ \int_{\Omega} f_\alpha^{n-1/2} v \, dx + \int_{\partial\Omega} D_\alpha \nabla_n \tilde{c}_\alpha v \, ds. \end{aligned} \quad (14)$$

4. Oasis

We now have all the variational forms that together constitute a fractional step solver for the Navier-Stokes equations, complemented with any

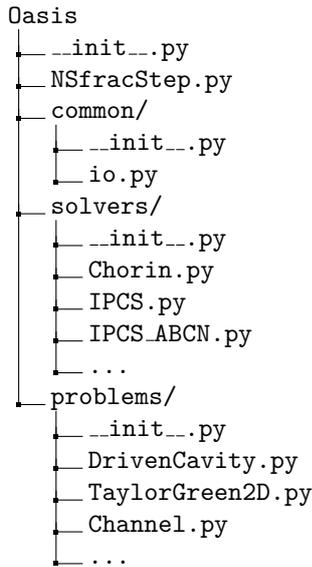


Figure 1: Directory tree structure of Python package *Oasis*.

number of scalar fields. We will now describe how the fractional step algorithm has been implemented in *Oasis* and discuss the design of the solver package. For installation of the software, see the user manual [19]. Note that this paper refers to version 1.3 of the *Oasis* solver, which in turn is consistent with version 1.3 of FEniCS.

4.1. Python package

The *Oasis* solver is designed as a Python package with tree structure shown in Fig. 1. The generic fractional step algorithm is implemented in the top level Python module `NSfracStep.py` and the solver is run by executing this module within a Python shell using appropriate keyword arguments, e.g.,

```
>>> python NSfracStep.py problem=Channel
      solver=IPCS
```

The fractional step solver pulls in a required mesh, parameters and functions from two submodules located in folders `solvers` and `problems`. The user communicates with the solver through the implementation of new problem modules in the `problems` folder. With the design choice of placing the solver at the root level of a Python module, there is a conscious decision of avoiding object oriented classes. However, remembering that everything in Python is an object, we still, as will be shown, make heavy use of overloading Python objects (functions, variables).

The fractional step module `NSfracStep.py` is merely one hundred lines of code (excluding comments and spaces) dedicated to allocation of necessary storage and variables, plus the implementation of the generic fractional step Algorithm 1. The first half of `NSfracStep.py` is shown in Fig. 2. Except from the fact that most details are kept in submodules, the design is very similar to most FEniCS Python demos, and, as such, *Oasis* should feel familiar and be quite easily accessible to new users with some FEniCS experience.

Consider the three functions towards the end of Fig. 2 that take `**vars()` as argument. The `body_force` function returns \mathbf{f} in (1) and should thus by default return a `Constant` vector of zero values (length 2 or 3 depending on whether the problem is 2D or 3D). The `initialize` function initializes the solution in `q_`, `q_1`, `q_2` and `create_bcs` must return a dictionary of boundary conditions. These functions are clearly problem specific and thus default implementations are found in the `problems/__init__.py` module that all new problems are required to import from. The default functions may then be overloaded as required by the user in the new problem module (see, e.g., Fig. 5). An interesting feature is the argument `**vars()`, which is used for all three functions. The Python built-in function `vars()` returns a dictionary of the current module's namespace, i.e., it is here `NSfracStep`'s namespace containing `v`, `q`, `u`, `v`, and all the other variables seen in Fig. 2. When `**vars()` is used in a function's signature, any variable declared within `NSfracStep`'s namespace may be unpacked in that function's list of arguments and accessed by reference. Figure 3 illustrates this nicely through the default implementations (found in `problems/__init__.py`) of the three previously mentioned functions.

After initialization the solution needs to be advanced in time. The entire implementation of the time integration performed in `NSfracStep.py` is shown in Fig. 4, that closely resembles Algorithm 1. In Fig. 4 the functions ending in `hook` are imported through the `problems` submodule, `save_solution` from `common` and the rest of the functions are imported from the `solvers` submodule.

The `common` submodule basically contains routines for parsing the command line and for storing and retrieving the solution (`common/io.py`). There is, for example, a routine here that can be used if the solver needs to be restarted from a previous simulation. The `problems` and `solvers` submodules are more elaborate and will be described next.

```

from common import *

commandline_kwargs = parse_command_line()

# Get the problem from commandline
problem = commandline_kwargs.get("problem", "DrivenCavity")

# import mesh, NS_parameters, body_force, create_bcs, velocity_degree, etc...
exec("from problems.{} import {}".format(problem))

# Update NS_parameters with parameters modified through the command line
NS_parameters.update(commandline_kwargs)
vars().update(NS_parameters)

# Import functionality from chosen solver
exec("from solvers.{} import {}".format(solver))

# 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 u_components}
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 at t
u_1 = as_vector([q_1[ui] for ui in u_components]) # Velocity vector at t - dt
u_2 = as_vector([q_2[ui] for ui in u_components]) # Velocity vector at t - 2*dt

# Set kinematic viscosity constant
nu = Constant(NS_parameters["nu"])

# Set body force
f = body_force(**vars())

# Initialize solution
initialize(**vars())

# Get boundary conditions
bcs = create_bcs(**vars())

```

Figure 2: The opening section of NSfracStep.py. Allocation of necessary storage and parameters for solving the momentum equation through its segregated components. Note that a mesh, some parameters (for e.g., viscosity, time step, end time etc), and some functions (for e.g., body force, boundary conditions or initializing the solution) must be imported from the problem module. The UFL function `as_vector` creates vectors (`u_`, `u_1`, `u_2`) from the segregated velocity components. The built-in function `vars()` returns the current modules namespace. Neglecting scalar components the list `sys_comp = ["u0", "u1", "p"]` for 2D and `["u0", "u1", "u2", "p"]` for 3D problems. The list is used as keys for the dictionary `bcs`.

```

def body_force(mesh, **NS_namespace):
    """Specify body force"""
    dim = mesh.geometry().dim()
    return Constant((0,)*dim)

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

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

```

Figure 3: Default implementations of three of the functions found in `problems/__init__.py`.

The problems submodule

Oasis is a programmable solver and the user is required to implement the problem that is to be solved. The implemented problem module's namespace must include at least a computational mesh and functions for specifying boundary conditions and initialization of the solution. Other than that, the user may interact with `NSfracStep` through certain `hook` files strategically placed within the time advancement loop, as seen in Fig. 4, and as such there is no need to modify `NSfracStep` itself.

Consider a lid driven cavity with $\Omega = [0, 1] \times [0, 1]$. The velocity boundary conditions are $\mathbf{u} = (1, 0)$ for the top lid ($y = 1$) and zero for the remaining walls. We start the simulations from a fluid at rest and advance the solution in time steps of $\Delta t = 0.001$ from $t = 0$ to $t = 1$. The viscosity is set to $\nu = 0.001$. This problem can be implemented as shown in Fig. 5. Here we have made use of the standard python package `numpy` and two `dolfin` classes `UnitSquareMesh` and `DirichletBC`. `UnitSquareMesh` creates a computational mesh on the unit square, whereas `DirichletBC` creates Dirichlet boundary conditions for certain segments of the boundary identified through two strings `noslip` and `top` (`x[0]` and `x[1]` represent coordinates x and y respectively). A default set of problem parameters can be found in the dictionary `NS_parameters` declared in `problems/__init__.py`, and all these parameters may be overloaded, either as shown in Fig.5, or through the command line.

A comprehensive list of parameters and their use is given in the user manual. We use preconditioned iterative Krylov solvers (`NS_parameters["use_krylov_solvers"]=True`), and not the default direct solvers based on LU decomposi-

```

# Preassemble and prepare solver
vars().update(setup(**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 Eq. (17)
        for i, ui in enumerate(u_components):
            velocity_tentative_assemble(**vars())
            velocity_tentative_hook (**vars())
            velocity_tentative_solve (**vars())

        # Solve Eq. (18)
        pressure_assemble(**vars())
        pressure_hook (**vars())
        pressure_solve (**vars())

    # Solve Eq. (19)
    velocity_update(**vars())

    # Solve for all scalar components (20)
    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 4: Time loop in `NSfracStep.py`

tion, since the former here are faster and require less memory (the exact choice of iterative solvers is discussed further in Sec7). Note that FEniCS interfaces to a wide range of different linear algebra solvers and preconditioners. The iterative solvers used by *Oasis* are defined in function `get_solvers` imported from the `solvers` submodule.

To run the solver for the driven cavity problem we need to specify this through the command line - along with any other parameter we wish to modify at runtime. For example, the default size of the computational mesh has been implemented in Fig. 5

```

from problems import *
from numpy import cos, pi

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

# Override some problem specific parameters
NS_parameters.update(
    nu = 0.001,
    T = 1.0,
    dt = 0.001,
    Nx = 50,
    Ny = 50,
    use_krylov_solvers = True)

# 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 = [])

# Initialize by enforcing boundary cond.
def initialize(q_1, q_2, bcs, **NS_namespace):
    for ui in q_2:
        for bc in bcs[ui]:
            bc.apply(q_1[ui].vector())
            bc.apply(q_2[ui].vector())

```

Figure 5: DrivenCavity.py - Implementation of the driven cavity problem.

as $N_x=N_y=50$. This may be overloaded through the command line while running the solver, like

```

>>> python NSfracStep.py
       problem=DrivenCavity Nx=20 Ny=20

```

The ability to overload parameters through the command line is useful for, e.g., fast convergence testing.

The computational mesh has to be part of the problem module's namespace. However, it does not need to be defined as a callable function, like that used in Fig. 5. Three equally valid examples are

```

mesh = UnitSquareMesh(10, 10)
mesh = Mesh("SomeMesh.xml.gz")
def mesh(N, **params):
    return UnitSquareMesh(N, N)

```

The first mesh is hardcoded in the module and can-

not be modified through the commandline. The second approach, `mesh = Mesh("some_mesh.xml.gz")`, is usually used whenever the mesh has been created by an external software. The third option uses a callable function, making it possible to modify the mesh size through the command line.

A complete list of all default functions and parameters that may be overloaded by the user in their implemented problem module is found in `problems/__init__.py`.

The solvers submodule

The finer details of the fractional step solver are implemented in the `solvers` submodule. A list of all functions that are imported by `NSfracStep` is found in the `solvers/__init__.py` module. The most important can be seen in Fig. 4. Note the special calling routine for the function `setup`

```

vars().update(setup(**vars()))

```

The purpose of this `setup` function is to prepare the solver for time advancement. This could mean either defining UFL forms of the variational problems (see Fig. 6) or to preassemble matrices that do not change in time, e.g., diffusion and mass matrices (see Sec. 5). The `setup` function returns a dictionary and this dictionary is updated and made part of the `NSfracStep` namespace through the use of `vars().update`.

We may now take the naive approach and implement all variational forms exactly as described in Sec. 3. A smart approach, on the other hand, will take advantage of certain special features of the Navier-Stokes equations. The starting point for implementing a new solver, though, will usually be the naive approach. A naive implementation requires very few lines of code, it is easy to debug and as such it can be very useful for verification of the slightly more complex and optimized solvers to be discussed in the next section.

The `solvers/IPCS.py` module contains a naive implementation of the variational forms (11), (12) and (13). The forms are implemented using the `setup` function shown in Fig. 6. Dictionaries are used to hold the forms for the velocity components, whereas there is only one form required for the pressure. Note the very close correspondence between the high-level Python code and the mathematical description of the variational forms. The variational forms are assembled and solved through the very compact routines `velocity_tentative_solve`, `pressure_solve` and `velocity_update` that are imple-

```

def setup(u_, u_1, q_, q_1, u, v, p, q,
         nu, dt, p_, f, u_components,
         phi_, **NS_namespace):
    F, Fu = {}, {}
    U_AB = 1.5*u_1 - 0.5*u_2
    for i, ui in enumerate(u_components):
        # Crank-Nicolson velocity
        U_CN = 0.5*(u+q_1[ui])

        # Tentative velocity variational form
        F[ui] = (1./dt*inner(u-q_1[ui], v)*dx
                + inner(dot(U_AB, grad(U_CN)), v)*dx
                + nu*inner(grad(U_CN), grad(v))*dx
                + inner(p_.dx(i), v)*dx
                - inner(f[i], v)*dx)

        # Velocity update variational form
        Fu[ui] = (inner(u, v)*dx
                 - inner(q_[ui], v)*dx
                 + dt*inner(phi_.dx(i), v)*dx)

        # Variational form for pressure
        phi = p - p_
        Fp = (inner(grad(q), grad(phi))*dx
              - (1./dt)*div(u_)*q*dx)

    return dict(F=F, Fu=Fu, Fp=Fp)

```

Figure 6: Naive implementation in `solvers/IPCS.py` of variational forms used for solving the momentum equation (11), pressure correction (12) and momentum update (13).

mented as shown in Fig. 7. The remaining default functions are left to do nothing, as implemented already in `solvers/_init_.py`, and as such these 4 functions shown in Figs. 6 and 7 are all it takes to complete the implementation of the naive incremental pressure correction solver. Note that this implementation works for any order of the velocity/pressure function spaces. There is simply no additional implementation cost for using higher order elements.

5. High-performance implementation

The naive solver described in the previous section is very easy to implement and understand, but for obvious reasons it is not very fast. For example, the entire coefficient matrix is reassembled each timestep (see Fig. 4), even though it is only the convection term that changes in time. We will now explain how the same incremental pressure correction solver can be implemented efficiently, at the cost of losing some intuitiveness. The implementation of the high-performance solver described in this section can be found in `solvers/IPCS_ABCN.py`.

```

def velocity_tentative_solve(ui, F, q_, bcs,
                            **NS_namespace):
    A, L = system(F[ui])
    solve(A == L, q_[ui], bcs[ui])

def pressure_solve(Fp, p_, bcs, phi_,
                  **NS_namespace):
    # Compute pressure
    phi_.vector()[:] = p_.vector()
    A, L = system(F[ui])
    solve(A == L, p_, bcs['p'])

    # Normalize pressure if no bcs['p']
    if bcs['p'] == []:
        normalize(p_.vector())

    # Compute correction
    phi_.vector()[:] = p_.vector() -
        phi_.vector()

def velocity_update(u_components, q_, bcs,
                  Fu, **NS_namespace):
    for ui in u_components:
        A, L = system(F[ui])
        solve(A == L, q_[ui], bcs[ui])

```

Figure 7: Implementation in `solvers/IPCS.py` of routines called in Fig. 4.

The most significant steps in the optimization can roughly be split into four contributions: (i) pre-assembling of constant matrices making up the variational forms, (ii) efficient assembly of the entire coefficient matrix, where in an intermediate form it is used also to compute large parts of the linear right hand side, (iii) use of constructed (constant) matrices for assembling terms on right hand side through fast matrix vector products and (iv) efficient use and re-use of iterative solvers with preconditioners.

To implement an efficient solver we need to split up the variational forms (11), (12) and (13) term by term and view the equations on an algebraic level. The finite element solution, which is the product of the solver, is then written as

$$u_k^I = \sum_{j=1}^{N_u} \mathcal{U}_j^{k,I} \phi_j, \quad (15)$$

where ϕ_j are the basis functions and $\{\mathcal{U}_j^{k,I}\}_{j=1}^{N_u}$ are the N_u degrees of freedom.

We start by inserting for the tentative velocity u_k^I and $v = \phi_i$ in the bilinear terms of the variational

form (11)

$$\int_{\Omega} u_k^I v \, dx = \sum_{j=1}^{N_u} \left(\int_{\Omega} \phi_j \phi_i \, dx \right) \mathcal{U}_j^{k,I}, \quad (16)$$

$$\int_{\Omega} \nabla u_k^I \cdot \nabla v \, dx = \sum_{j=1}^{N_u} \left(\int_{\Omega} \nabla \phi_j \cdot \nabla \phi_i \, dx \right) \mathcal{U}_j^{k,I}. \quad (17)$$

Each term inside the parenthesis on the right hand side represents a matrix

$$M_{ij} = \int_{\Omega} \phi_j \phi_i \, dx, \quad (18)$$

$$K_{ij} = \int_{\Omega} \nabla \phi_j \cdot \nabla \phi_i \, dx. \quad (19)$$

The two matrices are independent of time and can be preassembled once through (\mathbf{u} , \mathbf{v} are trial and test functions respectively)

```
M = assemble(inner(u, v)*dx)
K = assemble(inner(grad(u), grad(v))*dx)
```

Note that the solution vectors and matrices represent the major cost in terms of memory use for the solver. The matrices are sparse and allocated by the linear algebra backend, using appropriate wrappers that are hidden to the user. The allocation takes place just once, when the matrices/vectors are created.

The nonlinear convection form contains the evolving solution and requires special attention. We use the implicit convection form given in Eq. (9) and write out the implicit Crank-Nicolson convected velocity for component k

$$\bar{\mathbf{u}} \cdot \nabla \tilde{u}_k = \frac{1}{2} \bar{\mathbf{u}} \cdot \nabla (u_k^I + u_k^{n-1}). \quad (20)$$

Inserting for the algebraic form of the finite element trial and test functions, the variational form for the bilinear convection term becomes

$$\int_{\Omega} \bar{\mathbf{u}} \cdot \nabla u_k^I v \, dx = \sum_{j=1}^{N_u} \left(\int_{\Omega} \bar{\mathbf{u}} \cdot \nabla \phi_j \phi_i \, dx \right) \mathcal{U}_j^{k,I}, \quad (21)$$

where $\bar{\mathbf{u}} = 1.5 \mathbf{u}^{n-1} - 0.5 \mathbf{u}^{n-2}$. The convection matrix can be recognized as the term inside the parenthesis

$$C_{ij}^{n-1/2} = \int_{\Omega} \bar{\mathbf{u}} \cdot \nabla \phi_j \phi_i \, dx. \quad (22)$$

The convecting velocity is time-dependent and interpolated at $t^{n-1/2}$. As such, the convection matrix is also evaluated at $n - 1/2$ and needs to be

reassembled each timestep. To simplify notations, though, we have for the rest of this paper omitted the time notation on C_{ij} . The assembly of the C_{ij} matrix is prepared in the `setup` function:

```
# Defined in setup
u_ab = as_vector([Function(V) for i in
                  range(len(u_components))])
aconv = inner(v, dot(grad(u), u_ab))*dx
```

where `u_ab` is used as a container for the convecting velocity $\bar{\mathbf{u}}$. Note that `u_ab` is assembled (see Fig 8) before assembling the matrix C_{ij} , because this leads to code that is a factor 2 faster than simply using a form based on the velocity functions at the two previous levels directly (i.e., `aconv = inner(v, dot(grad(u), 1.5*u_1 - 0.5*u_2))*dx`).

Consider now the linear terms, where the known solution function is written as $u_k^{n-1} = \sum_{j=1}^{N_u} \mathcal{U}_j^{k,n-1} \phi_j$, where $\mathcal{U}_j^{k,n-1}$ are the known coefficients of velocity component k at the previous time step t^{n-1} . We have the following linear terms in Eq. (11)

$$\int_{\Omega} u_k^{n-1} v \, dx = M_{ij} \mathcal{U}_j^{k,n-1}, \quad (23)$$

$$\int_{\Omega} \nabla u_k^{n-1} \cdot \nabla v \, dx = K_{ij} \mathcal{U}_j^{k,n-1}, \quad (24)$$

$$\int_{\Omega} \bar{\mathbf{u}} \cdot \nabla u_k^{n-1} v \, dx = C_{ij} \mathcal{U}_j^{k,n-1}, \quad (25)$$

that are all very quickly computed using simple matrix vector products.

We may now reformulate our variational problem on the algebraic level using the three assembled matrices. It is required that for each test function $v = \phi_i$, $i = 1, \dots, N_u$, the following equations must hold

$$\frac{M_{ij} (\mathcal{U}_j^{k,I} - \mathcal{U}_j^{k,n-1})}{\Delta t} + \frac{C_{ij} (\mathcal{U}_j^{k,I} + \mathcal{U}_j^{k,n-1})}{2} + \nu \frac{K_{ij} (\mathcal{U}_j^{k,I} + \mathcal{U}_j^{k,n-1})}{2} = \Phi_i^{k,n-1/2}, \quad (26)$$

where

$$\Phi_i^{k,n-1/2} = \int_{\Omega} \left(-\nabla_k p^* + f_k^{n-1/2} \right) \phi_i \, dx. \quad (27)$$

If separated into bilinear and linear terms, the fol-

lowing system of algebraic equations is obtained

$$\begin{aligned} \left(\frac{M_{ij}}{\Delta t} + \frac{C_{ij}}{2} + \nu \frac{K_{ij}}{2} \right) \mathcal{U}_j^{k,I} = \\ \left(\frac{M_{ij}}{\Delta t} - \frac{C_{ij}}{2} - \nu \frac{K_{ij}}{2} \right) \mathcal{U}_j^{k,n-1} + \Phi_i^{k,n-1/2}. \end{aligned} \quad (28)$$

If now $A_{ij} = M_{ij}/\Delta t + C_{ij}/2 + \nu K_{ij}/2$ is used as the final coefficient matrix, then the equation may be written as

$$A_{ij} \mathcal{U}_j^{k,I} = \left(\frac{2M_{ij}}{\Delta t} - A_{ij} \right) \mathcal{U}_j^{k,n-1} + \Phi_i^{k,n-1/2}, \quad (29)$$

or

$$A_{ij} \mathcal{U}_j^{k,I} = b_i^{k,n-1/2}, \quad \text{for } k = 1, \dots, d, \quad (30)$$

where $b_i^{k,n-1/2}$ is the right hand side of (29). Note that the same coefficient matrix is used by all velocity components, even when there are Dirichlet boundary conditions applied.

An efficient algorithm (2) can now be designed to assemble both large parts of the right hand side and the left hand side of Eq. (30) at the same time.

$$\begin{aligned} \text{Assemble } A_{ij} &\leftarrow C_{ij} \\ A_{ij} &= M_{ij}/dt - A_{ij}/2 - \nu K_{ij}/2 \\ b_i^{k,n-1/2} &= f_i^{k,n-1/2} + A_{ij} \mathcal{U}_j^{k,n-1}, \\ &\quad \text{for } k = 1, \dots, d \\ A_{ij} &= -A_{ij} + 2M_{ij}/dt \end{aligned}$$

Algorithm 2: Efficient algorithm for assembling the coefficient matrix A_{ij} , where most of the right hand side of Eq. (30) is assembled in an intermediate step.

Algorithm (2) is implemented as shown in Fig. 8. At the end of this algorithm, most of $b_i^{k,n-1/2}$ (except from the pressure gradient) has been assembled and the coefficient matrix A_{ij} is ready to be used in Eq. (30). The convection matrix needs to be reassembled each new time step, but only on the first inner velocity pressure iteration since $\bar{\mathbf{u}}$ only contains old and known velocities, not the new u_k^n . For this reason the code in Fig. 8 is placed inside `assemble_first_inner_iter`, called in Fig. 4. Notice that there is no separate matrix used for

```
# assemble convecting velocity
for i, ui in enumerate(u_components):
    u_ab[i].vector().zero()
    u_ab[i].vector().axpy(1.5, x_1[ui])
    u_ab[i].vector().axpy(-0.5, x_2[ui])

# assemble convection into A
A = assemble(a_conv, tensor=A,
             reset_sparsity=False)

# Negative convection on the rhs
A._scale(-0.5)

# Add mass and diffusion matrix
A.axpy(1./dt, M, True)
A.axpy(-0.5*nu, K, True)

# Compute parts of rhs vector
for ui in u_components:
    b_tmp[ui].zero()
    # Add body force stored in b0
    b_tmp[ui].axpy(1., b0[ui])
    # Add transient, convection and diffusion
    b_tmp[ui].axpy(1., A*x_1[ui])

# Reset matrix for lhs
A._scale(-1.)
A.axpy(2./dt, M, True)

# Apply boundary conditions
[bc.apply(A) for bc in bcs['u0']]
```

Figure 8: Inside `assemble_first_inner_iter`. Fast assemble of coefficient matrix and parts of right hand side vector. A temporary rhs vector `b_tmp` is used for each velocity component since this routine is called only on the first inner iteration. `x_1` is the vector of degrees of freedom at t^{n-1} .

$2M_{ij}/\Delta t - A_{ij}$ or C_{ij} and the total memory cost of the algorithm is exactly three individual sparse matrices (A_{ij} , M_{ij} and K_{ij}). The sparsity pattern of the matrices is computed on the first assemble and the matrix axpy operations take advantage of the fact that all these matrices share the same pattern.

The linear term $\Phi_i^{k,n-1/2}$ needs some further comments. Neglecting the constant forcing, \mathbf{f} , the second part of $\Phi_i^{k,n-1/2}$ is

$$\int_{\Omega} -\nabla_k P^* \phi_i dx, \quad (31)$$

where $p^* = \sum_{j=1}^{N_p} \mathcal{P}_j^* \hat{\phi}_j$, $\hat{\phi}_j$ is the basis function for the pressure and \mathcal{P}_j^* are the known degrees of

freedom. On algebraic form we get

$$\begin{aligned} \int_{\Omega} \nabla_k \mathcal{P}^* \phi_i \, dx &= \sum_{j=1}^{N_p} \left(\int_{\Omega} \nabla_k \hat{\phi}_j \phi_i \, dx \right) \mathcal{P}_j^*, \\ &= d\mathcal{P}_{ij}^k \mathcal{P}_j^*, \end{aligned} \quad (32)$$

where $d\mathcal{P}_{ij}^k$ for $k = 1, \dots, d$ are d matrices that are constant in time. Since the matrices can be pre-assembled, the computation of $\Phi_i^{k,n-1/2}$ through a matrix vector product is very fast. Unfortunately, though, three additional matrices require storage (in 3D), which may be too expensive. In that case there is a parameter in *Oasis* that can be used. Setting

```
NS_parameters["low_memory_version"] = False
```

enables the creation of the matrices $d\mathcal{P}_{ij}^k$. If disabled the term is computed simply through

```
assemble(inner(p_.dx(k), v)*dx)
```

for $k = 0, \dots, d-1$. The pressure gradient is added to b^k in `velocity_tentative_assemble` and not in Fig. 8, since the pressure is modified on inner iterations.

The pressure correction equation can also be optimized on the algebraic level. Using trial function $p^{n-1/2} = \sum_{j=1}^{N_p} \mathcal{P}_j^{n-1/2} \hat{\phi}_j$ and test function $q = \hat{\phi}_i$ we can write (12) for each test function

$$\hat{K}_{ij} \mathcal{P}_j^{n-1/2} = \hat{K}_{ij} \mathcal{P}_j^* - \int_{\Omega} \frac{\nabla \cdot \mathbf{u}^I}{\Delta t} \hat{\phi}_i \, dx. \quad (33)$$

The Laplacian matrix \hat{K}_{ij} can be preassembled. If the pressure function space is the same as the velocity function space, then $\hat{K}_{ij} = K_{ij}$ and no additional work is required. The divergence term may be computed as

$$\begin{aligned} \int_{\Omega} \frac{\nabla \cdot \mathbf{u}^I}{\Delta t} \hat{\phi}_i \, dx &= \frac{1}{\Delta t} \sum_{k=1}^d \left(\sum_{j=1}^{N_u} \int_{\Omega} \nabla_k \phi_j \hat{\phi}_i \, dx \mathcal{U}_j^{k,I} \right), \\ &= \frac{1}{\Delta t} \sum_{k=1}^d d\mathcal{U}_{ij}^k \mathcal{U}_j^{k,I}, \end{aligned} \quad (34)$$

where the matrices $d\mathcal{U}_{ij}^k$ for $k = 1, \dots, d$ can be preassembled. Again, the cost is three additional sparse matrices, unless the function spaces of pressure and velocity are the same. In that case $d\mathcal{U}_{ij}^k = d\mathcal{P}_{ij}^k$ and memory can be saved. If the `low_memory_version` is chosen, then we simply use the slower finite element assembly

```
assemble((1/dt)*div(u_)*q*dx)
```

The final step for the fractional step solver is the velocity update that can be written for component k as

$$M_{ij} \mathcal{U}_j^{k,n} = M_{ij} \mathcal{U}_j^{k,I} - \Delta t d\mathcal{P}_{ij}^k \mathcal{P}_j^{n-1/2}, \quad (35)$$

where $\mathcal{U}_j^{k,I}$ and $\mathcal{P}_j^{n-1/2}$ now are the known degrees of freedom of tentative velocity and pressure respectively, whereas $\mathcal{U}_j^{k,n}$ represent the unknowns. The velocity update requires a linear algebra Krylov or direct solve and as such it is quite expensive even though the equation is cheap to assemble. For this reason the velocity update has an additional option to use either a weighted gradient matrix³ \mathcal{G}_{ij}^k or lumping of the mass matrix, that allows the update to be performed directly

$$\mathcal{U}_i^{k,n} = \mathcal{U}_i^{k,I} - \Delta t \mathcal{G}_{ij}^k \mathcal{P}_j^{n-1/2}, \text{ for } i = 1, \dots, N_u. \quad (36)$$

The parameter used to enable the direct approach is `NS_parameters["velocity_update_type"]` that can be set to "gradient_matrix" or "lumping".

6. Verification of implementation

The fractional step algorithm implemented in `NSfracStep` is targeting transient flows in large-scale applications, with turbulent as well as laminar or transitional flow. It is not intended to be used as a steady state solver.⁴ *Oasis* has previously been used to study, e.g., blood flow in highly complex intracranial aneurysms [21, 22], where the results compare very well with, e.g., the spectral element code NEKTAR [23]. Simulations by Steinman and Valen-Sendstad [22] are also commented by Ventikos [24], who state this is "the right way to do it" - referring to the need for highly resolved CFD simulations of transitional blood flow in aneurysms.

Considering the end use of the solver in biomedical applications and research, it is essential that we establish the accuracy as well as the efficiency of the solver.

6.1. 2D Taylor Green flow

Two dimensional Taylor-Green flow is one of very few non-trivial analytical and transient solutions to the Navier-Stokes equations. For this

³Requires the *fenicstools* [20] package.

⁴As of 14 Sep 2014 *Oasis* ships with a coupled steady state solver for this purpose.

reason it is often used for verification of computer codes. The implementation can be found in `Oasis/problems/TaylorGreen.py` and the Taylor Green solution reads

$$\mathbf{u}_e = \left(-\sin(\pi y) \cos(\pi x) \exp(-2\pi^2 \nu t), \quad (37)$$

$$\sin(\pi x) \cos(\pi y) \exp(-2\pi^2 \nu t) \right), \quad (38)$$

$$p_e = -\frac{1}{4} (\cos(2\pi x) + \cos(2\pi y)) \exp(-4\pi^2 \nu t), \quad (39)$$

on the doubly periodic domain $(x, y) = [0, 2] \times [0, 2]$. The analytical solution is used to initialize the solver and to compute the norms of the error, i.e., $\|\mathbf{u} - \mathbf{u}_e\|_h$ and $\|p - p_e\|_h$, where $\|\cdot\|_h$ represents an L2 error norm. The mesh consists entirely of right triangles and is uniform in both spatial directions. The mesh size h is computed as two times the circumradius of a triangle. The kinematic viscosity is set to $\nu = 0.01$ and time is integrated for $t = [0, 1]$ with a short timestep ($\Delta t = 0.001$) to practically eliminate temporal integration errors. The solver is run for a range of mesh sizes and the order of convergence is shown in Table 1. The velocity is either piecewise quadratic (P2) or piecewise linear (P1), whereas the pressure is always piecewise linear. The P2P1 solver achieves fourth order accuracy in velocity and second order in pressure, whereas the P1P1 solver is second order accurate in both. Note that the fourth order in velocity is due to superconvergence [25] and it will drop to three for a mesh that is not regularly sized and aligned with the coordinate axis. The order of the L2 error (k) is computed by comparing the error norm of two consecutive discretization levels i and $i-1$, and assuming that the error can be written as $E_i = Ch_i^k$, where C is an arbitrary constant. Comparing $E_i = Ch_i^k$ and $E_{i-1} = Ch_{i-1}^k$ we can isolate $k = \ln(E_i/E_{i-1})/\ln(h_i/h_{i-1})$.

To verify the convergence of the transient fractional step scheme, we isolate temporal errors by practically eliminating spatial discretization errors through the use of high order P4 and P3 elements for velocity and pressure respectively. The solver is then run for a range of time step sizes for $t = [0, 6]$. The error norms at the end of the runs are shown in Table 2 indicating that both pressure and velocity achieve second order accuracy in time. Note that in Table 2, the order of the error is computed from $E_i = Cdt_i^k$, where dt_i is the time step used at level i .

Table 1: Taylor-Green flow convergence errors $O(h^k)$, where h and k are mesh size and order of convergence respectively. $\|\cdot\|_h$ represents an L2 norm. The velocity is either quadratic (P2) or linear (P1), whereas the pressure is always linear (P1).

P2P1				
h	$\ \mathbf{u} - \mathbf{u}_e\ _h$	k	$\ p - p_e\ _h$	k
2.83E-01	2.14E-02	-	1.81E-02	-
1.41E-01	1.44E-03	3.89	5.49E-03	1.72
9.43E-02	2.84E-04	4.01	2.46E-03	1.97
7.07E-02	8.94E-05	4.01	1.39E-03	2.00
5.66E-02	3.65E-05	4.01	8.88E-04	2.00
P1P1				
h	$\ \mathbf{u} - \mathbf{u}_e\ _h$	k	$\ p - p_e\ _h$	k
2.83E-01	9.31E-03	-	4.97E-03	-
1.41E-01	2.36E-03	1.98	1.55E-03	1.68
9.43E-02	1.06E-03	1.98	7.12E-04	1.92
7.07E-02	5.98E-04	1.99	4.05E-04	1.97
5.66E-02	3.83E-04	1.99	2.60E-04	1.98

Table 2: Taylor-Green flow convergence errors $O(dt^k)$, where dt and k are time step and order of convergence respectively. The velocity uses Lagrange elements of degree four (P4), whereas the pressure uses third degree (P3).

P4P3				
dt	$\ \mathbf{u} - \mathbf{u}_e\ _h$	k	$\ p - p_e\ _h$	k
5.00E-01	5.08E-01	-	1.29E+00	-
2.50E-01	1.36E-01	1.91	2.97E-01	2.11
1.25E-01	3.42E-02	1.99	7.12E-02	2.06
6.25E-02	8.62E-03	1.99	1.77E-02	2.01
3.12E-02	2.17E-03	1.99	4.41E-03	2.00

6.2. Turbulent channel flow

The second test case is a direct numerical simulation⁵ of turbulent, fully developed, plane channel flow. The flow is bounded between two parallel planes located at $y = \pm 1$ and is periodic in the x and z directions. The flow is driven by an applied constant pressure gradient (forcing) in the

⁵A direct numerical simulation indicates a simulation where all scales of turbulence have been resolved.

x -direction. This flow has been studied extensively with numerous CFD-codes, often using spectral accuracy since it is of primary importance to capture the rate of dissipation of turbulent kinetic energy, allowing no (or very little) numerical diffusion. To verify our implementation we will here attempt to reproduce the classical simulations of Moser, Kim and Mansour (MKM, [26]) for $Re_\tau = 180$, based on the wall friction velocity $u_\tau = \sqrt{\nu \partial u / \partial y_{wall}}$. The computational box is of size $L_x = 4\pi, L_y = 2$ and $L_z = 4\pi/3$. The resolution of MKM was a box of size 128^3 , uniform in x and z -directions and skewed towards the walls using Chebyshev points in the y -direction. In this test we use one under-resolved box of size 64^3 and one of the same size as MKM to show convergence towards the correct solution. Since each hexahedron is further divided into 6 tetrahedrons, this corresponds to $6 \cdot 64^3$ and $6 \cdot 128^3$ finite elements⁶. MKM performed their simulations using spectral accuracy with Fourier representation in the periodic directions and a Chebyshev-tau formulation in the y -direction. Here we use piecewise linear Lagrange elements (P1P1) of second order accuracy. The creation of the mesh and boundary conditions in module `problems/Channel.py` is shown in Fig. 9. The sampling of statistics is performed using routines from the *fenicstools* [20] package and are not described in detail here. Reference is given to the complete source code in `problems/Channel.py` in the *Oasis* repository. Figure 10 shows the statistically converged mean velocity in the x -direction across the channel normalized by u_τ . The black curve shows the spectral solution of MKM. The dashed and dotted curves show, respectively, the *Oasis* solution using $6 \cdot 64^3$ and $6 \cdot 128^3$ computational cells. The coarse solution represents an under-resolved simulation where the sharpest velocity gradients cannot be captured. The total amount of dissipation within the flow is thus under-predicted and the mean predicted velocity is consequently higher than it should be. This result is in agreement with the understanding of under-resolved Large Eddy Simulations (LES) of turbulent flows, that in effect adds viscosity to the large resolved scales to counteract the lack of dissipation from the unresolved small scales. Hence, simply by increasing the kinematic viscosity, the predicted mean flow could be forced closer to the spectral

⁶Due to two periodic directions the number of degrees of freedom for the fine mesh are $128 \cdot 129 \cdot 128$ for each velocity component and pressure, which is the same as used by MKM.

```

from numpy import arctan, pi
N = 128
Lx, Ly, Lz = 2.0*pi, 1.0, 2.0*pi/3.0
def mesh(Nx, Ny, Nz, **params):
    m = BoxMesh(0, -Ly, -Lz, Lx, Ly, Lz,
                N, N, N)
    x = m.coordinates()
    x[:, 1] = arctan(pi*(x[:, 1])) /
              arctan(pi)

nu = 2.e-5
Re_tau = 395.
NS_parameters.update(
    nu = nu,
    Re_tau = Re_tau,
    dt = 0.05,
    velocity_degree = 1,
    folder = "channel_results",
    use_krylov_solvers = True)

def walls(x, on_boundary):
    return (on_boundary and
            near((x[1]+Ly)*(x[1]-Ly), 0.0))

def create_bcs(V, u_components, **NS_name):
    bc = {ui: [DirichletBC(V, 0, walls)]
           for ui in u_components}
    bcs['p'] = []
    return bcs

utau = nu * Re_tau
def body_force(**NS_namespace):
    return Constant((utau**2, 0., 0.))

```

Figure 9: Implementation of the Channel problem.

DNS solution seen in Fig. 10. Another option is, of course, to refine the mesh and thereby resolve the smallest scales. As expected, we see in Fig. 10 that the $6 \cdot 128^3$ simulations are in much closer agreement with the spectral DNS. There is still a slight mismatch, though, that should be attributed to the lower order of the *Oasis* solver, incapable of capturing all the finest scales of turbulence. It is worth mentioning that the Galerkin finite element method used by *Oasis* contains no, or very little, numerical diffusion. A dissipative solver, like, e.g., a finite volume using an upwind scheme or a monotonically integrated implicit LES [27], would have the same effect as a LES model that adds viscosity and as such could lead to coarse simulations with mean velocity profiles closer to MKM.

Figure 11 shows the normal, non-dimensionalized, Reynolds stresses. The results confirm that the under-resolved stresses are under-predicted close to the wall, whereas the fine simulations converge towards the spectral MKM

results.

The channel simulations do not require more computational power than can be provided by a relatively new laptop computer. However, since these simulations are run for more than 30000 timesteps to sample statistics, we have performed parallel computations on the Abel supercomputer at the University of Oslo. The simulations scale weakly when approximately 200,000 elements are used per CPU, and thus we have run our simulations using 8 CPUs for the coarse mesh and 64 for the fine, which contains 12.5 million tetrahedrons. Simulations for the fine grid take approximately 1.5-1.7 seconds real time per computational timestep depending on traffic and distribution on Abel (20-25 % lower for the coarse simulations) and thus close to 12 hours for the entire test (30000 timesteps). Approximately 75 % of the computing time is spent in the linear algebra backend's iterative Krylov solvers and assembling of the coefficient matrix, as detailed in Sec. 5, is responsible for most of the remaining time. The backend (here PETSc) and the iterative linear algebra solvers are thus key to performance. For the tentative velocity computations we have used a stabilized version of a biconjugate gradient squared solver [28] with a very cheap (fast, low memory) Jacobi preconditioner imported from method `get_solvers`, where it is specified as `KrylovSolver('bicgstab', 'jacobi')`. This choice is justified since the tentative velocity coefficient matrix is diagonally dominant due to the short timesteps, and each solve requires approximately 10 iterations to converge (the same number for coarse and fine). The pressure coefficient matrix represents a symmetric and elliptic system and thus we choose a solver based on minimal residuals [29] and the hypre [30] algebraic multigrid preconditioner (`KrylovSolver('minres', 'hypre_amg')`). The pressure solver uses an average of 6 iterations (same for coarse and fine) to converge to a given tolerance. The velocity update is computed using a lumped mass matrix and no linear algebra solver is thus required for this final step.

7. Concluding notes on performance

The computational speed of any implicit, large-scale Navier-Stokes solver is determined by many competing factors, but most likely it will be limited by hardware and by routines for setting up (assembly) and solving for its linear algebra subsystems. In *Oasis*, and many comparable Navier-Stokes

solvers, the linear algebra is performed through routines provided by a backend (here PETSc) and are thus arguably beyond our control. Accepting that we cannot do better than limits imposed by hardware and the backend, the best we can really hope for through high-level implementations is to eliminate the cost of assembly. In *Oasis* we take all conceivable measures to do just this, as well as even reducing the number of required linear algebra solves. As mentioned in the previous section, for the turbulent channel case with 12.5 mill. tetrahedrons, 75 % of the computational time was found spent inside very efficient Krylov solvers and we are thus, arguably, pushing at the very boundaries of what may be achieved by a solver developed with similar numerical schemes, using the same backend.

To further support this claim, without making a complete comparison in terms of accuracy, we have also set up and tested the channel simulations on Abel for two low-level, second order accurate, semi-implicit, fractional step solvers, OpenFOAM[18] and CDP [31], that both are targeting high performance on massively parallel clusters. We used *channelFoam*, distributed with OpenFOAM version 2.2.1 [32] and 2.5.0 version of CDP (requires license). The *channelFoam* LES solver was modified slightly to run with constant viscosity, and parameters were set to match the finest channel simulations using 128^3 hexahedral cells. OpenFOAM used for the tentative velocity a biconjugate gradient [28] solver with a diagonal incomplete-LU preconditioner. For the pressure a conjugate gradient solver was used with a diagonal incomplete Cholesky preconditioner. The CDP solver was set up with the same hexahedral mesh as OpenFOAM, using no model for the LES subgrid viscosity. The linear solvers used by CDP were very similar to those used by *Oasis*, with a Jacobi preconditioned biconjugate gradient solver for the tentative velocity and a generalized minimum residual method [33] with the hypre algebraic multigrid preconditioner. Depending on traffic on Abel, both CDP and *channelFoam* required approximately 1.4-1.7 seconds real time per timestep, which is very close to the speed obtained by *Oasis*. For both CDP and OpenFOAM speed was strongly dominated by the Krylov solvers and both showed the same type of weak scaling as *Oasis* on the Abel supercomputer.

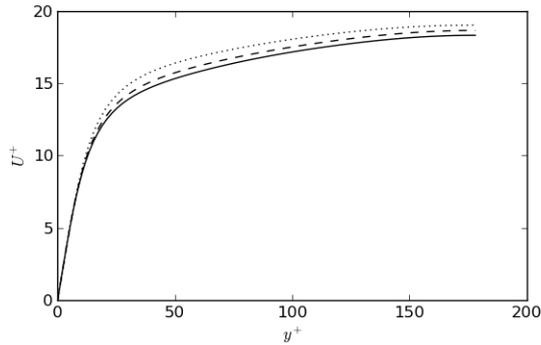


Figure 10: Mean velocity in x -direction normalized by u_τ as a function of scaled distance to the wall y^+ . Dotted and dashed curves are computed with *Oasis* using respectively $6 \cdot 64^3$ and $6 \cdot 128^3$ computational cells. The black curve is the reference solution from MKM.

Acknowledgements

This work has been supported by a Center of Excellence grant from the Research Council of Norway to the Center for Biomedical Computing at Simula Research Laboratory.

References

- [1] VMTK - The Vascular Modeling Toolkit.
URL <http://www.vmtk.org>
- [2] Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities.
URL <http://www.geuz.org/gmsh/>
- [3] The CUBIT geometry and mesh generation toolkit.
URL <https://cubit.sandia.gov>
- [4] S. Balay, et.al, PETSc Web page,
<http://www.mcs.anl.gov/petsc> (2013).
- [5] M. A. Heroux, et.al, An overview of the trilinos project, *ACM Trans. Math. Softw.* 31 (3) (2005) 397–423.
- [6] OpenFVM.
URL <http://openfvm.sourceforge.net/>
- [7] Fluidity.
URL imperial.ac.uk/earthscienceandengineering
- [8] Oofem - object oriented finite element solver.
URL <http://www.oofem.org/en/oofem.html>
- [9] FEniCS.
URL <http://fenicsproject.org>
- [10] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, G. N. Wells., Unified form language: A domain-specific language for weak formulations of partial differential equations, *ACM Transactions on Mathematical Software* 40 (2).
- [11] R. C. Kirby, A. Logg, A compiler for variational forms, *ACM Trans. Math. Softw.* 32 (3) (2006) 417–444. doi: 10.1145/1163641.1163644.
URL <http://doi.acm.org/10.1145/1163641.1163644>
- [12] FEniCS tutorial.
URL <http://fenicsproject.org/documentation/tutorial>

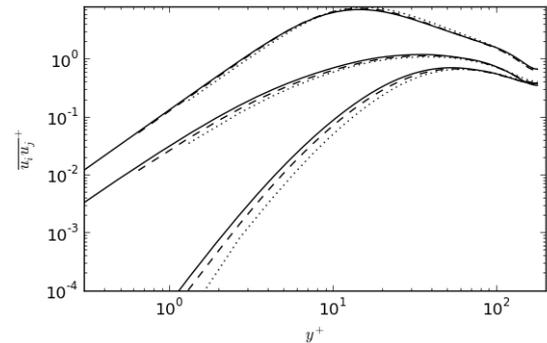


Figure 11: Normal Reynolds stresses scaled by u_τ^2 shown as functions of scaled distance to the wall y^+ . Dotted and dashed curves are computed with *Oasis* using respectively $6 \cdot 64^3$ and $6 \cdot 128^3$ computational cells. The black curves are from the reference solution of MKM. The three different profiles represent, in decreasing magnitude, the normal stresses \overline{uu}^+ , \overline{ww}^+ and \overline{vv}^+ , where u, v and w are velocity fluctuations in x, y and z directions respectively.

- [13] M. A. Christon, P. M. Gresho, S. B. Sutton, Computational predictability of time-dependent natural convection flows in enclosures (including a benchmark solution), *Int. J. for Num. Meth. in Fluids*.
- [14] J. Simo, 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 (1994) 111–154.
- [15] R. I. Issa, Solution of the implicitly discretized fluid flow equations by operator-splitting, *Journal of Computational Physics* 62 (1985) 40–65.
- [16] Ansys - fluent.
URL www.ansys.com
- [17] Star-CD.
URL <http://www.cd-adapco.com/products/star-cd>
- [18] OpenFOAM - The open source CFD toolbox.
URL www.openfoam.com
- [19] Oasis user manual.
URL <https://github.com/mikaem/Oasis/blob/master/doc/usermanual.pdf>
- [20] fenicstools.
URL <https://github.com/mikaem/fenicstools>
- [21] D. A. Steinman, et.al, Variability of computational fluid dynamics solutions for pressure and flow in a giant aneurysm: The asme 2012 summer bioengineering conference cfd challenge, *Journal of Biomechanical Engineering* 135 (2).
- [22] K. Valen-Sendstad, D. A. Steinman, Mind the gap: Impact of computational fluid dynamics solution strategy on prediction of intracranial aneurysm hemodynamics and rupture status indicators, *American Journal of Neuroradiology* 35 (3) (2014) 536–543.
- [23] NEKTAR.
URL <http://www.imperial.ac.uk/ssherw/spectralhp/nektar>
- [24] Y. Ventikos, Resolving the issue of resolution (2014).
- [25] F. Guillén-gonzález, G. Tierra, Superconvergence in ve-

- locity and pressure for the 3D time-dependent Navier-Stokes Equations, *SeMA Journal* 57 (1) (2012) 49–67.
- [26] R. D. Moser, J. Kim, N. N. Mansour, Direct numerical simulation of turbulent channel flow up to $re_{\tau} = 590$, *Phys. Fluids* 11 (1999) 943–945.
- [27] C. Fureby, F. F. Grinstein., Monotonically integrated large eddy simulation of free shear flows, *AIAA Journal* 37 (5) (1999) 544–556.
- [28] H. van der Vorst, Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems, *SIAM Journal on Scientific and Statistical Computing* 13 (2) (1992) 631–644.
- [29] C. C. Paige, M. A. Saunders, Solution of sparse indefinite systems of linear equations, *SIAM J. Numerical Analysis* 12 (1975) 617–629.
- [30] Hypre.
URL <http://acts.nersc.gov/hypre/>
- [31] CDP.
URL <http://web.stanford.edu/group/cits/research/combustor/cdp.html>
- [32] github.com/openfoam/openfoam-2.1.x/tree/master/applications/solvers/incompressible/channelFoam.
URL github.com/OpenFOAM/OpenFOAM-2.1.x/tree/master/applications/solvers/incompressible/channelFoam
- [33] Y. Saad, M. Schultz, Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems, *SIAM Journal on Scientific and Statistical Computing* 7 (3) (1986) 856–869.