19 *A comparison of some finite element schemes for the incompressible Navier–Stokes equations*

By K. Valen-Sendstad, A. Logg, K.-A. Mardal, H. Narayanan and M. Mortensen

Numerical algorithms for the computation of fluid flow have been an active area of research for several decades and still remains an active area of research. As a result, there exists a large literature on discretization schemes for the incompressible Navier–Stokes equations, and it can be hard to judge which method works best for any particular problem. Furthermore, since the development of any particular discretization scheme is often a long process and tied to a specific implementation, comparisons of different methods are seldom made.

FEniCS is a flexible platform for the implementation of different kinds of schemes based on finite element methods. To illustrate the simplicity by which different schemes can be implemented in FEniCS, we have implemented a test consisting of six different schemes. All schemes have been tested on six different test problems to compare the schemes in terms of their accuracy and efficiency. The schemes we have implemented are Chorin's projection scheme by Chorin [1968] and Temam [1969], the incremental pressure correction scheme (IPCS) by Goda [1979], the consistent splitting scheme (CSS) by Guermond et al. [2006], a least-squares stabilized Galerkin scheme (G2) by Hoffman and Johnson [2007], and a saddle-point solver based on a Richardson iteration on the pressure Schur complement (GRPC) as described in Turek [1996].

All solvers and test problems have been implemented in Python (with a few C++ extensions) using DOLFIN. The source code for all solvers and test problems is available online[1] and can be used to reproduce all results shown in this chapter.

19.1 *Preliminaries*

We consider the incompressible Navier–Stokes equations with unit fluid density written in the form

$$\begin{aligned} \dot{u} + \nabla u\, u - \nabla \cdot \sigma &= f, \qquad (19.1) \\ \nabla \cdot u &= 0, \qquad (19.2) \end{aligned}$$

where σ is the Cauchy stress tensor which for a Newtonian fluid is defined as

$$\sigma(u, p) = 2\nu\varepsilon(u) - pI. \qquad (19.3)$$

Here, u is the unknown velocity vector, p is the unknown pressure, ν is the (kinematic) viscosity, f is the body force per unit volume, and $\epsilon(u)$ is the symmetric gradient:

$$\epsilon(u) = \frac{1}{2}(\nabla u + \nabla u^{\top}). \qquad (19.4)$$

The above quantities σ and ϵ may be defined as follows in DOLFIN/UFL:

[1] http://launchpad.net/nsbench/

Python code
```python
def epsilon(u):
    return 0.5*(grad(u) + grad(u).T)
```

Python code
```python
def sigma(u, p, nu):
    return 2*nu*epsilon(u) - p*Identity(u.cell().d)
```

In all discretization schemes below, V_h and Q_h refer to the discrete finite element spaces used to discretize the velocity u and pressure p, respectively. For all schemes except the G2 scheme, V_h is the space of vector-valued continuous piecewise quadratic polynomials, and Q_h is the space of scalar continuous piecewise linear polynomials (Taylor–Hood elements). For the G2 scheme, continuous piecewise linears are used for both the velocity and the pressure. We will further use h to denote the local mesh size, $k_n = t_n - t_{n-1}$ to denote the size of the local time step, and $D_t^n u_h$ to denote the discretized form of the time derivative $(u_h^n - u_h^{n-1})/k_n$. For all schemes, except the fully implicit schemes G2 and GRPC described below, the convective term is treated explicitly.

19.2 Implementation

We have implemented the solvers and test problems as two class-hierarchies in Python, where the base classes are `SolverBase` and `ProblemBase`, respectively. The solvers, derived from `SolverBase` implement the scheme, that is, they define the finite element spaces, assemble and solve linear systems, and perform time-stepping. Code from several solvers will be shown throughout this chapter. The problems, derived from the `ProblemBase` class, define the mesh, initial and boundary conditions, and other parameters.

A main script `ns` allows a user to solve a given problem with a given solver. All available problems and solvers may be listed by typing

Bash code
```bash
$ ns list
```

which results in the following output:

Bash code
```
Usage: ns problem solver

Available problems:

  drivencavity
  channel
  taylorgreen
  cylinder
  beltrami
  aneurysm

Available solvers:

  chorin
  css1
  css2
  ipcs
  g2
  grpc
```

The `ns` script accepts a number of optional parameters to enable refinement in space and time, storing the solution in VTK or DOLFIN XML format, computing stresses, or plotting the solution directly to screen. As an example, to solve the lid-driven cavity test problem using Chorin's method and plot the solution, one may issue the following command:

Scheme 1: Chorin's projection method

1. Compute a tentative velocity $u_h^\bigstar$ by solving

$$\langle D_t^n u_h^\bigstar, v\rangle + \langle \nabla u_h^{n-1}\, u_h^{n-1}, v\rangle + \langle \nu \nabla u_h^\bigstar, \nabla v\rangle = \langle f^n, v\rangle \quad \forall\, v \in V_h, \tag{19.7}$$

including any boundary conditions for the velocity.

2. Compute the corrected pressure p_h^n by solving

$$\langle \nabla p_h^n, \nabla q\rangle = -\langle \nabla\cdot u_h^\bigstar, q\rangle / k_n \quad \forall\, q \in Q_h, \tag{19.8}$$

including any boundary conditions for the pressure.

3. Compute the corrected velocity u_h^n by solving

$$\langle u_h^n, v\rangle = \langle u_h^\bigstar, v\rangle - k_n\langle \nabla p_h^n, v\rangle \quad \forall\, v \in V_h, \tag{19.9}$$

including any boundary conditions for the velocity.

Bash code
```
$ ns drivencavity chorin plot_solution=True
```

Another script named `bench` allows a user to iterate over all solvers for a given problem, over all problems for a given solver, or over all problems and all solvers. As an example, the following command may be used to solve the channel test problem with all solvers on a mesh refined twice:

Bash code
```
$ bench channel refinement_level=2
```

19.3 *Solvers*

In this section, we present an overview of the six different schemes that have been tested.

Chorin's projection method

This scheme, often referred to as a non-incremental pressure correction scheme was first proposed by Chorin [1968] and Temam [1969]. For simplicity, we will here refer to this scheme as *Chorin*. To solve the system of equations (19.1)–(19.2), the idea is to first compute a tentative velocity by neglecting the pressure in the momentum equation and then projecting the velocity onto the space of divergence free vector fields. The projection step is a Darcy problem for u_h^n and p_h^n:

$$\frac{u_h^n - u_h^\bigstar}{k_n} + \nabla p_h^n = 0, \tag{19.5}$$

$$\nabla\cdot u_h^n = 0, \tag{19.6}$$

which is in fact reducible to a Poisson problem $-\Delta p_h^n = -\nabla\cdot u_h^\bigstar / k_n$ for the corrected pressure p_h^n. This is summarized in Scheme 1 and the implementation is shown in Figure 19.1. We note that since the velocity correction step is implemented as the solution of a linear system (involving a mass matrix that has not been lumped), the discrete incompressibility constraint is not satisfied exactly. On the other hand, the Dirichlet boundary conditions for the velocity are applied strongly as part of the velocity correction step and are thus satisfied exactly (at the nodal points).

Python code

```
# Tentative velocity step
F1 = (1/k)*inner(u - u0, v)*dx \
   + inner(grad(u0)*u0, v)*dx \
   + nu*inner(grad(u), grad(v))*dx - inner(f, v)*dx
a1 = lhs(F1)
L1 = rhs(F1)

# Poisson problem for the pressure
a2 = inner(grad(p), grad(q))*dx
L2 = -(1/k)*div(us)*q*dx

# Velocity update
a3 = inner(u, v)*dx
L3 = inner(us, v)*dx - k*inner(grad(p1), v)*dx
```

Figure 19.1: Implementation of variational forms for the Chorin solver.

Scheme 2: Incremental pressure correction (IPCS)

1. Compute the tentative velocity $u_h^\star$ by solving

$$\langle D_t^n u_h^\star, v\rangle + \langle \nabla u_h^{n-1}\, u_h^{n-1}, v\rangle + \langle \sigma(u_h^{n-\frac{1}{2}}, p_h^{n-1}), \epsilon(v)\rangle + \langle p_h^{n-1} n, v\rangle_{\partial\Omega} - \langle \nu(\nabla u_h^{n-\frac{1}{2}})^\top n, v\rangle_{\partial\Omega} = \langle f^n, v\rangle \tag{19.10}$$

for all $v \in V_h$, including any boundary conditions for the velocity. Here, $u_h^{n-\frac{1}{2}} = (u_h^\star + u_h^{n-1})/2$.

2. Compute the corrected pressure p_h^n by solving

$$\langle \nabla p_h^n, \nabla q\rangle = \langle \nabla p_h^{n-1}, \nabla q\rangle - \langle \nabla \cdot u_h^\star, q\rangle / k_n, \tag{19.11}$$

including any boundary conditions for the pressure.

3. Compute the corrected velocity u_h^n by solving

$$\langle u_h^n, v\rangle = \langle u_h^\star, v\rangle - k_n \langle \nabla(p_h^n - p_h^{n-1}), v\rangle \quad \forall\, v \in V_h, \tag{19.12}$$

including any boundary conditions for the velocity.

Incremental pressure correction scheme (IPCS)

An improvement of the non-incremental pressure correction scheme is possible if the previous value for the pressure is used to compute the tentative velocity. This idea was first introduced by Goda [1979]. The IPCS scheme is summarized in Scheme 2 and the implementation is shown in Figure 19.2. The IPCS scheme as implemented here also differs from the Chorin scheme in that the viscous term is evaluated at $(t_{n-1} + t_n)/2$ and a stress formulation is used in place of the Laplacian formulation used for the Chorin scheme. Note the importance of the term $\langle \nu(\nabla u_h^{n-1/2})^\top n, v\rangle_{\partial\Omega}$ which arises as a result of integrating the stress term by parts. Without this term, an incorrect velocity profile is obtained at inlets and outlets where the velocity will tend to "creep" around the corners.

Consistent splitting scheme (CSS)

The consistent splitting scheme, as described in Guermond et al. [2006], Guermond and Shen [2003], is derived differently from the other splitting schemes and requires a more detailed description. The scheme is based on deriving an equation for the pressure p by testing the momentum equation (19.1) against ∇q. In combination with the incompressibility constraint, an equation for the pressure results. After solving for the pressure, the velocity is updated based solely on the momentum equation by an appropriate approximation (extrapolation) of the pressure. The derivation of the consistent splitting scheme is as follows. Multiply the momentum equation (19.1) by ∇q for $q \in H^1(\Omega)$ and integrate over the domain Ω to obtain $\langle \dot{u} + \nabla u\, u - \nu\Delta u + \nabla p, \nabla q\rangle = \langle f, \nabla q\rangle$. Since $\langle \dot{u}, \nabla q\rangle = \langle \nabla\cdot\dot{u}, -q\rangle + \langle \dot{u}, qn\rangle_{\partial\Omega}$, it follows by (19.2) that

$$\langle \nabla p, \nabla q\rangle = \langle f - \nabla u\, u + \nu\Delta u, \nabla q\rangle, \tag{19.13}$$

if we assume that $\dot{u} = 0$ on $\partial\Omega$. Next, we use the identity $\Delta v \equiv \nabla\nabla\cdot v - \nabla\times\nabla\times v$ together with the incompressibility constraint (19.2) to write the diffusive term of (19.13) in *rotational form*:

$$\langle \nabla p, \nabla q\rangle = \langle f - \nabla u\, u - \nu\nabla\times\nabla\times u, \nabla q\rangle. \tag{19.14}$$

This equation is the basis for the consistent splitting scheme. At this point, we may formulate the CSS scheme as the solution of the following pair of variational problems:

$$\langle D_t^n u_h, v\rangle + \langle \nabla u_h^{n-1}\, u_h^{n-1}, v\rangle + \langle \nu\nabla u_h^n, \nabla v\rangle - \langle p_h^\star, \nabla\cdot v\rangle = \langle f^n, v\rangle, \tag{19.15}$$

$$\langle \nabla p_h^n, \nabla q\rangle = \langle f^n - \nabla u_h^{n-1}\, u_h^{n-1} - \nu\nabla\times\nabla\times u_h^n, \nabla q\rangle, \tag{19.16}$$

where $D_t^n u_h$ is an appropriate approximation of $\dot{u}_h$ and $p_h^\star$ is an appropriate approximation of the pressure. In the simplest case, one may chose $p_h^\star = p_h^{n-1}$ but higher order approximations are also possible. For example, one may take $p_h^\star$ to be the linear extrapolation of p_h from p_h^{n-2} and p_h^{n-1} given by $p_h^\star = p_h^{n-1} + (p_h^{n-1} - p_h^{n-2}) = 2p_h^{n-1} - p_h^{n-2}$. We will refer to the simplest approximation as CSS$_1$ and to the higher-order approximation as CSS$_2$.

To avoid having to compute the term $\nabla\times\nabla\times u_h^n$ in (19.16), we take the inner product of (19.15) with ∇q and subtract the result from (19.16) to obtain

$$\begin{aligned}\langle \nabla p_h^n - \nabla p_h^\star, \nabla q\rangle &= \langle D_t^n u_h - \nu\nabla\times\nabla\times u_h^n - \nu\Delta u_h^n, \nabla q\rangle \\ &= \langle D_t^n u_h - \nu\nabla\nabla\cdot u_h^n, \nabla q\rangle,\end{aligned} \tag{19.17}$$

where we have again used the identity $\Delta v \equiv \nabla\nabla\cdot v - \nabla\times\nabla\times v$. Finally, we define an auxiliary field $\psi_h^n = p_h^n - p_h^\star + \nu\nabla\cdot u_h^n$ to write (19.17) in the form

$$\langle \nabla\psi_h^n, \nabla q\rangle = \langle D_t^n u_h, \nabla q\rangle. \tag{19.18}$$

The CSS scheme is summarized in Scheme 3/4.

To solve for the auxiliary variable ψ, appropriate boundary conditions must be used. Since ψ is a *pressure correction* and not the pressure itself, we use homogenized versions of the pressure boundary conditions which are zero at the boundary in the case of Dirichlet boundary conditions. This can be accomplished in DOLFIN using the function `homogenize`.

We remark that the derivation of the consistent splitting scheme is based on the assumption that $\dot{u} = 0$ on $\partial\Omega$ which gives $\langle \dot{u}, \nabla q\rangle = -\langle \nabla\cdot u, q\rangle + \langle \dot{u}_{\partial\Omega}, qn\rangle = -\langle \nabla\cdot u, q\rangle$. For non-constant Dirichlet boundary conditions, this assumption is not valid. This issue is not addressed in Guermond and Shen [2003], but it is easy to add the missing term as shown in Figure 19.3 where the missing term is included in the linear form `L2`.

A least-squares stabilized Galerkin method (G2)

The G2 method is a stabilized finite element method using piecewise linear discretization in space and time. For further reading we refer to Hoffman and Johnson [2007]. In each time step, the G2 solution is defined by

$$\begin{aligned}\langle D_t^n u_h, v\rangle + \langle \nabla u_n^n\cdot w, v\rangle + \langle \sigma(u_h^{n-\frac{1}{2}}, p_h^n), \epsilon(v)\rangle - \langle \nu(\nabla u_h^{n-\frac{1}{2}})^\top n, v\rangle_{\partial\Omega} + \langle \bar{p}n, v\rangle_{\partial\Omega} + SD_\delta = \langle f^n, v\rangle, \\ \langle \nabla p_h^n, \nabla q\rangle = -\langle \nabla\cdot u_h^n/\delta_1, q\rangle,\end{aligned} \tag{19.23}$$

Python code

```
# Tentative velocity step
U = 0.5*(u0 + u)
F1 = (1/k)*inner(u - u0, v)*dx \
   + inner(grad(u0)*u0, v)*dx \
   + inner(sigma(U, p0, nu), epsilon(v))*dx \
   + inner(p0*n, v)*ds \
   - beta*nu*inner(grad(U).T*n, v)*ds \
   - inner(f, v)*dx
a1 = lhs(F1)
L1 = rhs(F1)

# Pressure correction
a2 = inner(grad(p), grad(q))*dx
L2 = inner(grad(p0), grad(q))*dx \
   - (1.0/k)*div(u1)*q*dx

# Velocity correction
a3 = inner(u, v)*dx
L3 = inner(u1, v)*dx - k*inner(grad(p1 - p0), v)*dx
```

Figure 19.2: Implementation of variational forms for the IPCS solver. The flag `beta = 1` is set to zero in the case when periodic boundary conditions are used.

Scheme 3/4: Consistent splitting

1. Compute the pressure approximation (extrapolation) $p_h^\bigstar$ by

$$p_h^\bigstar = \begin{cases} p_h^{n-1}, & \text{for CSS}_1, \\ 2p_h^{n-1} - p_h^{n-2}, & \text{for CSS}_2. \end{cases} \tag{19.19}$$

2. Compute the velocity u_h^n by solving

$$\langle D_t^n u_h, v\rangle + \langle \nabla u_h^{n-1}\, u_h^{n-1}, v\rangle + \langle \sigma(u_h^{n-\frac{1}{2}}, p_h^\bigstar), \epsilon(v)\rangle + \langle \bar{p}n, v\rangle_{\partial\Omega} - \langle \nu(\nabla \bar{u}_h^n)^\top n, v\rangle_{\partial\Omega} = \langle f^n, v\rangle, \tag{19.20}$$

including any boundary conditions for the velocity. Here, $u_h^{n-\frac{1}{2}} = (u_h^n + u_h^{n-1})/2$ and $\bar{p}$ is a given boundary condition for the pressure.

3. Compute the pressure correction ψ_h^n by solving

$$\langle \nabla \psi_h^n, \nabla q\rangle = \langle u_h^n - u_h^{n-1}, \nabla q\rangle / k_n - \langle u_h^n - u_h^{n-1}, qn\rangle_{\partial\Omega} / k_n \quad \forall q \in Q_h. \tag{19.21}$$

4. Compute the corrected pressure p_h^n by solving

$$\langle p_h^n, q\rangle = \langle p_h^\bigstar + \psi_h^n - \nu \nabla \cdot u_h^n, q\rangle \quad \forall q \in Q_h, \tag{19.22}$$

including any boundary conditions for the pressure.

Python code

```
# Tentative pressure
if self.order == 1:
    ps = p1
else:
    ps = 2*p1 - p0

# Tentative velocity step
F1 = (1/k)*inner(u - u0, v)*dx \
   + inner(grad(u0)*u0, v)*dx \
   + inner(sigma(u, ps, nu), epsilon(v))*dx \
   - beta*nu*inner(grad(u).T*n, v)*ds \
   + inner(pbar*n, v)*ds \
   - inner(f, v)*dx
a1 = lhs(F1)
L1 = rhs(F1)

# Pressure correction
a2 = inner(grad(p), grad(q))*dx
L2 = (1/k)*inner(u1 - u0, grad(q))*dx \
   - (1/k)*inner(u1 - u0, q*n)*ds

# Pressure update
a3 = p*q*dx
L3 = p1*q*dx + psi*q*dx - nu*div(u1)*q*dx
```

Figure 19.3: Implementation of variational forms for the CSS solver(s). The flag `beta = 1` is set to zero in the case when periodic boundary conditions are used.

for all $(v,q) \in V_h \times Q_h$, where $u^{n-\frac{1}{2}} = (u_h^n + u_h^{n-1})/2$ and

$$SD_\delta = \langle \delta_1 \nabla u_h^{n-\frac{1}{2}}\, u_h^{n-\frac{1}{2}}, \nabla v\, u_h^{n-\frac{1}{2}} \rangle + \langle \delta_2 \nabla \cdot u_h^{n-\frac{1}{2}}, \nabla \cdot v \rangle. \tag{19.24}$$

The G2 equations may be obtained by testing the incompressible Navier–Stokes equations against modified test functions $v \to v + \delta_1(\nabla v \cdot \bar{u}^n + \nabla q)$ and $q \to q + \delta_2 \nabla \cdot v$ and dropping a number of terms, including all stabilizing terms involving the time derivative $D_t^n u_h$. The stabilization parameters are set to $\delta_1 = \frac{\kappa_1}{2}(k_n^{-2} + |u^{n-1}|^2 h_n^{-2})^{-\frac{1}{2}}$ and $\delta_2 = \kappa_2 h_n$ in the convection dominated case, that is, if $\nu < uh$. In the diffusion dominated case, the parameters are set to $\delta_1 = \kappa_1 h_n^2$ and $\delta_2 = \kappa_2 h_n^2$. The constants κ_1 and κ_2 are here set to $\kappa_1 = 4$ and $\kappa_2 = 2$.

The discrete system of equations is solved by a direct fixed-point iteration between the velocity and pressure equations obtained by setting the test functions $q = 0$ and $v = 0$ respectively. Note that as a result of the stabilization, one obtains a Poisson equation for the pressure involving the stabilization parameter δ_1. The G2 scheme is summarized in Scheme 5 and the implementation is shown in Figure 19.4.

A saddle-point solver for a pure Galerkin discretization (GRPC)

Finally, we test a scheme based on a pure space-time Galerkin finite element discretization of the incompressible Navier–Stokes equations and iterative solution of the resulting saddle-point system. The saddle-point system is obtained by testing the momentum equation (19.1) against a test function $v \in V_h$ and the continuity equation (19.2) against a test function $q \in Q_h$ and integrating over $\Omega \times [t_{n-1}, t_n]$. This corresponds to a space-time discretization using continuous piecewise quadratic and linear polynomials in space (for V_h and Q_h respectively), and continuous piecewise linear polynomials in time (with discontinuous piecewise constant test functions in time). Integrating the stress term by parts, one obtains the following variational problem: find (u_h^n, p_h^n) in $V_h \times Q_h$ such that

$$\frac{1}{k_n}\langle u_h^n - u_h^{n-1}, v\rangle + \langle \nabla u_h^{n-\frac{1}{2}}\, u_h^{n-\frac{1}{2}}, v\rangle + \langle \sigma(u_h^{n-\frac{1}{2}}, p_h^n), \epsilon(v)\rangle - \langle \nu(\nabla u_h^{n-\frac{1}{2}})^\top \cdot n, v\rangle_{\partial\Omega} + \langle \bar{p} n, v\rangle = \langle f, v\rangle \tag{19.27}$$

$$\langle \nabla \cdot u_h^{n-\frac{1}{2}}, q\rangle = 0 \tag{19.28}$$

Scheme 5: G2

1. Compute stabilization parameters δ_1 and δ_2.
2. Repeat until convergence:

   (a) Update the pressure p_h^n by solving

$$\langle \nabla p_h^n, \nabla q \rangle = -\langle \nabla \cdot u_h^n / \delta_1, q \rangle \quad \forall q \in Q_h, \tag{19.25}$$

   including any boundary conditions for the pressure.

   (b) Update the velocity u_h^n by solving

$$\begin{aligned}\langle D_t^n u_h, v \rangle + \langle \nabla u_n^n \cdot w, v \rangle + \langle \sigma(u_h^{n-\frac{1}{2}}, p_h^n), \epsilon(v) \rangle - \langle \nu (\nabla u_h^{n-\frac{1}{2}})^\top n, v \rangle_{\partial\Omega} + \langle \bar{p} n, v \rangle_{\partial\Omega} \\ + \langle \delta_1 \nabla u_h^{n-\frac{1}{2}} \cdot w, \nabla v \cdot w \rangle + \langle \delta_2 \nabla \cdot u_h^{n-\frac{1}{2}}, \nabla \cdot v \rangle = \langle f^n, v \rangle\end{aligned} \tag{19.26}$$

   for all $v \in V_h$, including any boundary conditions for the velocity. Here, $u_h^{n-\frac{1}{2}} = (u_h^n + u_h^{n-1})/2$, $\bar{p}$ is a given boundary condition for the pressure, and w is an approximation of the velocity u_h^n from the previous iteration.

   (c) Compute a piecewise constant approximation w of u_h^n.

   (d) Compute the residuals of the momentum and continuity equations and check for convergence.

Python code

```python
# Velocity system
U = 0.5*(u0 + u)
P = p1
Fv = (1/k)*inner(u - u0, v)*dx \
   + inner(grad(U)*W, v)*dx \
   + inner(sigma(U, P, nu), epsilon(v))*dx \
   - beta*nu*inner(grad(U).T*n, v)*ds \
   + inner(pbar*n, v)*ds \
   - inner(f, v)*dx \
   + d1*inner(grad(U)*W, grad(v)*W)*dx \
   + d2*div(U)*div(v)*dx
av = lhs(Fv)
Lv = rhs(Fv)

# Pressure system
ap = inner(grad(p), grad(q))*dx
Lp = -(1/d1)*div(u1)*q*dx

# Projection of velocity
aw = inner(w, z)*dx
Lw = inner(u1, z)*dx
```

Figure 19.4: Implementation of variational forms for the G2 solver.

Scheme 6: GRPC

1. Repeat until convergence:

   (a) Assemble the residual vector R_U of the momentum equation.

   (b) Update the velocity vector U according to

   $$U := U - K^{-1}R_U. \tag{19.30}$$

   (c) Assemble the residual vector R_P of the continuity equation.

   (d) Update the pressure vector P according to

   $$P := P - \tau_1 L_1^{-1} R_P - \tau_2 L_2^{-1} R_P. \tag{19.31}$$

where $u_h^{n-\frac{1}{2}} = (u_h^n + u_h^{n-1})/2$ and $\bar{p}$ is a given boundary condition for the pressure. The resulting algebraic system of equations takes the form

$$\begin{bmatrix} M + \Delta t N(U) & \Delta t B \\ \Delta t B^T & 0 \end{bmatrix} \begin{bmatrix} U \\ P \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}, \tag{19.29}$$

where U and P are the vectors of degrees of freedom for u_h^n and p_h^n respectively, M is the mass matrix, N is a convection–diffusion operator (depending on U^n), B is the discrete gradient, and b is a vector depending on the solution on the previous time step, body forces and boundary conditions. Notice that we have multiplied the incompressibility constraint by Δt to obtain symmetry in case N is symmetric.

To solve this system of equations, we employ an algebraic splitting technique sometimes referred to as generalized Richardson iteration on the pressure Schur complement (GRPC) [Turek, 1999]. The convergence of this method depends critically on the efficiency of two preconditioners, K and L. The preconditioner K should approximate $M + \Delta t N$, while L which should approximate the pressure Schur complement $B^T(M + \Delta t N)^{-1}B$. It is well known that if an explicit scheme is used for convection, then order-optimal solution algorithms for both $M + \Delta t N$ and $B^T(M + \Delta t N)^{-1}B$ are readily available [Cahouet and Chabard, 1988, Turek, 1999, Mardal and Winther, 2004, 2011]. In fact $L^{-1} \approx \Delta t M_Q^{-1} + A_Q^{-1}$, where M_Q and A_Q are the mass and stiffness matrices associated with the pressure discretization. Hence, we let $L_1 = \frac{1}{\Delta t} M_Q$ and $L_2 = A_Q$ and approximate L^{-1} by $\tau_1 L_1^{-1} + \tau_2 L_2^{-1}$. For simplicity, we here let $\tau_1 = \tau_2 = 2$. For a further discussion on these preconditioners, we refer to the Chapter 30. In the implementation, we have chosen to exclude the convective term in the preconditioners K and L to avoid reassembly. The GRPC scheme is summarized in Scheme 6 and the implementation is shown in Figure 19.5.

Python code

```python
# Velocity and pressure residuals
U = 0.5*(u0 + u1)
P = p01
Ru = inner(u1 - u0, v)*dx \
   + k*inner(grad(U)*U, v)*dx \
   + k*inner(sigma(U, P, nu), epsilon(v))*dx \
   - beta*k*nu*inner(grad(U).T*n, v)*ds \
   + k*inner(pbar*n, v)*ds \
   - k*inner(f, v)*dx
Rp = k*div(U)*q*dx
```

Figure 19.5: Implementation of variational forms for the GRPC solver.

Table 19.1: Summary of test problems.

Problems	Functionals / norms
Driven cavity, 2D	Minimum of stream function at $t = 2.5$
Channel flow, 2D	Velocity u_x at $(x, y) = (1, 0.5)$ at $t = 0.5$
Flow past a cylinder, 2D	Pressure difference across cylinder at $t = 8$
Taylor–Green vortex, 2D	Kinetic energy at $t = 0.5$
Beltrami flow, 3D	Relative L^2 error in velocity at $t = 0.5$
Idealized aneurysm, 3D	Velocity u_x at $(x, y, z) = (0.025, -0.006, 0)$ at $t = 0.05$

19.4 Test problems and results

To test the accuracy and efficiency of Schemes 1–6, we apply the schemes to a set of test problems. For each test problem, we make an *ad hoc* choice for how to measure the accuracy; we either measure the error in a certain functional of interest or a norm of the global error. The choice of test problems and functionals clearly affects the conclusions one may draw regarding the schemes. However, together the six test problems should give a good indication of the accuracy and efficiency of the tested schemes. We emphasize that all schemes have been implemented in the same framework and with minor differences in their implementation to make a fair comparison. All test problems represent laminar flow for small to moderate size Reynolds numbers in the range 1–1000. The test problems are listed in Table 19.1.

Common parameters

For all solvers, the time step is chosen based on an approximate CFL condition $k = 0.2\,h/U$ where U is an estimate of the maximum velocity.

Comparisons of solvers are made by plotting the CPU time / seconds and error against the number of degrees of freedom. Since all solvers except the G2 solver use the same type of discretization ($\mathcal{P}_2$–$\mathcal{P}_1$), this is equivalent to plotting CPU times and errors against refinement level or mesh size for those solvers. However, since the G2 method uses a $\mathcal{P}_1$–$\mathcal{P}_1$ discretization, the graphs will change depending on whether the x-axis is given by the number of degrees of freedom or the mesh size. In particular, the G2 method will seem slower (but at the same time more accurate) when plotting against the number of degrees of freedom, while seeming to be faster (but at the same time less accurate) when plotting against mesh size.

All simulations have been performed on a Linux cluster on a single node with 8 GB of memory. The test problems have been solved several times and the recorded CPU times have been compared with previous runs to ensure that the results are not influenced by any “noise”.

To ensure accurate solution of linear systems, the absolute and relative tolerances for the DOLFIN (PETSc) Krylov solvers were set to 1e-25 and 1e-12 respectively. In all cases, the velocity system was solved using GMRES with ILU preconditioning and the pressure system was solved using GMRES with an algebraic multigrid preconditioner (Hypre). For the iterative methods G2 and GRPC, the tolerance for the main iteration was set to a value between 1e-6 to 1e-12 with higher values in cases where the convergence was slow (or non-existent).

Driven cavity (2D)

A classical benchmark problem for fluid flow solvers is the two-dimensional lid-driven cavity problem. We consider a square cavity with sides of unit length and kinematic viscosity $\nu = 1/1000$. No-slip boundary conditions are imposed on each edge of the square, except at the upper edge where the velocity is set to $u = (1, 0)$. Figure 19.6 shows the implementation of these boundary conditions in DOLFIN. The initial condition for the velocity is set to zero. The resulting flow is a vortex developing in the upper right corner and then traveling towards the center of the square as the flow evolves.

Python code

```
class BoundaryValue(Expression):
    def eval(self, values, x):
        if x[0] > DOLFIN_EPS and \
           x[0] < 1.0 - DOLFIN_EPS and \
           x[1] > 1.0 - DOLFIN_EPS:
            values[0] = 1.0
            values[1] = 0.0
        else:
            values[0] = 0.0
            values[1] = 0.0
```

Figure 19.6: Implementation of velocity boundary conditions for the driven cavity test problem.

As a functional of interest, we consider the minimum value of the stream function at final time $T = 2.5$. Reference values for this functional are available in Pandit et al. [2007], where a reference value of $\min \psi = -0.0585236$ is reported, and in Chudanov et al. [2007], where a value of $\min \psi = -0.058048$ is reported. These values differ already in the third decimal. To obtain a better reference value, we have therefore computed the solution using the spectral element code Semtex [Blackburn, 2011, Blackburn and Sherwin, 2004] with up to 80×80 10^{th} order elements, heavily refined in the area in the vicinity of the minimum of the stream function. The time-stepping for computing the reference solution was handled by a third order implicit discretization and a very short time step was used to minimize temporal errors. The resulting reference value for the minimum of the stream function was $\min \psi = -0.061076605$. This value differs remarkably much from the available reference values in the literature, but seems to be correct judging from the convergence plots for the different solvers in Figure 19.8.

Computing the stream function. The stream function is defined as

$$u_x = \frac{\partial \psi}{\partial y}, \quad u_y = \frac{\partial \psi}{\partial x}, \tag{19.32}$$

and can be computed by solving the Poisson problem

$$-\nabla^2 \psi = \omega, \tag{19.33}$$

where ω is the vorticity given by

$$\omega = \frac{\partial u_x}{\partial y} - \frac{\partial u_y}{\partial x}. \tag{19.34}$$

For a more thorough description, see White [1999] or White [1991]. Figure 19.7 shows how to compute the stream function in DOLFIN.

Results. Figure 19.8 shows the results for the driven cavity test problem. The smallest errors are obtained with the Chorin and GRPC schemes. The GRPC solver is also the slowest solver. We further observe a clear difference between CSS_1 and CSS_2.

Pressure-driven channel flow (2D)

As a second test problem, we seek the solution of the Navier–Stokes equations in a two-dimensional pressure-driven channel. The geometry of the channel is the unit square $[0, 1]^2$ and the kinematic viscosity is $\nu = 1/8$. No-slip boundary conditions are applied to the velocity at the upper and lower walls and Neumann boundary conditions are applied at the inlet and outlet. Dirichlet boundary conditions are applied to the pressure at the inlet and outlet, with $p = 1$ at the inlet and $p = 0$ at the outlet. The initial condition is $u = (0, 0)$ for the velocity. As a functional of interest, we consider the x-component of the

Python code

```
# Define variational problem
V   = u.function_space().sub(0)
psi = TrialFunction(V)
q   = TestFunction(V)
a   = dot(grad(psi), grad(q))*dx
L   = dot(u[1].dx(0) - u[0].dx(1), q)*dx

# Define boundary condition
g  = Constant(0)
bc = DirichletBC(V, g, DomainBoundary())

# Compute solution
problem = VariationalProblem(a, L, bc)
psi = problem.solve()
```

Figure 19.7: Computing the stream function in DOLFIN.

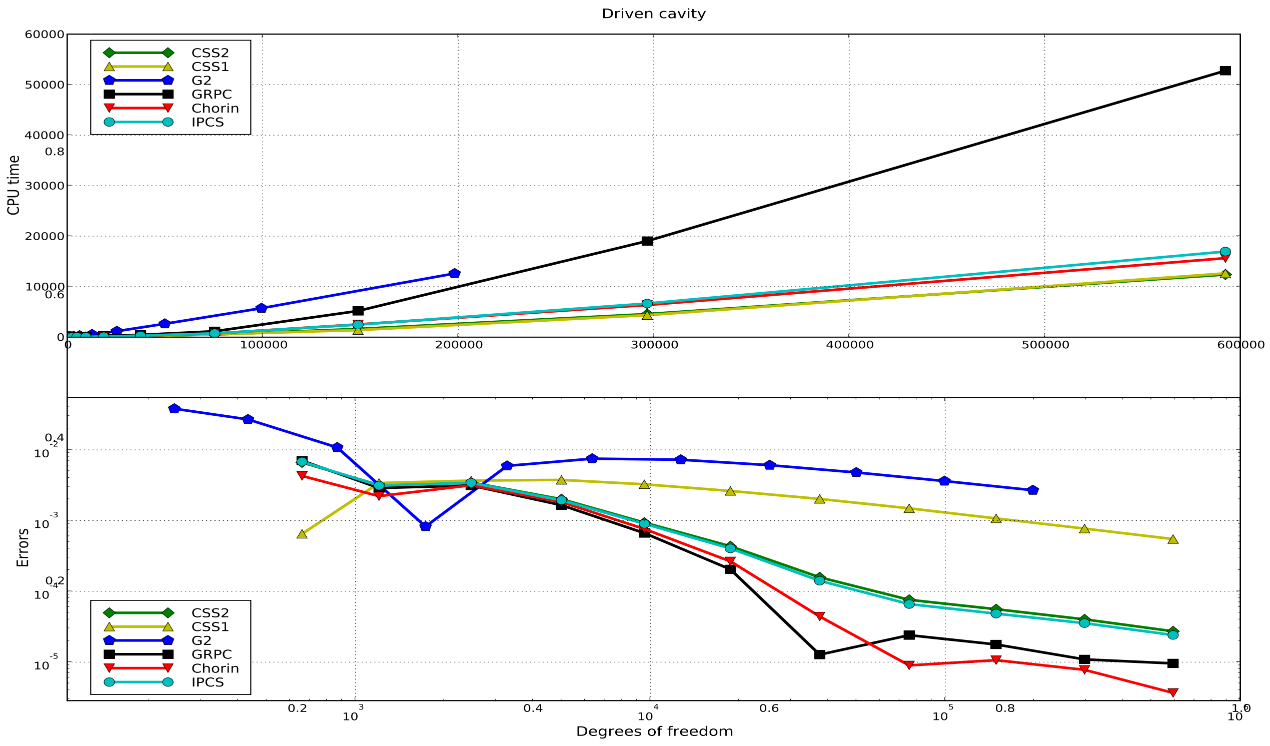


Figure 19.8: Results for the driven cavity test problem.

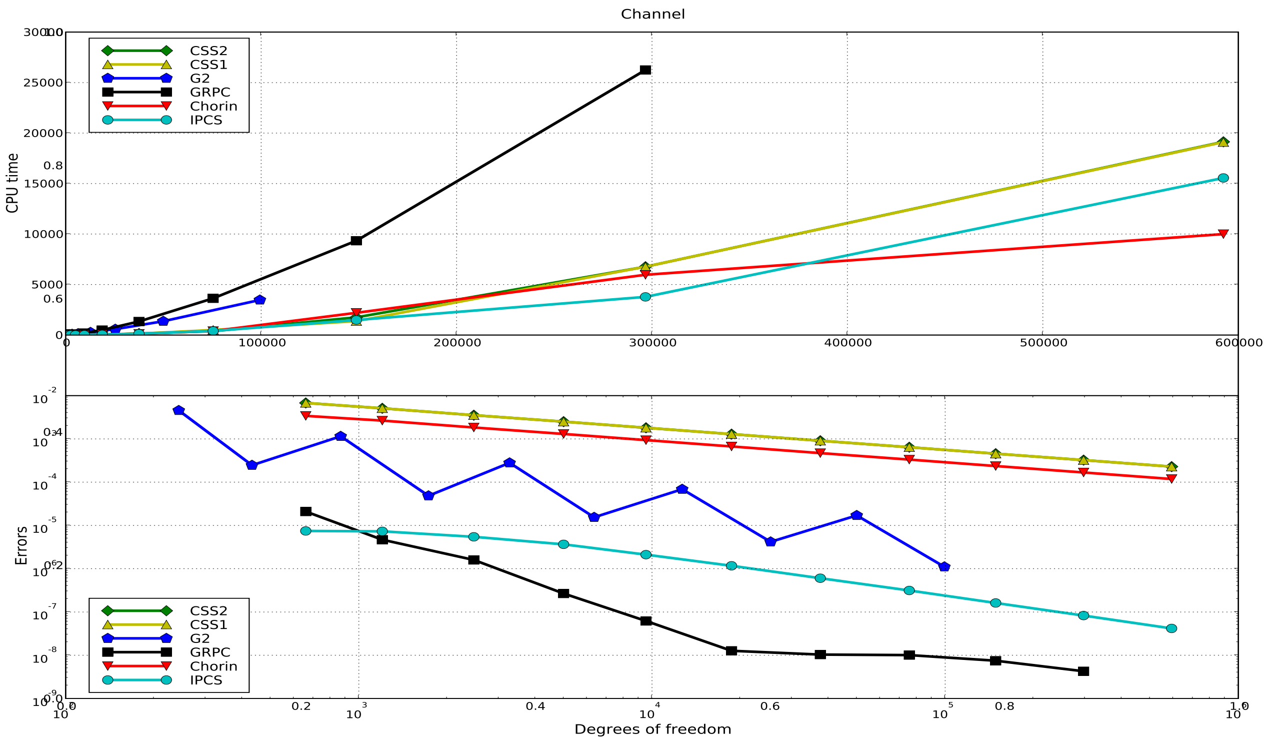


Figure 19.9: Results for the pressure-driven channel test problem.

velocity at $(x, y) = (1, 0.5)$ at final time $T = 0.5$. By a Fourier series expansion, it is easy to show that the exact value of the velocity at this point is given by

$$u_x(1, 0.5, t) = 1 - \sum_{n=1,3,\ldots}^{\infty} \frac{32}{\pi^3 n^3} e^{-\frac{\pi^2 n^2 t}{8}} (-1)^{(n-1)/2}. \tag{19.35}$$

At final time $T = 0.5$, this values is $u_x(1, 0.5, 0.5) \approx 0.44321183655681595$.

Results. Figure 19.9 shows the results for the pressure-driven channel test problem. Again, the smallest error is obtained with the GRPC solver, closely followed by the IPCS solver. The W-shaped curve for the G2 solver is an effect of the $\mathcal{P}_1$–$\mathcal{P}_1$ discretization which results in a vertex located at $(x, y) = (1, 0.5)$ only for every other refinement level.

Taylor–Green vortex (2D)

As our next test problem, we consider the Taylor–Green vortex described in Canuto et al. [2007], which is a periodic flow with exact solution given by

$$\begin{aligned} u(x,y,t) =& (\cos(\pi x)\sin(\pi y)e^{-2t\nu\pi^2}, \cos(\pi y)\sin(\pi x)e^{-2t\nu\pi^2}), \\ p(x,y,t) =& -0.25(\cos(2\pi x) + \cos(2\pi y))e^{-4t\nu\pi^2}, \end{aligned} \tag{19.36}$$

on the domain $[-1, 1]^2$. The kinematic viscosity is set to $\nu = 1/100$. Periodic boundary conditions are imposed in both the x and y directions. The implementation of these boundary conditions in DOLFIN is shown in Figure 19.10. The initial velocity and pressure fields are shown in Figure 19.11. As a functional of interest, we measure the kinetic energy $K = \frac{1}{2}\|u\|^2_{L^2}$ at final time $T = 0.5$.

Python code

```python
class PeriodicBoundaryX(SubDomain):
    def inside(self, x, on_boundary):
        return x[0] < (-1.0 + DOLFIN_EPS) and \
               x[0] > (-1.0 - DOLFIN_EPS) and \
               on_boundary

    def map(self, x, y):
        y[0] = x[0] - 2.0
        y[1] = x[1]

class PeriodicBoundaryY(SubDomain):
    def inside(self, x, on_boundary):
        return x[1] < (-1.0 + DOLFIN_EPS) and \
               x[1] > (-1.0 - DOLFIN_EPS) and \
               on_boundary

    def map(self, x, y):
        y[0] = x[0]
        y[1] = x[1] - 2.0
```

Figure 19.10: Implementation of periodic boundary conditions for the Taylor–Green vortex test problem.

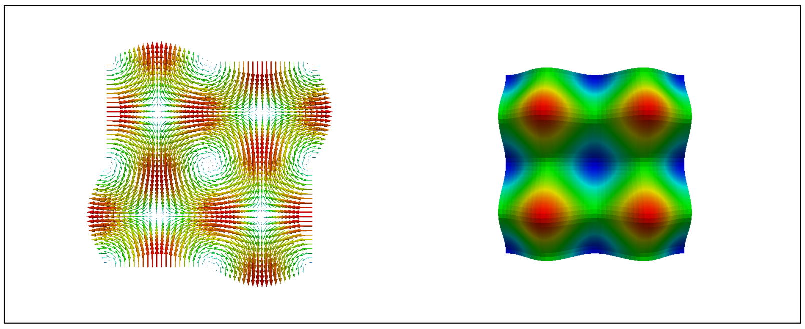


Figure 19.11: An illustration of the initial conditions for the Taylor–Green vortex test problem. On the left is the velocity field with vectors and on the right is the corresponding pressure field.

Results. Figure 19.12 shows the results for the Taylor–Green test problem. The smallest error is obtained with the IPCS solver. For this test problem, the G2 solver is overly dissipative and produces an error which is six orders of magnitude larger than that of the IPCS solver.

Flow past a cylinder (2D)

We next consider a test problem from Turek [1996], which is a two-dimensional cylinder submerged into a fluid and surrounded by solid walls as illustrated in Figure 19.13. The cylinder is slightly displaced from the center of the channel and the resulting flow is a vortex street forming behind the cylinder. No-slip boundary conditions are applied to the cylinder as well as the upper and lower walls of the channel. A zero Dirichlet boundary condition is imposed on the pressure at the outlet. The inflow velocity is a time-varying parabolic profile given by

$$u(0,y,t) = (4U_m y(H-y)\sin(\pi t/8)/H^4, 0), \quad t \leqslant 8, \tag{19.37}$$

where $U_m = 1.5$ and $H = 0.41$. The kinematic viscosity is $\nu = 1/1000$. As a functional of interest, we consider the pressure difference between the front and back of the cylinder at final time $T = 8$, that is,

$$\Delta p = p(0.45, 0.2, 8) - p(0.55, 0.2, 8). \tag{19.38}$$

A reference value -0.11144 for this functional was obtained using the IPCS solver on a mesh that was approximately of twice the size (in terms of the number of cells) as the finest mesh used in the test, with a time step of approximately half the size of the finest used time step.

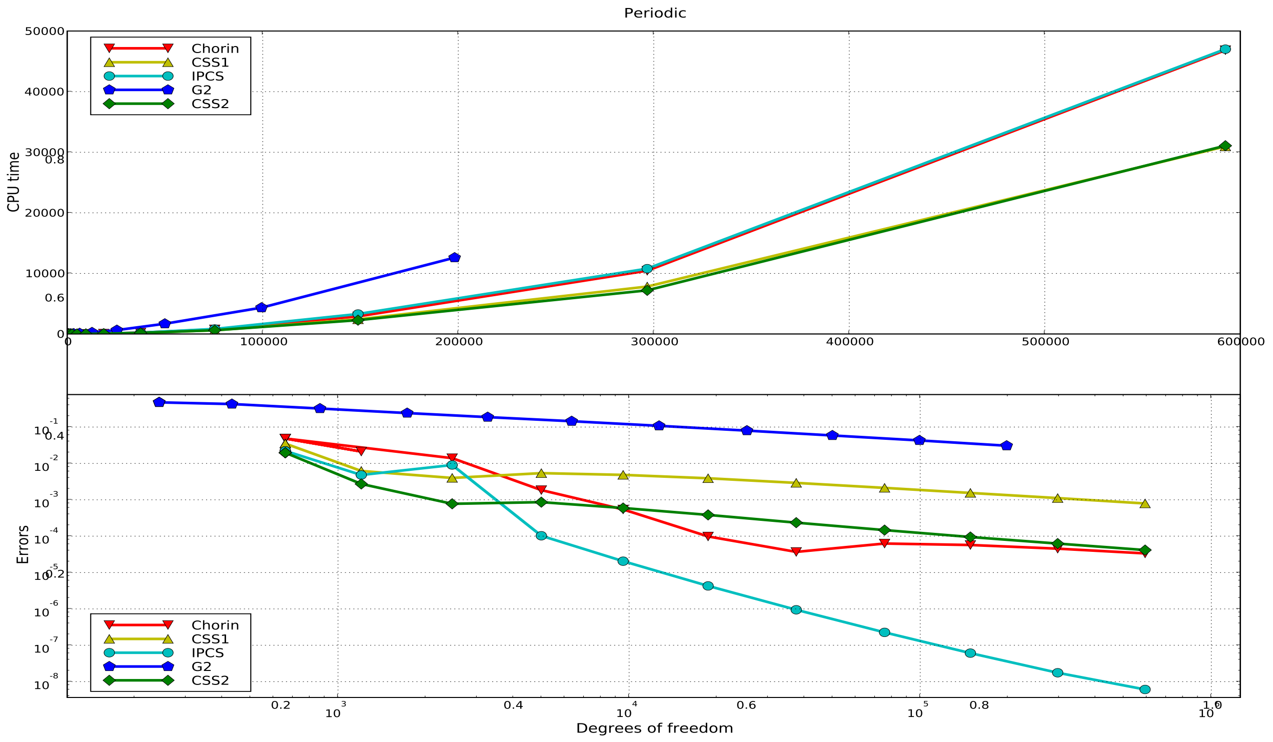


Figure 19.12: Results for the Taylor–Green vortex test problem.

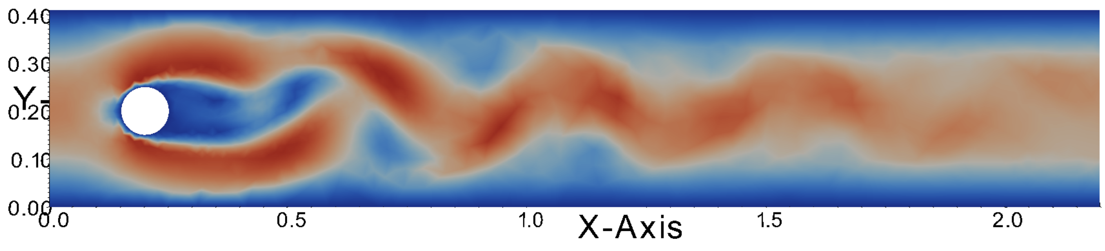


Figure 19.13: Illustration of the velocity field for the cylinder test problem at $t = 5$.

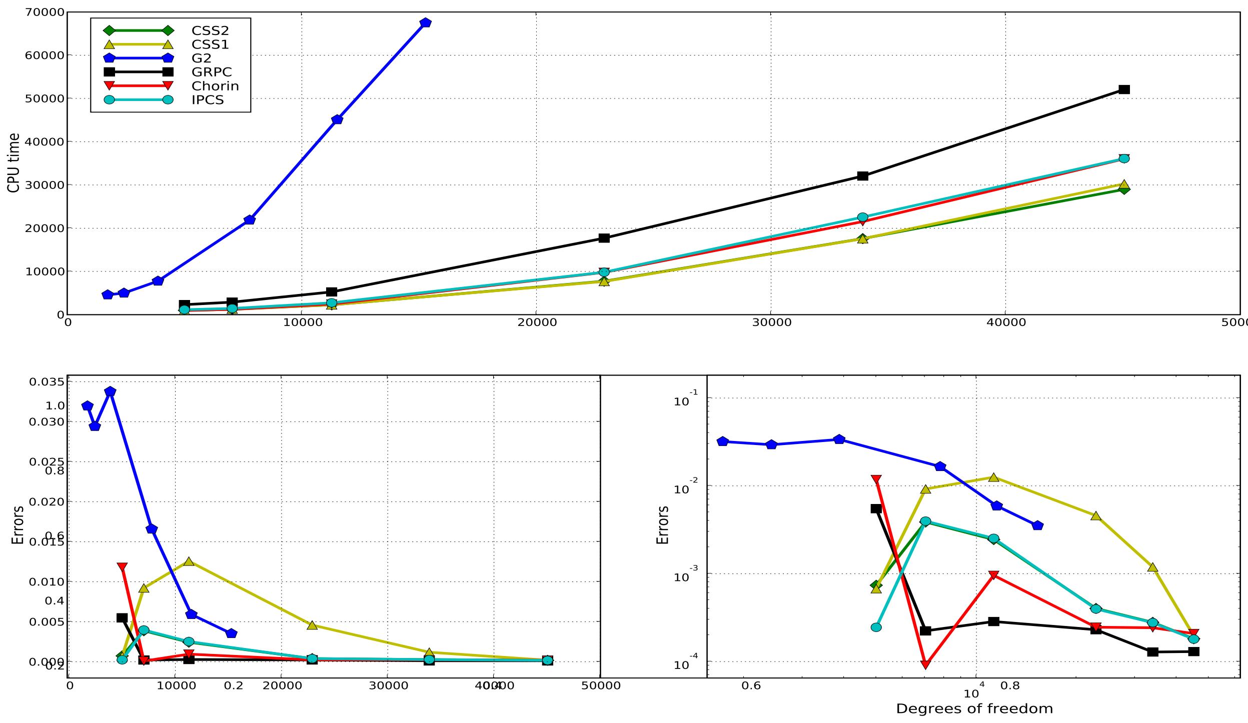


Figure 19.14: Results for the cylinder test problem.

Results. Figure 19.14 shows the results for the cylinder test problem. The smallest error is obtained with the GRPC solver closely followed by CSS_2 and IPCS. It is interesting to note that for this test problem, the CSS_2 solver is also the fastest.

Beltrami flow (3D)

We next consider a problem described in Ethier and Steinmann [1994] where an exact fully three-dimensional solution of the Navier–Stokes equations is derived. The flow is a so-called Beltrami flow, which has the property that the velocity and vorticity vectors are aligned. The domain is a cube with dimensions $[-1,1]^3$. The exact velocity is given by

$$\begin{aligned}
u(x,y,z,t) &= -a[e^{ax}\sin(ay+dz) + e^{az}\cos(ax+dy)]e^{-d^2t},\\
v(x,y,z,t) &= -a[e^{ay}\sin(az+dx) + e^{ax}\cos(ay+dz)]e^{-d^2t},\\
w(x,y,z,t) &= -a[e^{az}\sin(ax+dy) + e^{ay}\cos(az+dx)]e^{-d^2t},
\end{aligned} \tag{19.39}$$

and the exact pressure is given by

$$\begin{aligned}
p(x,y,z,t) = -a^2e^{-2d^2t}\left(e^{2ax}+e^{2ay}+e^{2az}\right)\times\\
\times\big(\sin(ax+dy)\cos(az+dx)e^{a(y+z)}+\\
\sin(ay+dz)\cos(ax+dy)e^{a(x+z)}+\\
\sin(az+dx)\cos(ay+dz)e^{a(x+y)}\big).
\end{aligned} \tag{19.40}$$

The solution is visualized in Figure 19.15. The constants a and d may be chosen arbitrarily, and have been set to $a = \pi/4$ and $d = \pi/2$ as in Ethier and Steinmann [1994]. The kinematic viscosity is $\nu = 1$. To

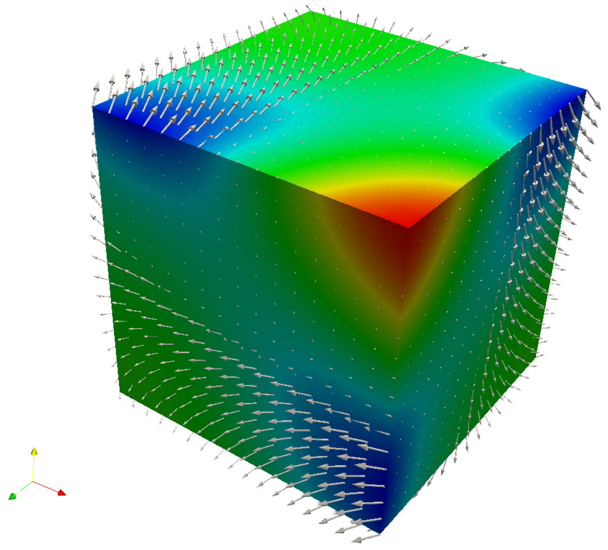


Figure 19.15: Solution of the Beltrami flow test problem.

measure the error, we compute the L^2 norm of the error in the velocity field at final time $T = 0.5$ divided by the L^2 norm of the exact solution as in Ethier and Steinmann [1994].

Results. Figure 19.16 shows the results for the Beltrami test problem. The smallest errors are obtained with the GRPC solver, while the largest errors are obtained with the CSS_1 solver.

Aneurysm (3D)

Finally, we consider an idealized geometry modeling an artery with a saccular aneurysm (see Chapter 23). The diameter of the artery is set to 4 mm and the length is set to 50 mm. The aneurysm is of medium size with a radius of 2.5 mm. Inserting the density and viscosity of blood and suitably scaling to dimensionless quantities, we obtain a kinematic viscosity of size $\nu = 3.5/(1.025 \cdot 10^3) \approx 3.4146 \cdot 10^{-6}$. The geometry and flow at the final time $T = 0.05$ (ms) is shown in Figure 19.17. We impose no-slip boundary conditions on the vessel walls. At the inlet, we set the velocity to $u(x, y, z, t) = \sin(30t)\,(1 - (y^2 + x^2)/r^2)$ where $r = 0.002$ (mm). At the outlet, we enforce a zero Dirichlet boundary condition for the pressure. As a functional of interest, we consider the x-component of the velocity at a point $(x, y, z) = (0.025, -0.006, 0)$ (mm) located inside the aneurysm at final time $T = 0.05$ (ms). A reference value -0.0355 (mm/ms) for this functional was obtained using the IPCS solver on a fine mesh.

Results. Figure 19.18 shows the results for the aneurysm test problem. Reasonable convergence is obtained for all solvers except the G2 solver which does not seem to converge towards the computed reference value.

19.5 *Summary of results*

To summarize the results for all solvers and test problems, we plot all timings and errors in a single scatter plot. The rationale behind the plot is to get an indication of which solver(s) is most accurate and efficient. Each data point in the plot is the result of solving one of the above test problems using one particular solver on one particular refinement level. To be able to compare different test problems (which vary in simulation time and size of error), the CPU time is scaled by the average CPU time for all solvers on each refinement level and the errors are scaled similarly. We also scale CPU times and errors by the number of degrees of freedom (total number of unknowns for both velocity and pressure). The resulting scatter plot is shown in Figure 19.19. An ideal solver (which is both fast and accurate) should be located in the lower left corner of this plot.

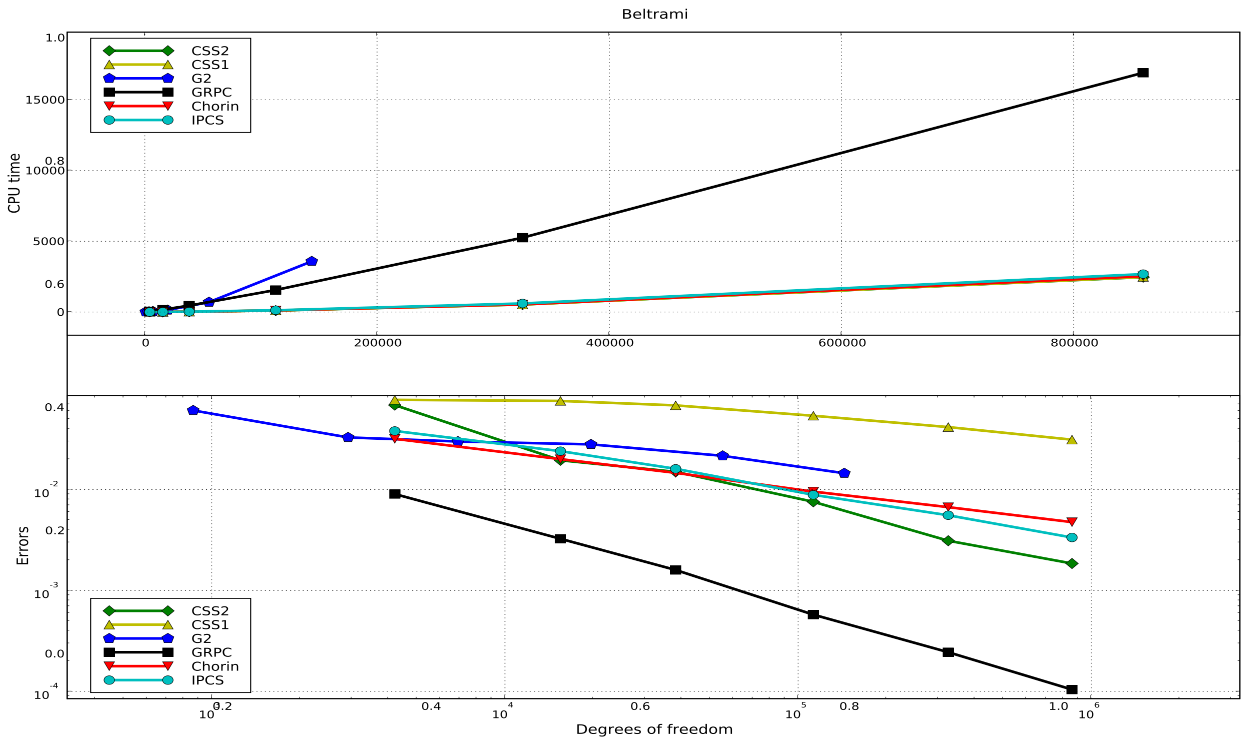
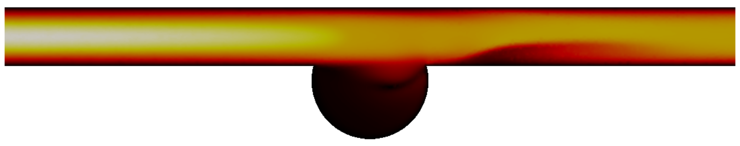


Figure 19.16: Results for the Beltrami flow test problem.

Figure 19.17: Velocity magnitude for the aneurysm test problem sliced at the center at final time $T = 0.05$ ms.



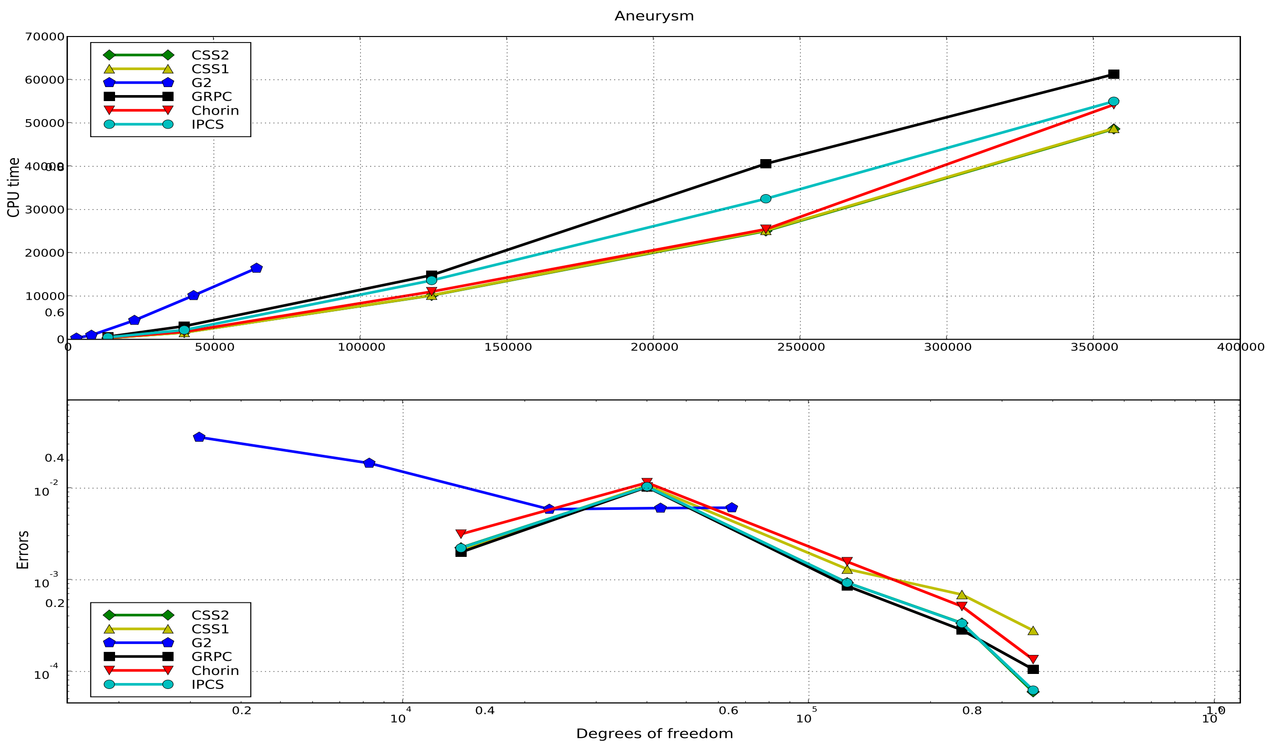
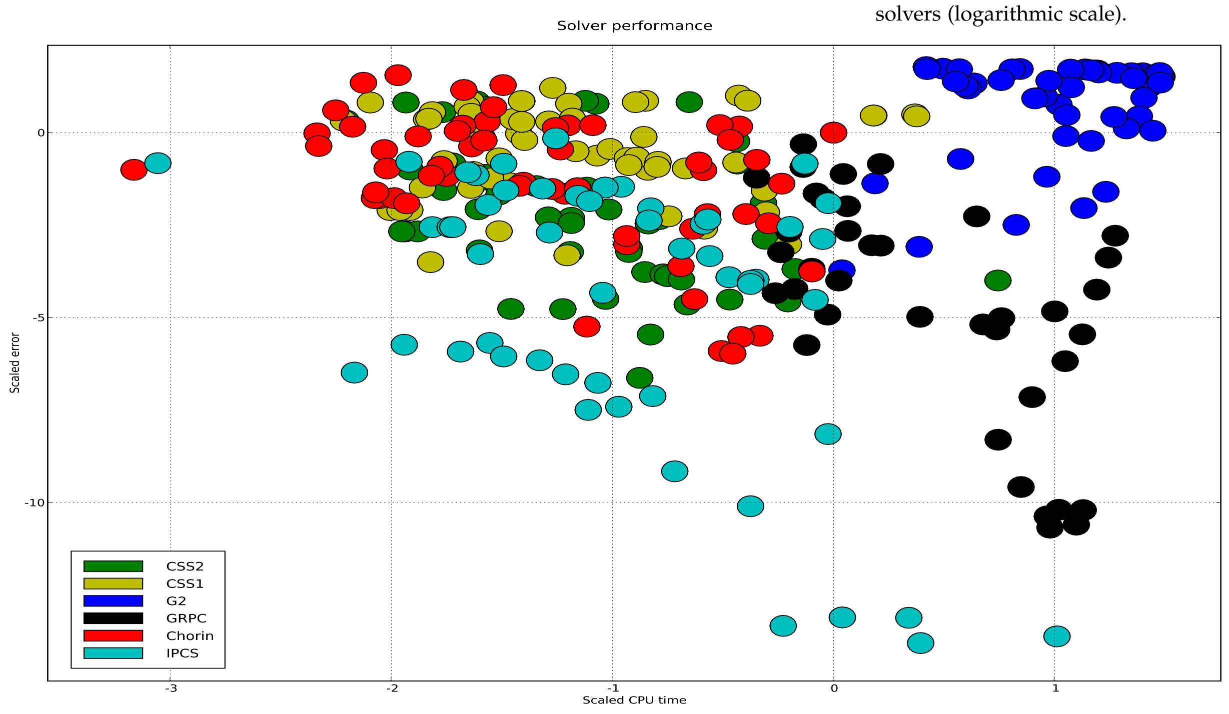


Figure 19.18: Results for the aneurysm test problem.

Figure 19.19: Scatter plot summarizing the results for all test problems and solvers (logarithmic scale).



As can be seen in Figure 19.19, the Chorin, CSS_1 and CSS_2 solvers have an average performance and are mostly clustered around the center of mass of the scatter plot. The G2 solver is mainly located in the upper right corner. The results for the IPCS solver are less clustered but it is the solver with most points located in the lower left corner. The GRPC solver is mostly located in the lower right corner of the scatter plot, indicating that it is accurate but expensive.

19.6 Discussion

Numerical boundary layers

As pointed out in Guermond et al. [2006], the fractional step solvers are usually plagued by an artificial boundary layer, because the boundary condition $\nabla p_h^n \cdot n|_{\partial\Omega} = 0$ is enforced on the pressure. This 'unphysical' Neumann boundary condition can create a numerical boundary layer simply because the velocity update $u_h^n = u^{n-1} - \Delta t \nabla p_h^n$ may lead to non-zero velocities in the tangential direction on no-slip walls (this follows since there is nothing preventing the pressure gradient from being non-zero in the tangential direction). However, in this work the velocity is being updated through a weak form where the no-slip boundary condition is strongly enforced. As such, the tangential velocity is set to zero and an artificial boundary layer is not observed in our simulations using the fractional step solvers Chorin and IPCS.

Time discretization

For the channel test problem, the convective term is zero and the discretization of the diffusive term is of particular importance. A formally second-order accurate in time Crank–Nicolson type scheme for the viscous term will in general improve the accuracy over the merely first-order explicit or fully implicit schemes. This is why the GRPC, IPCS and G2 solvers perform well on this problem. The channel test problem is the problem where G2 performs best relative to the other solvers, which could also be attributed to the fact that both stabilization terms in the momentum equation of G2 are zero for this flow.

19.7 Conclusions

From the scatter plot in Figure 19.19, we conclude that the IPCS solver is overall the most efficient and accurate method. Another advantage of the IPCS method is that it is easy to implement and does not require the iterative solution of a nonlinear system in each time step. The GRPC method (straightforward standard finite element Galerkin discretization) also obtains high accuracy, but does not deliver the same speed. It is possible that better tuning of the iterative solution of the saddle-point system would change this picture.