## Chapter 3
# A Gallery of finite element solvers

The goal of this chapter is to demonstrate how a range of important PDEs from science and engineering can be quickly solved with a few lines of FEniCS code. We start with the heat equation and continue with a nonlinear Poisson equation, the equations for linear elasticity, the Navier–Stokes equations, and finally look at how to solve systems of nonlinear advection–diffusion–reaction equations. These problems illustrate how to solve time-dependent problems, nonlinear problems, vector-valued problems, and systems of PDEs. For each problem, we derive the variational formulation and express the problem in Python in a way that closely resembles the mathematics.

## 3.1 The heat equation

As a first extension of the Poisson problem from the previous chapter, we consider the time-dependent heat equation, or the time-dependent diffusion equation. This is the natural extension of the Poisson equation describing the stationary distribution of heat in a body to a time-dependent problem.

We will see that by discretizing time into small time intervals and applying standard time-stepping methods, we can solve the heat equation by solving a sequence of variational problems, much like the one we encountered for the Poisson equation.

### 3.1.1 PDE problem

Our model problem for time-dependent PDEs reads

© The Author(s) 2016

H.P Langtangen and A. Logg, *Solving PDEs in Python*,
Simula SpringerBriefs on Computing 3, DOI 10.1007/978-3-319-52462-7_3

$$\frac{\partial u}{\partial t} = \nabla^2 u + f \quad \text{in } \Omega \times (0,T], \tag{3.1}$$

$$u = u_\mathrm{D} \quad \text{on } \partial\Omega \times (0,T], \tag{3.2}$$

$$u = u_0 \quad \text{at } t = 0\,. \tag{3.3}$$

Here, $u$ varies with space and time, e.g., $u = u(x,y,t)$ if the spatial domain $\Omega$ is two-dimensional. The source function $f$ and the boundary values $u_\mathrm{D}$ may also vary with space and time. The initial condition $u_0$ is a function of space only.

### 3.1.2 Variational formulation

A straightforward approach to solving time-dependent PDEs by the finite element method is to first discretize the time derivative by a finite difference approximation, which yields a sequence of stationary problems, and then turn each stationary problem into a variational formulation.

Let superscript $n$ denote a quantity at time $t_n$, where $n$ is an integer counting time levels. For example, $u^n$ means $u$ at time level $n$. A finite difference discretization in time first consists of sampling the PDE at some time level, say $t_{n+1}$:

$$\left(\frac{\partial u}{\partial t}\right)^{n+1} = \nabla^2 u^{n+1} + f^{n+1}\,. \tag{3.4}$$

The time-derivative can be approximated by a difference quotient. For simplicity and stability reasons, we choose a simple backward difference:

$$\left(\frac{\partial u}{\partial t}\right)^{n+1} \approx \frac{u^{n+1} - u^n}{\Delta t}, \tag{3.5}$$

where $\Delta t$ is the time discretization parameter. Inserting (3.5) in (3.4) yields

$$\frac{u^{n+1} - u^n}{\Delta t} = \nabla^2 u^{n+1} + f^{n+1}\,. \tag{3.6}$$

This is our time-discrete version of the heat equation (3.1), a so-called *backward Euler* or *implicit Euler* discretization.

We may reorder (3.6) so that the left-hand side contains the terms with the unknown $u^{n+1}$ and the right-hand side contains computed terms only. The result is a sequence of spatial (stationary) problems for $u^{n+1}$, assuming $u^n$ is known from the previous time step:

$$u^0 = u_0, \tag{3.7}$$

$$u^{n+1} - \Delta t \nabla^2 u^{n+1} = u^n + \Delta t f^{n+1}, \quad n = 0, 1, 2, \ldots \tag{3.8}$$

Given $u_0$, we can solve for $u^0$, $u^1$, $u^2$, and so on.

An alternative to (3.8), which can be convenient in implementations, is to collect all terms on one side of the equality sign:

$$u^{n+1} - \Delta t \nabla^2 u^{n+1} - u^n - \Delta t f^{n+1} = 0, \quad n = 0, 1, 2, \ldots \tag{3.9}$$

We use a finite element method to solve (3.7) and either of the equations (3.8) or (3.9). This requires turning the equations into weak forms. As usual, we multiply by a test function $v \in \hat{V}$ and integrate second-derivatives by parts. Introducing the symbol $u$ for $u^{n+1}$ (which is natural in the program), the resulting weak form arising from formulation (3.8) can be conveniently written in the standard notation:

$$a(u, v) = L_{n+1}(v),$$

where

$$a(u, v) = \int_\Omega (uv + \Delta t \nabla u \cdot \nabla v) \,\mathrm{d}x, \tag{3.10}$$

$$L_{n+1}(v) = \int_\Omega \left(u^n + \Delta t f^{n+1}\right) v \,\mathrm{d}x. \tag{3.11}$$

The alternative form (3.9) has an abstract formulation

$$F_{n+1}(u; v) = 0,$$

where

$$F_{n+1}(u; v) = \int_\Omega \left(uv + \Delta t \nabla u \cdot \nabla v - (u^n + \Delta t f^{n+1}) v\right) \mathrm{d}x. \tag{3.12}$$

In addition to the variational problem to be solved in each time step, we also need to approximate the initial condition (3.7). This equation can also be turned into a variational problem:

$$a_0(u, v) = L_0(v),$$

with

$$a_0(u,v) = \int_\Omega uv\,\mathrm{d}x, \tag{3.13}$$

$$L_0(v) = \int_\Omega u_{_0} v\,\mathrm{d}x\,. \tag{3.14}$$

When solving this variational problem, $u^0$ becomes the $L^2$ projection of the given initial value $u_{_0}$ into the finite element space. The alternative is to construct $u^0$ by just interpolating the initial value $u_{_0}$; that is, if $u^0 = \sum_{j=1}^N U_j^0 \phi_j$, we simply set $U_j = u_{_0}(x_j, y_j)$, where $(x_j, y_j)$ are the coordinates of node number $j$. We refer to these two strategies as computing the initial condition by either projection or interpolation. Both operations are easy to compute in FEniCS through a single statement, using either the `project` or `interpolate` function. The most common choice is `project`, which computes an approximation to $u_{_0}$, but in some applications where we want to verify the code by reproducing exact solutions, one must use `interpolate` (and we use such a test problem here!).

In summary, we thus need to solve the following sequence of variational problems to compute the finite element solution to the heat equation: find $u^0 \in V$ such that $a_0(u^0, v) = L_0(v)$ holds for all $v \in \hat{V}$, and then find $u^{n+1} \in V$ such that $a(u^{n+1}, v) = L_{n+1}(v)$ for all $v \in \hat{V}$, or alternatively, $F_{n+1}(u^{n+1}, v) = 0$ for all $v \in \hat{V}$, for $n = 0, 1, 2, \ldots$.

### 3.1.3 FEniCS implementation

Our program needs to implement the time-stepping manually, but can rely on FEniCS to easily compute $a_0$, $L_0$, $a$, and $L$ (or $F_{n+1}$), and solve the linear systems for the unknowns.

**Test problem 1: A known analytical solution.** Just as for the Poisson problem from the previous chapter, we construct a test problem that makes it easy to determine if the calculations are correct. Since we know that our first-order time-stepping scheme is exact for linear functions, we create a test problem which has a linear variation in time. We combine this with a quadratic variation in space. We thus take

$$u = 1 + x^2 + \alpha y^2 + \beta t, \tag{3.15}$$

which yields a function whose computed values at the nodes will be exact, regardless of the size of the elements and $\Delta t$, as long as the mesh is uniformly partitioned. By inserting (3.15) into the heat equation (3.1), we find that the right-hand side $f$ must be given by $f(x,y,t) = \beta - 2 - 2\alpha$. The boundary value is $u_{_\mathrm{D}}(x,y,t) = 1 + x^2 + \alpha y^2 + \beta t$ and the initial value is $u_{_0}(x,y) = 1 + x^2 + \alpha y^2$.

**FEniCS implementation.** A new programming issue is how to deal with functions that vary in space *and time*, such as the boundary condi-

tion $u_{\mathrm{D}}(x,y,t) = 1 + x^2 + \alpha y^2 + \beta t$. A natural solution is to use a FEniCS `Expression` with time $t$ as a parameter, in addition to the parameters $\alpha$ and $\beta$:

```
alpha = 3; beta = 1.2
u_D = Expression('1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t',
                 degree=2, alpha=alpha, beta=beta, t=0)
```

This `Expression` uses the components of `x` as independent variables, while `alpha`, `beta`, and `t` are parameters. The time `t` can later be updated by

```
u_D.t = t
```

The essential boundary conditions, along the entire boundary in this case, are implemented in the same way as we have previously implemented the boundary conditions for the Poisson problem:

```
def boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u_D, boundary)
```

We shall use the variable `u` for the unknown $u^{n+1}$ at the new time step and the variable `u_n` for $u^n$ at the previous time step. The initial value of `u_n` can be computed by either projection or interpolation of $u_0$. Since we set `t = 0` for the boundary value `u_D`, we can use `u_D` to specify the initial condition:

```
u_n = project(u_D, V)
# or
u_n = interpolate(u_D, V)
```

**Projecting versus interpolating the initial condition**

To actually recover the exact solution (3.15) to machine precision, it is important to compute the discrete initial condition by interpolating $u_0$. This ensures that the degrees of freedom are exact (to machine precision) at $t = 0$. Projection results in approximate values at the nodes.

We may either define $a$ or $L$ according to the formulas above, or we may just define $F$ and ask FEniCS to figure out which terms should go into the bilinear form $a$ and which should go into the linear form $L$. The latter is convenient, especially in more complicated problems, so we illustrate that construction of $a$ and $L$:

```
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(beta - 2 - 2*alpha)
```

```
F = u*v*dx + dt*dot(grad(u), grad(v))*dx - (u_n + dt*f)*v*dx
a, L = lhs(F), rhs(F)
```

Finally, we perform the time-stepping in a loop:

```
u = Function(V)
t = 0
for n in range(num_steps):

    # Update current time
    t += dt
    u_D.t = t

    # Solve variational problem
    solve(a == L, u, bc)

    # Update previous solution
    u_n.assign(u)
```

In the last step of the time-stepping loop, we assign the values of the variable `u` (the new computed solution) to the variable `u_n` containing the values at the previous time step. This must be done using the `assign` member function. If we instead try to do `u_n = u`, we will set the `u_n` variable to be the *same* variable as `u` which is not what we want. (We need two variables, one for the values at the previous time step and one for the values at the current time step.)

**Remember to update expression objects with the current time!**

Inside the time loop, observe that `u_D.t` must be updated before the `solve` statement to enforce computation of Dirichlet conditions at the current time step. A Dirichlet condition defined in terms of an `Exression` looks up and applies the value of a parameter such as `t` when it gets evaluated and applied to the linear system.

The time-stepping loop above does not contain any comparison of the numerical and the exact solutions, which we must include in order to verify the implementation. As for the Poisson equation in Section 2.3, we compute the difference between the array of nodal values for `u` and the array of nodal values for the interpolated exact solution. This may be done as follows:

```
u_e = interpolate(u_D, V)
error = np.abs(u_e.vector().array() - u.vector().array()).max()
print('t = %.2f: error = %.3g' % (t, error))
```

For the Poisson example, we used the function `compute_vertex_values` to extract the function values at the vertices. Here we illustrate an alternative method to extract the vertex values, by calling the function `vector`, which

returns the vector of degrees of freedom. For a $\mathsf{P}_1$ function space, this vector of degrees of freedom will be equal to the array of vertex values obtained by calling `compute_vertex_values`, albeit possibly in a different order.

The complete program for solving the heat equation goes as follows:

```
from fenics import *
import numpy as np

T = 2.0            # final time
num_steps = 10     # number of time steps
dt = T / num_steps # time step size
alpha = 3          # parameter alpha
beta = 1.2         # parameter beta

# Create mesh and define function space
nx = ny = 8
mesh = UnitSquareMesh(nx, ny)
V = FunctionSpace(mesh, 'P', 1)

# Define boundary condition
u_D = Expression('1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t',
                 degree=2, alpha=alpha, beta=beta, t=0)

def boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u_D, boundary)

# Define initial value
u_n = interpolate(u_D, V)
#u_n = project(u_D, V)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(beta - 2 - 2*alpha)

F = u*v*dx + dt*dot(grad(u), grad(v))*dx - (u_n + dt*f)*v*dx
a, L = lhs(F), rhs(F)

# Time-stepping
u = Function(V)
t = 0
for n in range(num_steps):

    # Update current time
    t += dt
    u_D.t = t

    # Compute solution
    solve(a == L, u, bc)

    # Plot solution
```

```
    plot(u)

    # Compute error at vertices
    u_e = interpolate(u_D, V)
    error = np.abs(u_e.vector().array() - u.vector().array()).max()
    print('t = %.2f: error = %.3g' % (t, error))

    # Update previous solution
    u_n.assign(u)

# Hold plot
interactive()
```

This example program can be found in the file `ft03_heat.py`.

**Test problem 2: Diffusion of a Gaussian function.** Let us now solve a more interesting test problem, namely the diffusion of a Gaussian hill. We take the initial value to be

$$u_0(x,y) = e^{-ax^2-ay^2}$$

for $a=5$ on the domain $[-2,2]\times[2,2]$. For this problem we will use homogeneous Dirichlet boundary conditions ($u_D=0$).

**FEniCS implementation.** Which are the required changes to our previous program? One major change is that the domain is no longer a unit square. The new domain can be created easily in FEniCS using `RectangleMesh`:

```
nx = ny = 30
mesh = RectangleMesh(Point(-2, -2), Point(2, 2), nx, ny)
```

Note that we have used a much higher resolution than before to better resolve the features of the solution. We also need to redefine the initial condition and the boundary condition. Both are easily changed by defining a new `Expression` and by setting $u=0$ on the boundary.

To be able to visualize the solution in an external program such as ParaView, we will save the solution to a file in VTK format in each time step. We do this by first creating a `File` with the suffix `.pvd`:

```
vtkfile = File('heat_gaussian/solution.pvd')
```

Inside the time loop, we may then append the solution values to this file:

```
vtkfile << (u, t)
```

This line is called in each time step, resulting in the creation of a new file with suffix `.vtu` containing all data for the time step (the mesh and the vertex values). The file `heat_gaussian/solution.pvd` will contain the time values and references to the `.vtu` file, which means that the `.pvd` file will be a single small file that points to a large number of `.vtu` files containing the actual data. Note that we choose to store the solution to a subdirectory named `heat_gaussian`. This is to avoid cluttering our source directory with

all the generated data files. One does not need to create the directory before running the program as it will be created automatically by FEniCS.

The complete program appears below.

```
from fenics import *
import time

T = 2.0            # final time
num_steps = 50     # number of time steps
dt = T / num_steps # time step size

# Create mesh and define function space
nx = ny = 30
mesh = RectangleMesh(Point(-2, -2), Point(2, 2), nx, ny)
V = FunctionSpace(mesh, 'P', 1)

# Define boundary condition
def boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, Constant(0), boundary)

# Define initial value
u_0 = Expression('exp(-a*pow(x[0], 2) - a*pow(x[1], 2))',
                 degree=2, a=5)
u_n = interpolate(u_0, V)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(0)

F = u*v*dx + dt*dot(grad(u), grad(v))*dx - (u_n + dt*f)*v*dx
a, L = lhs(F), rhs(F)

# Create VTK file for saving solution
vtkfile = File('heat_gaussian/solution.pvd')

# Time-stepping
u = Function(V)
t = 0
for n in range(num_steps):

    # Update current time
    t += dt

    # Compute solution
    solve(a == L, u, bc)

    # Save to file and plot solution
    vtkfile << (u, t)
    plot(u)

    # Update previous solution
```

```
    u_n.assign(u)

# Hold plot
interactive()
```

This example program can be found in the file `ft04_heat_gaussian.py`.

**Visualization in ParaView.** To visualize the diffusion of the Gaussian hill, start ParaView, choose **File–Open...**, open `heat_gaussian/solution.pvd`, and click **Apply** in the Properties pane. Click on the play button to display an animation of the solution. To save the animation to a file, click **File–Save Animation...** and save the file to a desired file format, for example AVI or Ogg/Theora. Once the animation has been saved to a file, you can play the animation offline using a player such as mplayer or VLC, or upload your animation to YouTube. Figure 3.1 shows a sequence of snapshots of the solution.

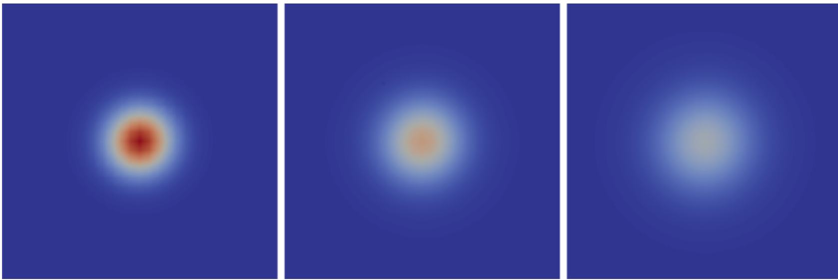

**Fig. 3.1** A sequence of snapshots of the solution of the Gaussian hill problem created with ParaView.

## 3.2 A nonlinear Poisson equation

We shall now address how to solve nonlinear PDEs. We will see that nonlinear problems can be solved just as easily as linear problems in FEniCS, by simply defining a nonlinear variational problem and calling the `solve` function. When doing so, we will encounter a subtle difference in how the variational problem is defined.

### 3.2.1 PDE problem

As a model problem for the solution of nonlinear PDEs, we take the following nonlinear Poisson equation:

$$-\nabla\cdot(q(u)\nabla u)=f, \tag{3.16}$$

in $\Omega$, with $u=u_{\mathrm{D}}$ on the boundary $\partial\Omega$. The coefficient $q=q(u)$ makes the equation nonlinear (unless $q(u)$ is constant in $u$).

### 3.2.2 Variational formulation

As usual, we multiply our PDE by a test function $v\in\hat{V}$, integrate over the domain, and integrate the second-order derivatives by parts. The boundary integral arising from integration by parts vanishes wherever we employ Dirichlet conditions. The resulting variational formulation of our model problem becomes: find $u\in V$ such that

$$F(u;v)=0\quad\forall v\in\hat{V}, \tag{3.17}$$

where

$$F(u;v)=\int_{\Omega}(q(u)\nabla u\cdot\nabla v-fv)\,\mathrm{d}x, \tag{3.18}$$

and

$$V=\{v\in H^1(\Omega): v=u_{\mathrm{D}} \text{ on } \partial\Omega\},$$
$$\hat{V}=\{v\in H^1(\Omega): v=0 \text{ on } \partial\Omega\}.$$

The discrete problem arises as usual by restricting $V$ and $\hat{V}$ to a pair of discrete spaces. As before, we omit any subscript on the discrete spaces and discrete solution. The discrete nonlinear problem is then written as: find $u\in V$ such that

$$F(u;v)=0\quad\forall v\in\hat{V}, \tag{3.19}$$

with $u=\sum_{j=1}^{N}U_j\phi_j$. Since $F$ is nonlinear in $u$, the variational statement gives rise to a system of nonlinear algebraic equations in the unknowns $U_1,\ldots,U_N$.

### 3.2.3 FEniCS implementation

**Test problem.** To solve a test problem, we need to choose the right-hand side $f$, the coefficient $q(u)$ and the boundary value $u_{\mathrm{D}}$. Previously, we have worked with manufactured solutions that can be reproduced without approximation errors. This is more difficult in nonlinear problems, and the algebra

is more tedious. However, we may utilize SymPy for symbolic computing and integrate such computations in the FEniCS solver. This allows us to easily experiment with different manufactured solutions. The forthcoming code with SymPy requires some basic familiarity with this package. In particular, we will use the SymPy functions `diff` for symbolic differentiation and `ccode` for C/C++ code generation.

We take $q(u) = 1 + u^2$ and define a two-dimensional manufactured solution that is linear in $x$ and $y$:

```
# Warning: from fenics import * will import both ‘sym‘ and
# ‘q‘ from FEniCS. We therefore import FEniCS first and then
# overwrite these objects.
from fenics import *

def q(u):
    "Return nonlinear coefficient"
    return 1 + u**2

# Use SymPy to compute f from the manufactured solution u
import sympy as sym
x, y = sym.symbols('x[0], x[1]')
u = 1 + x + 2*y
f = - sym.diff(q(u)*sym.diff(u, x), x) - sym.diff(q(u)*sym.diff(u, y), y)
f = sym.simplify(f)
u_code = sym.printing.ccode(u)
f_code = sym.printing.ccode(f)
print('u =', u_code)
print('f =', f_code)
```

**Define symbolic coordinates as required in `Expression` objects**

Note that we would normally write `x, y = sym.symbols('x, y')`, but if we want the resulting expressions to have valid syntax for FEniCS `Expression` objects, we must use `x[0]` and `x[1]`. This is easily accomplished with `sympy` by defining the names of `x` and `y` as `x[0]` and `x[1]`: `x, y = sym.symbols('x[0], x[1]')`.

Turning the expressions for `u` and `f` into C or C++ syntax for FEniCS `Expression` objects needs two steps. First, we ask for the C code of the expressions:

```
u_code = sym.printing.ccode(u)
f_code = sym.printing.ccode(f)
```

In some cases, one will need to edit the result to match the required syntax of `Expression` objects, but not in this case. (The primary example is that `M_PI` for $\pi$ in C/C++ must be replaced by `pi` for `Expression` objects.) In the present case, the output of `u_code` and `f_code` is

```
x[0] + 2*x[1] + 1
-10*x[0] - 20*x[1] - 10
```

After having defined the mesh, the function space, and the boundary, we define the boundary value `u_D` as

```
u_D = Expression(u_code, degree=1)
```

Similarly, we define the right-hand side function as

```
f = Expression(f_code, degree=1)
```

**Name clash between FEniCS and program variables**

In a program like the one above, strange errors may occur due to name clashes. If you define `sym` and `q` prior to doing `from fenics import *`, the latter statement will also import variables with the names `sym` and `q`, overwriting the objects you have previously defined! This may lead to strange errors. The safest solution is to do `import fenics` instead of `from fenics import *` and then prefix all FEniCS object names by `fenics`. The next best solution is to do `from fenics import *` first and then define your own variables that overwrite those imported from `fenics`. This is acceptable if we do not need `sym` and `q` from `fenics`.

**FEniCS implementation.** A solver for the nonlinear Poisson equation is as easy to implement as a solver for the linear Poisson equation. All we need to do is to state the formula for $F$ and call `solve(F == 0, u, bc)` instead of `solve(a == L, u, bc)` as we did in the linear case. Here is a minimalistic code:

```
from fenics import *

def q(u):
    return 1 + u**2

mesh = UnitSquareMesh(8, 8)
V = FunctionSpace(mesh, 'P', 1)
u_D = Expression(u_code, degree=1)

def boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u_D, boundary)

u = Function(V)
v = TestFunction(V)
f = Expression(f_code, degree=1)
F = q(u)*dot(grad(u), grad(v))*dx - f*v*dx
```

```
solve(F == 0, u, bc)
```

A complete version of this example program can be found in the file `ft05_poisson_nonlinear.py`.

The major difference from a linear problem is that the unknown function `u` in the variational form in the nonlinear case must be defined as a `Function`, not as a `TrialFunction`. In some sense this is a simplification from the linear case where we must define `u` first as a `TrialFunction` and then as a `Function`.

The `solve` function takes the nonlinear equations, derives symbolically the Jacobian matrix, and runs a Newton method to compute the solution.

When we run the code, FEniCS reports on the progress of the Newton iterations. With $2 \cdot (8 \times 8)$ cells, we reach convergence in eight iterations with a tolerance of $10^{-9}$, and the error in the numerical solution is about $10^{-16}$. These results bring evidence for a correct implementation. Thinking in terms of finite differences on a uniform mesh, $\mathsf{P}_1$ elements mimic standard second-order differences, which compute the derivative of a linear or quadratic function exactly. Here, $\nabla u$ is a constant vector, but then multiplied by $(1+u^2)$, which is a second-order polynomial in $x$ and $y$, which the divergence "difference operator" should compute exactly. We can therefore, even with $\mathcal{P}_1$ elements, expect the manufactured $u$ to be reproduced by the numerical method. With a nonlinearity like $1+u^4$, this will not be the case, and we would need to verify convergence rates instead.

The current example shows how easy it is to solve a nonlinear problem in FEniCS. However, experts on the numerical solution of nonlinear PDEs know very well that automated procedures may fail for nonlinear problems, and that it is often necessary to have much better manual control of the solution process than what we have in the current case. We return to this problem in [23] and show how we can implement taylored solution algorithms for nonlinear equations and also how we can steer the parameters in the automated Newton method used above.

## 3.3 The equations of linear elasticity

Analysis of structures is one of the major activities of modern engineering, which likely makes the PDE modeling the deformation of elastic bodies the most popular PDE in the world. It takes just one page of code to solve the equations of 2D or 3D elasticity in FEniCS, and the details follow below.

### 3.3.1 PDE problem

The equations governing small elastic deformations of a body $\Omega$ can be written as

$$-\nabla \cdot \sigma = f \text{ in } \Omega, \tag{3.20}$$

$$\sigma = \lambda \operatorname{tr}(\varepsilon) I + 2\mu\varepsilon, \tag{3.21}$$

$$\varepsilon = \frac{1}{2}\left(\nabla u + (\nabla u)^{\top}\right), \tag{3.22}$$

where $\sigma$ is the stress tensor, $f$ is the body force per unit volume, $\lambda$ and $\mu$ are Lamé's elasticity parameters for the material in $\Omega$, $I$ is the identity tensor, tr is the trace operator on a tensor, $\varepsilon$ is the symmetric strain-rate tensor (symmetric gradient), and $u$ is the displacement vector field. We have here assumed isotropic elastic conditions.

We combine (3.21) and (3.22) to obtain

$$\sigma = \lambda(\nabla \cdot u) I + \mu(\nabla u + (\nabla u)^{\top}). \tag{3.23}$$

Note that (3.20)–(3.22) can easily be transformed to a single vector PDE for $u$, which is the governing PDE for the unknown $u$ (Navier's equation). In the derivation of the variational formulation, however, it is convenient to keep the equations split as above.

### 3.3.2 Variational formulation

The variational formulation of (3.20)–(3.22) consists of forming the inner product of (3.20) and a *vector* test function $v \in \hat{V}$, where $\hat{V}$ is a vector-valued test function space, and integrating over the domain $\Omega$:

$$-\int_{\Omega} (\nabla \cdot \sigma) \cdot v \,\mathrm{d}x = \int_{\Omega} f \cdot v \,\mathrm{d}x.$$

Since $\nabla \cdot \sigma$ contains second-order derivatives of the primary unknown $u$, we integrate this term by parts:

$$-\int_{\Omega} (\nabla \cdot \sigma) \cdot v \,\mathrm{d}x = \int_{\Omega} \sigma : \nabla v \,\mathrm{d}x - \int_{\partial\Omega} (\sigma \cdot n) \cdot v \,\mathrm{d}s,$$

where the colon operator is the inner product between tensors (summed pairwise product of all elements), and $n$ is the outward unit normal at the boundary. The quantity $\sigma \cdot n$ is known as the *traction* or stress vector at the boundary, and is often prescribed as a boundary condition. We here assume that it is prescribed on a part $\partial\Omega_T$ of the boundary as $\sigma \cdot n = T$. On the remaining

part of the boundary, we assume that the value of the displacement is given as a Dirichlet condition. We thus obtain

$$\int_{\Omega} \sigma : \nabla v \,\mathrm{d}x = \int_{\Omega} f \cdot v \,\mathrm{d}x + \int_{\partial\Omega_T} T \cdot v \,\mathrm{d}s\,.$$

Inserting the expression (3.23) for $\sigma$ gives the variational form with $u$ as unknown. Note that the boundary integral on the remaining part $\partial\Omega \setminus \partial\Omega_T$ vanishes due to the Dirichlet condition.

We can now summarize the variational formulation as: find $u \in V$ such that

$$a(u,v) = L(v) \quad \forall v \in \hat{V}, \tag{3.24}$$

where

$$a(u,v) = \int_{\Omega} \sigma(u) : \nabla v \,\mathrm{d}x, \tag{3.25}$$

$$\sigma(u) = \lambda(\nabla \cdot u)I + \mu(\nabla u + (\nabla u)^{\top}), \tag{3.26}$$

$$L(v) = \int_{\Omega} f \cdot v \,\mathrm{d}x + \int_{\partial\Omega_T} T \cdot v \,\mathrm{d}s\,. \tag{3.27}$$

One can show that the inner product of a symmetric tensor $A$ and an anti-symmetric tensor $B$ vanishes. If we express $\nabla v$ as a sum of its symmetric and anti-symmetric parts, only the symmetric part will survive in the product $\sigma : \nabla v$ since $\sigma$ is a symmetric tensor. Thus replacing $\nabla u$ by the symmetric gradient $\epsilon(u)$ gives rise to the slightly different variational form

$$a(u,v) = \int_{\Omega} \sigma(u) : \varepsilon(v) \,\mathrm{d}x, \tag{3.28}$$

where $\varepsilon(v)$ is the symmetric part of $\nabla v$:

$$\varepsilon(v) = \frac{1}{2}\left(\nabla v + (\nabla v)^{\top}\right).$$

The formulation (3.28) is what naturally arises from minimization of elastic potential energy and is a more popular formulation than (3.25).

### 3.3.3 FEniCS implementation

**Test problem.** As a test example, we will model a clamped beam deformed under its own weight in 3D. This can be modeled by setting the right-hand side body force per unit volume to $f = (0,0,-\varrho g)$ with $\varrho$ the density of the beam and $g$ the acceleration of gravity. The beam is box-shaped with length

$L$ and has a square cross section of width $W$. We set $u = u_{\mathrm{D}} = (0,0,0)$ at the clamped end, $x = 0$. The rest of the boundary is traction free; that is, we set $T = 0$.

**FEniCS implementation.** We first list the code and then comment upon the new constructions compared to the previous examples we have seen.

```
from fenics import *

# Scaled variables
L = 1; W = 0.2
mu = 1
rho = 1
delta = W/L
gamma = 0.4*delta**2
beta = 1.25
lambda_ = beta
g = gamma

# Create mesh and define function space
mesh = BoxMesh(Point(0, 0, 0), Point(L, W, W), 10, 3, 3)
V = VectorFunctionSpace(mesh, 'P', 1)

# Define boundary condition
tol = 1E-14

def clamped_boundary(x, on_boundary):
    return on_boundary and x[0] < tol

bc = DirichletBC(V, Constant((0, 0, 0)), clamped_boundary)

# Define strain and stress

def epsilon(u):
    return 0.5*(nabla_grad(u) + nabla_grad(u).T)
    #return sym(nabla_grad(u))

def sigma(u):
    return lambda_*nabla_div(u)*Identity(d) + 2*mu*epsilon(u)

# Define variational problem
u = TrialFunction(V)
d = u.geometric_dimension()  # space dimension
v = TestFunction(V)
f = Constant((0, 0, -rho*g))
T = Constant((0, 0, 0))
a = inner(sigma(u), epsilon(v))*dx
L = dot(f, v)*dx + dot(T, v)*ds

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution
```

```
plot(u, title='Displacement', mode='displacement')

# Plot stress
s = sigma(u) - (1./3)*tr(sigma(u))*Identity(d)  # deviatoric stress
von_Mises = sqrt(3./2*inner(s, s))
V = FunctionSpace(mesh, 'P', 1)
von_Mises = project(von_Mises, V)
plot(von_Mises, title='Stress intensity')

# Compute magnitude of displacement
u_magnitude = sqrt(dot(u, u))
u_magnitude = project(u_magnitude, V)
plot(u_magnitude, 'Displacement magnitude')
print('min/max u:',
      u_magnitude.vector().array().min(),
      u_magnitude.vector().array().max())
```

This example program can be found in the file `ft06_elasticity.py`.

**Vector function spaces.** The primary unknown is now a vector field $u$ and not a scalar field, so we need to work with a vector function space:

```
V = VectorFunctionSpace(mesh, 'P', 1)
```

With `u = Function(V)` we get `u` as a vector-valued finite element function with three components for this 3D problem.

**Constant vectors.** For the boundary condition $u = (0,0,0)$, we must set a vector value to zero, not just a scalar. Such a vector constant is specified as `Constant((0, 0, 0))` in FEniCS. The corresponding 2D code would use `Constant((0, 0))`. Later in the code, we also need `f` as a vector and specify it as `Constant((0, 0, rho*g))`.

**`nabla_grad`.** The gradient and divergence operators now have a prefix `nabla_`. This is strictly not necessary in the present problem, but recommended in general for vector PDEs arising from continuum mechanics, if you interpret $\nabla$ as a vector in the PDE notation; see the box about `nabla_grad` in Section 3.4.2.

**Stress computation.** As soon as the displacement `u` is computed, we can compute various stress measures. We will compute the von Mises stress defined as $\sigma_M = \sqrt{\frac{3}{2}s:s}$ where $s$ is the deviatoric stress tensor

$$s = \sigma - \frac{1}{3}\mathrm{tr}\,(\sigma)\,I\,.$$

There is a one-to-one mapping between these formulas and the FEniCS code:

```
s = sigma(u) - (1./3)*tr(sigma(u))*Identity(d)
von_Mises = sqrt(3./2*inner(s, s))
```

The `von_Mises` variable is now an expression that must be projected to a finite element space before we can visualize it:

```
V = FunctionSpace(mesh, 'P', 1)
von_Mises = project(von_Mises, V)
plot(von_Mises, title='Stress intensity')
```

**Scaling.** It is often advantageous to scale a problem as it reduces the need for setting physical parameters, and one obtains dimensionsless numbers that reflect the competition of parameters and physical effects. We develop the code for the original model with dimensions, and run the scaled problem by tweaking parameters appropriately. Scaling reduces the number of active parameters from 6 to 2 for the present application.

In Navier's equation for $u$, arising from inserting (3.21) and (3.22) into (3.20),

$$-(\lambda+\mu)\nabla(\nabla\cdot u) - \mu\nabla^2 u = f,$$

we insert coordinates made dimensionless by $L$, and $\bar{u} = u/U$, which results in the dimensionless governing equation

$$-\beta\bar{\nabla}(\bar{\nabla}\cdot\bar{u}) - \bar{\nabla}^2\bar{u} = \bar{f}, \quad \bar{f} = (0,0,\gamma),$$

where $\beta = 1 + \lambda/\mu$ is a dimensionless elasticity parameter and where

$$\gamma = \frac{\varrho g L^2}{\mu U}$$

is a dimensionless variable reflecting the ratio of the load $\varrho g$ and the shear stress term $\mu\nabla^2 u \sim \mu U/L^2$ in the PDE.

One option for the scaling is to chose $U$ such that $\gamma$ is of unit size ($U = \varrho g L^2/\mu$). However, in elasticity, this leads to displacements of the size of the geometry, which makes plots look very strange. We therefore want the characteristic displacement to be a small fraction of the characteristic length of the geometry. This can be achieved by choosing $U$ equal to the maximum deflection of a clamped beam, for which there actually exists a formula: $U = \frac{3}{2}\varrho g L^2\delta^2/E$, where $\delta = L/W$ is a parameter reflecting how slender the beam is, and $E$ is the modulus of elasticity. Thus, the dimensionless parameter $\delta$ is very important in the problem (as expected, since $\delta \gg 1$ is what gives beam theory!). Taking $E$ to be of the same order as $\mu$, which is the case for many materials, we realize that $\gamma \sim \delta^{-2}$ is an appropriate choice. Experimenting with the code to find a displacement that "looks right" in plots of the deformed geometry, points to $\gamma = 0.4\delta^{-2}$ as our final choice of $\gamma$.

The simulation code implements the problem with dimensions and physical parameters $\lambda$, $\mu$, $\varrho$, $g$, $L$, and $W$. However, we can easily reuse this code for a scaled problem: just set $\mu = \varrho = L = 1$, $W$ as $W/L$ ($\delta^{-1}$), $g = \gamma$, and $\lambda = \beta$.

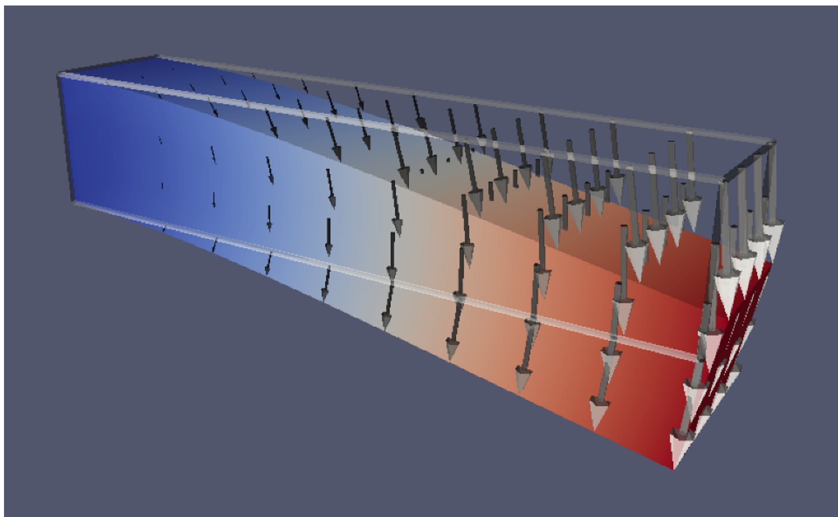

**Fig. 3.2** Plot of gravity-induced deflection in a clamped beam for the elasticity problem.

## 3.4 The Navier–Stokes equations

For the next example, we will solve the incompressible Navier–Stokes equations. This problem combines many of the challenges from our previously studied problems: time-dependence, nonlinearity, and vector-valued variables. We shall touch on a number of FEniCS topics, many of them quite advanced. But you will see that even a relatively complex algorithm such as a second-order splitting method for the incompressible Navier–Stokes equations, can be implemented with relative ease in FEniCS.

### 3.4.1 PDE problem

The incompressible Navier–Stokes equations form a system of equations for the velocity $u$ and pressure $p$ in an incompressible fluid:

$$\varrho\left(\frac{\partial u}{\partial t} + u\cdot\nabla u\right) = \nabla\cdot\sigma(u,p) + f, \tag{3.29}$$

$$\nabla\cdot u = 0. \tag{3.30}$$

The right-hand side $f$ is a given force per unit volume and just as for the equations of linear elasticity, $\sigma(u,p)$ denotes the stress tensor, which for a Newtonian fluid is given by

$$\sigma(u,p) = 2\mu\epsilon(u) - pI, \tag{3.31}$$

where $\epsilon(u)$ is the strain-rate tensor

$$\epsilon(u) = \frac{1}{2}\left(\nabla u + (\nabla u)^T\right).$$

The parameter $\mu$ is the dynamic viscosity. Note that the momentum equation (3.29) is very similar to the elasticity equation (3.20). The difference is in the two additional terms $\varrho(\partial u/\partial t + u\cdot\nabla u)$ and the different expression for the stress tensor. The two extra terms express the acceleration balanced by the force $F = \nabla\cdot\sigma + f$ per unit volume in Newton's second law of motion.

### 3.4.2 Variational formulation

The Navier–Stokes equations are different from the time-dependent heat equation in that we need to solve a system of equations and this system is of a special type. If we apply the same technique as for the heat equation; that is, replacing the time derivative with a simple difference quotient, we obtain a nonlinear system of equations. This in itself is not a problem for FEniCS as we saw in Section 3.2, but the system has a so-called *saddle point structure* and requires special techniques (special preconditioners and iterative methods) to be solved efficiently.

Instead, we will apply a simpler and often very efficient approach, known as a *splitting method*. The idea is to consider the two equations (3.29) and (3.30) separately. There exist many splitting strategies for the incompressible Navier–Stokes equations. One of the oldest is the method proposed by Chorin [6] and Temam [31], often referred to as *Chorin's method*. We will use a modified version of Chorin's method, the so-called incremental pressure correction scheme (IPCS) due to [13] which gives improved accuracy compared to the original scheme at little extra cost.

The IPCS scheme involves three steps. First, we compute a *tentative velocity* $u^\star$ by advancing the momentum equation (3.29) by a midpoint finite difference scheme in time, but using the pressure $p^n$ from the previous time interval. We will also linearize the nonlinear convective term by using the known velocity $u^n$ from the previous time step: $u^n\cdot\nabla u^n$. The variational problem for this first step is

$$\begin{aligned}\langle\varrho(u^\star - u^n)/\Delta t, v\rangle + \langle\varrho u^n\cdot\nabla u^n, v\rangle + \langle\sigma(u^{n+\frac{1}{2}}, p^n), \epsilon(v)\rangle \\ + \langle p^n n, v\rangle_{\partial\Omega} - \langle\mu\nabla u^{n+\frac{1}{2}}\cdot n, v\rangle_{\partial\Omega} = \langle f^{n+1}, v\rangle.\end{aligned} \tag{3.32}$$

This notation, suitable for problems with many terms in the variational formulations, requires some explanation. First, we use the short-hand notation

$$\langle v, w \rangle = \int_\Omega vw \, \mathrm{d}x, \quad \langle v, w \rangle_{\partial\Omega} = \int_{\partial\Omega} vw \, \mathrm{d}s.$$

This allows us to express the variational problem in a more compact way. Second, we use the notation $u^{n+\frac{1}{2}}$. This notation refers to the value of $u$ at the midpoint of the interval, usually approximated by an arithmetic mean:

$$u^{n+\frac{1}{2}} \approx (u^n + u^{n+1})/2.$$

Third, we notice that the variational problem (3.32) arises from the integration by parts of the term $\langle -\nabla\cdot\sigma, v\rangle$. Just as for the elasticity problem in Section 3.3, we obtain

$$\langle -\nabla\cdot\sigma, v\rangle = \langle \sigma, \epsilon(v)\rangle - \langle T, v\rangle_{\partial\Omega},$$

where $T = \sigma\cdot n$ is the boundary traction. If we solve a problem with a free boundary, we can take $T = 0$ on the boundary. However, if we compute the flow through a channel or a pipe and want to model flow that continues into an "imaginary channel" at the outflow, we need to treat this term with some care. The assumption we then make is that the derivative of the velocity in the direction of the channel is zero at the outflow, corresponding to a flow that is "fully developed" or doesn't change significantly downstream of the outflow. Doing so, the remaining boundary term at the outflow becomes $pn - \mu\nabla u\cdot n$, which is the term appearing in the variational problem (3.32). Note that this argument and the implementation depends on the exact definition of $\nabla u$, as either the matrix with components $\partial u_i/\partial x_j$ or $\partial u_j/\partial x_i$. We here choose the latter, $\partial u_j/\partial x_i$, which means that we must use the FEniCS operator `nabla_grad` for the implementation. If we use the `grad` operator and the definition $\partial u_i/\partial x_j$, we must instead keep the terms $pn - \mu(\nabla u)^\top\cdot n$!

**`grad(u)` vs. `nabla_grad(u)`**

For scalar functions, $\nabla u$ has a clear meaning as the vector

$$\nabla u = \left(\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}, \frac{\partial u}{\partial z}\right).$$

However, if $u$ is vector-valued, the meaning is less clear. Some sources define $\nabla u$ as the matrix with elements $\partial u_j/\partial x_i$, while other sources prefer $\partial u_i/\partial x_j$. In FEniCS, `grad(u)` is defined as the matrix with elements $\partial u_i/\partial x_j$, which is the natural definition of $\nabla u$ if we think of this as the *gradient* or *derivative* of $u$. This way, the matrix $\nabla u$ can be applied to a differential $\mathrm{d}x$ to give an increment $\mathrm{d}u = \nabla u \, \mathrm{d}x$. Since the alterna-

tive interpretation of $\nabla u$ as the matrix with elements $\partial u_j/\partial x_i$ is very common, in particular in continuum mechanics, FEniCS provides the operator `nabla_grad` for this purpose. For the Navier–Stokes equations, it is important to consider the term $u \cdot \nabla u$ which should be interpreted as the vector $w$ with elements $w_i = \sum_j \left( u_j \frac{\partial}{\partial x_j} \right) u_i = \sum_j u_j \frac{\partial u_i}{\partial x_j}$. This term can be implemented in FEniCS either as `grad(u)*u`, since this is expression becomes $\sum_j \partial u_i/\partial x_j u_j$, or as `dot(u, nabla_grad(u))` since this expression becomes $\sum_i u_i \partial u_j/\partial x_i$. We will use the notation `dot(u, nabla_grad(u))` below since it corresponds more closely to the standard notation $u \cdot \nabla u$.

To be more precise, there are three different notations used for PDEs involving gradient, divergence, and curl operators. One employs $\mathrm{grad}\, u$, $\mathrm{div}\, u$, and $\mathrm{curl}\, u$ operators. Another employs $\nabla u$ as a synonym for $\mathrm{grad}\, u$, $\nabla \cdot u$ means $\mathrm{div}\, u$, and $\nabla \times u$ is the name for $\mathrm{curl}\, u$. The third operates with $\nabla u$, $\nabla \cdot u$, and $\nabla \times u$ in which $\nabla$ is a *vector* and, e.g., $\nabla u$ is a dyadic expression: $(\nabla u)_{i,j} = \partial u_j/\partial x_i = (\mathrm{grad}\, u)^\top$. The latter notation, with $\nabla$ as a vector operator, is often handy when deriving equations in continuum mechanics, and if this interpretation of $\nabla$ is the foundation of your PDE, you must use `nabla_grad`, `nabla_div`, and `nabla_curl` in FEniCS code as these operators are compatible with dyadic computations. From the Navier–Stokes equations we can easily see what $\nabla$ means: if the convective term has the form $u \cdot \nabla u$, actually meaning $(u \cdot \nabla)u$, then $\nabla$ is a vector and the implementation becomes `dot(u, nabla_grad(u))` in FEniCS, but if we see $\nabla u \cdot u$ or $(\mathrm{grad} u) \cdot u$, the corresponding FEniCS expression is `dot(grad(u), u)`.

Similarly, the divergence of a tensor field like the stress tensor $\sigma$ can also be expressed in two different ways, as either `div(sigma)` or `nabla_div(sigma)`. The first case corresponds to the components $\partial \sigma_{ij}/\partial x_j$ and the second to $\partial \sigma_{ij}/\partial x_i$. In general, these expressions will be different but when the stress measure is symmetric, the expressions have the same value.

We now move on to the second step in our splitting scheme for the incompressible Navier–Stokes equations. In the first step, we computed the tentative velocity $u^\star$ based on the pressure from the previous time step. We may now use the computed tentative velocity to compute the new pressure $p^n$:

$$\langle \nabla p^{n+1}, \nabla q \rangle = \langle \nabla p^n, \nabla q \rangle - \Delta t^{-1} \langle \nabla \cdot u^\star, q \rangle. \tag{3.33}$$

Note here that $q$ is a scalar-valued test function from the pressure space, whereas the test function $v$ in (3.32) is a vector-valued test function from the velocity space.

One way to think about this step is to subtract the Navier–Stokes momentum equation (3.29) expressed in terms of the tentative velocity $u^\star$ and the

pressure $p^n$ from the momentum equation expressed in terms of the velocity $u^{n+1}$ and pressure $p^{n+1}$. This results in the equation

$$(u^{n+1} - u^{\star})/\Delta t + \nabla p^{n+1} - \nabla p^n = 0. \tag{3.34}$$

Taking the divergence and requiring that $\nabla \cdot u^{n+1} = 0$ by the Navier–Stokes continuity equation (3.30), we obtain the equation $-\nabla \cdot u^{\star}/\Delta t + \nabla^2 p^{n+1} - \nabla^2 p^n = 0$, which is a Poisson problem for the pressure $p^{n+1}$ resulting in the variational problem (3.33).

Finally, we compute the corrected velocity $u^{n+1}$ from the equation (3.34). Multiplying this equation by a test function $v$, we obtain

$$\langle u^{n+1}, v\rangle = \langle u^{\star}, v\rangle - \Delta t \langle \nabla (p^{n+1} - p^n), v\rangle. \tag{3.35}$$

In summary, we may thus solve the incompressible Navier–Stokes equations efficiently by solving a sequence of three linear variational problems in each time step.

### 3.4.3 FEniCS implementation

**Test problem 1: Channel flow.** As a first test problem, we compute the flow between two infinite plates, so-called channel or Poiseuille flow. As we shall see, this problem has a known analytical solution. Let $H$ be the distance between the plates and $L$ the length of the channel. There are no body forces.

We may scale the problem first to get rid of seemingly independent physical parameters. The physics of this problem is governed by viscous effects only, in the direction perpendicular to the flow, so a time scale should be based on diffusion accross the channel: $t_c = H^2/\nu$. We let $U$, some characteristic inflow velocity, be the velocity scale and $H$ the spatial scale. The pressure scale is taken as the characteristic shear stress, $\mu U/H$, since this is a primary example of shear flow. Inserting $\bar{x} = x/H$, $\bar{y} = y/H$, $\bar{z} = z/H$, $\bar{u} = u/U$, $\bar{p} = Hp/(\mu U)$, and $\bar{t} = H^2/\nu$ in the equations results in the scaled Navier–Stokes equations (dropping bars after the scaling):

$$\frac{\partial u}{\partial t} + \mathrm{Re}\, u \cdot \nabla u = -\nabla p + \nabla^2 u,$$
$$\nabla \cdot u = 0.$$

Here, $\mathrm{Re} = \varrho U H/\mu$ is the Reynolds number. Because of the time and pressure scales, which are different from convection-dominated fluid flow, the Reynolds number is associated with the convective term and not the viscosity term.

The exact solution is derived by assuming $u = (u_x(x,y,z), 0, 0)$, with the $x$ axis pointing along the channel. Since $\nabla \cdot u = 0$, $u$ cannot depend on $x$.

The physics of channel flow is also two-dimensional so we can omit the $z$ coordinate (more precisely: $\partial/\partial z = 0$). Inserting $u = (u_x, 0, 0)$ in the (scaled) governing equations gives $u_x''(y) = \partial p/\partial x$. Differentiating this equation with respect to $x$ shows that $\partial^2 p/\partial^2 x = 0$ so $\partial p/\partial x$ is a constant, here called $-\beta$. This is the driving force of the flow and can be specified as a known parameter in the problem. Integrating $u_x''(y) = -\beta$ over the width of the channel, $[0,1]$, and requiring $u = (0,0,0)$ at the channel walls, results in $u_x = \frac{1}{2}\beta y(1-y)$. The characteristic inlet velocity $U$ can be taken as the maximum inflow at $y = 1/2$, implying $\beta = 8$. The length of the channel, $L/H$ in the scaled model, has no impact on the result, so for simplicity we just compute on the unit square. Mathematically, the pressure must be prescribed at a point, but since $p$ does not depend on $y$, we can set $p$ to a known value, e.g. zero, along the outlet boundary $x = 1$. The result is $p(x) = 8(1-x)$ and $u_x = 4y(1-y)$.

The boundary conditions can be set as $p = 8$ at $x = 0$, $p = 0$ at $x = 1$ and $u = (0,0,0)$ on the walls $y = 0, 1$. This defines the pressure drop and should result in unit maximum velocity at the inlet and outlet and a parabolic velocity profile without further specifications. Note that it is only meaningful to solve the Navier–Stokes equations in 2D or 3D geometries, although the underlying mathematical problem collapses to two 1D problems, one for $u_x(y)$ and one for $p(x)$.

The scaled model is not so easy to simulate using a standard Navier–Stokes solver with dimensions. However, one can argue that the convection term is zero, so the Re coefficient in front of this term in the scaled PDEs is not important and can be set to unity. In that case, setting $\varrho = \mu = 1$ in the original Navier–Stokes equations resembles the scaled model.

For a specific engineering problem one wants to simulate a specific fluid and set corresponding parameters. A general solver is therefore most naturally implemented with dimensions and the original physical parameters. However, scaling may greatly simplify numerical simulations. First of all, it shows that all fluids behave in the same way: it does not matter whether we have oil, gas, or water flowing between two plates, and it does not matter how fast the flow is (up to some criticial value of the Reynolds number where the flow becomes unstable and transitions to a complicated turbulent flow of totally different nature). This means that one simulation is enough to cover all types of channel flow! In other applications, scaling shows that it might be necessary to set just the fraction of some parameters (dimensionless numbers) rather than the parameters themselves. This simplifies exploring the input parameter space which is often the purpose of simulation. Frequently, the scaled problem is run by setting some of the input parameters with dimension to fixed values (often unity).

**FEniCS implementation.** Our previous examples have all started out with the creation of a mesh and then the definition of a `FunctionSpace` on the mesh. For the Navier–Stokes splitting scheme we will need to define two function spaces, one for the velocity and one for the pressure:

```
V = VectorFunctionSpace(mesh, 'P', 2)
Q = FunctionSpace(mesh, 'P', 1)
```

The first space `V` is a vector-valued function space for the velocity and the second space `Q` is a scalar-valued function space for the pressure. We use piecewise quadratic elements for the velocity and piecewise linear elements for the pressure. When creating a `VectorFunctionSpace` in FEniCS, the value-dimension (the length of the vectors) will be set equal to the geometric dimension of the finite element mesh. One can easily create vector-valued function spaces with other dimensions in FEniCS by adding the keyword parameter `dim`:

```
V = VectorFunctionSpace(mesh, 'P', 2, dim=10)
```

**Stable finite element spaces for the Navier–Stokes equations**

It is well-known that certain finite element spaces are not *stable* for the Navier–Stokes equations, or even for the simpler Stokes equations. The prime example of an unstable pair of finite element spaces is to use first degree continuous piecewise polynomials for both the velocity and the pressure. Using an unstable pair of spaces typically results in a solution with *spurious* (unwanted, non-physical) oscillations in the pressure solution. The simple remedy is to use continuous piecewise quadratic elements for the velocity and continuous piecewise linear elements for the pressure. Together, these elements form the so-called *Taylor-Hood* element. Spurious oscillations may occur also for splitting methods if an unstable element pair is used.

Since we have two different function spaces, we need to create two sets of trial and test functions:

```
u = TrialFunction(V)
v = TestFunction(V)
p = TrialFunction(Q)
q = TestFunction(Q)
```

As we have seen in previous examples, boundaries may be defined in FEniCS by defining Python functions that return `True` or `False` depending on whether a point should be considered part of the boundary, for example

```
def boundary(x, on_boundary):
    return near(x[0], 0)
```

This function defines the boundary to be all points with $x$-coordinate equal to (near) zero. The `near` function comes from FEniCS and performs a test with tolerance: `abs(x[0] - 0) < 3E-16` so we do not run into rounding troubles. Alternatively, we may give the boundary definition as a

string of C++ code, much like we have previously defined expressions such as `u_D = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]', degree=2)`. The above definition of the boundary in terms of a Python function may thus be replaced by a simple C++ string:

```
boundary = 'near(x[0], 0)'
```

This has the advantage of moving the computation of which nodes belong to the boundary from Python to C++, which improves the efficiency of the program.

For the current example, we will set three different boundary conditions. First, we will set $u = 0$ at the walls of the channel; that is, at $y = 0$ and $y = 1$. Second, we will set $p = 8$ at the inflow ($x = 0$) and, finally, $p = 0$ at the outflow ($x = 1$). This will result in a pressure gradient that will accelerate the flow from the initial state with zero velocity. These boundary conditions may be defined as follows:

```
# Define boundaries
inflow   = 'near(x[0], 0)'
outflow  = 'near(x[0], 1)'
walls    = 'near(x[1], 0) || near(x[1], 1)'

# Define boundary conditions
bcu_noslip  = DirichletBC(V, Constant((0, 0)), walls)
bcp_inflow  = DirichletBC(Q, Constant(8), inflow)
bcp_outflow = DirichletBC(Q, Constant(0), outflow)
bcu = [bcu_noslip]
bcp = [bcp_inflow, bcp_outflow]
```

At the end, we collect the boundary conditions for the velocity and pressure in Python lists so we can easily access them in the following computation.

We now move on to the definition of the variational forms. There are three variational problems to be defined, one for each step in the IPCS scheme. Let us look at the definition of the first variational problem. We start with some constants:

```
U   = 0.5*(u_n + u)
n   = FacetNormal(mesh)
f   = Constant((0, 0))
k   = Constant(dt)
mu  = Constant(mu)
rho = Constant(rho)
```

The next step is to set up the variational form for the first step (3.32) in the solution process. Since the variational problem contains a mix of known and unknown quantities we will use the following naming convention: `u` is the unknown (mathematically $u^{n+1}$) as a trial function in the variational form, `u_` is the most recently computed approximation ($u^{n+1}$ available as a `Function` object), `u_n` is $u^n$, and the same convention goes for `p`, `p_` ($p^{n+1}$), and `p_n` ($p^n$).

```
# Define strain-rate tensor
def epsilon(u):
    return sym(nabla_grad(u))

# Define stress tensor
def sigma(u, p):
    return 2*mu*epsilon(u) - p*Identity(len(u))

# Define variational problem for step 1
F1 = rho*dot((u - u_n) / k, v)*dx + \
     rho*dot(dot(u_n, nabla_grad(u_n)), v)*dx \
   + inner(sigma(U, p_n), epsilon(v))*dx \
   + dot(p_n*n, v)*ds - dot(mu*nabla_grad(U)*n, v)*ds \
   - dot(f, v)*dx
a1 = lhs(F1)
L1 = rhs(F1)
```

Note that we take advantage of the Python programming language to define our own operators `sigma` and `epsilon`. Using Python this way makes it easy to extend the mathematical language of FEniCS with special operators and constitutive laws.

Also note that FEniCS can sort out the bilinear form $a(u,v)$ and linear form $L(v)$ forms by the `lhs` and `rhs` functions. This is particularly convenient in longer and more complicated variational forms.

The splitting scheme requires the solution of a sequence of three variational problems in each time step. We have previously used the built-in FEniCS function `solve` to solve variational problems. Under the hood, when a user calls `solve(a == L, u, bc)`, FEniCS will perform the following steps:

```
A = assemble(A)
b = assemble(L)
bc.apply(A, b)
solve(A, u.vector(), b)
```

In the last step, FEniCS uses the overloaded `solve` function to solve the linear system `AU = b` where `U` is the vector of degrees of freedom for the function $u(x) = \sum_{j=1} U_j \phi_j(x)$.

In our implementation of the splitting scheme, we will make use of these low-level commands to first assemble and then call solve. This has the advantage that we may control when we assemble and when we solve the linear system. In particular, since the matrices for the three variational problems are all time-independent, it makes sense to assemble them once and for all outside of the time-stepping loop:

```
A1 = assemble(a1)
A2 = assemble(a2)
A3 = assemble(a3)
```

Within the time-stepping loop, we may then assemble only the right-hand side vectors, apply boundary conditions, and call the solve function as here for the first of the three steps:

```
# Time-stepping
t = 0
for n in range(num_steps):

    # Update current time
    t += dt

    # Step 1: Tentative velocity step
    b1 = assemble(L1)
    [bc.apply(b1) for bc in bcu]
    solve(A1, u_.vector(), b1)
```

Notice the Python *list comprehension* `[bc.apply(b1) for bc in bcu]` which iterates over all `bc` in the list `bcu`. This is a convenient and compact way to construct a loop that applies all boundary conditions in a single line. Also, the code works if we add more Dirichlet boundary conditions in the future. Note that the boundary conditions only need to be applied to the right-hand side vectors as they have already been applied to the matrices (not shown).

Finally, let us look at an important detail in how we use parameters such as the time step `dt` in the definition of our variational problems. Since we might want to change these later, for example if we want to experiment with smaller or larger time steps, we wrap these using a FEniCS `Constant`:

```
k = Constant(dt)
```

The assembly of matrices and vectors in FEniCS is based on code generation. This means that whenever we change a variational problem, FEniCS will have to generate new code, which may take a little time. New code will also be generated and compiled when a float value for the time step is changed. By wrapping this parameter using `Constant`, FEniCS will treat the parameter as a generic constant and not as a specific numerical value, which prevents repeated code generation. In the case of the time step, we choose a new name `k` instead of `dt` for the `Constant` since we also want to use the variable `dt` as a Python float as part of the time-stepping.

The complete code for simulating 2D channel flow with FEniCS can be found in the file `ft07_navier_stokes_channel.py`.

**Verification.** We compute the error at the nodes as we have done before to verify that our implementation is correct. Our Navier–Stokes solver computes the solution to the time-dependent incompressible Navier–Stokes equations, starting from the initial condition $u = (0,0)$. We have not specified the initial condition explicitly in our solver which means that FEniCS will initialize all variables, in particular the previous and current velocities `u_n` and `u_`, to zero. Since the exact solution is quadratic, we expect the solution to be exact to within machine precision at the nodes at infinite time. For our implementation, the error quickly approaches zero and is approximately $10^{-6}$ at time $T = 10$.

**Test problem 2: Flow past a cylinder.** We now turn our attention to a more challenging problem: flow past a circular cylinder. The geometry and pa-

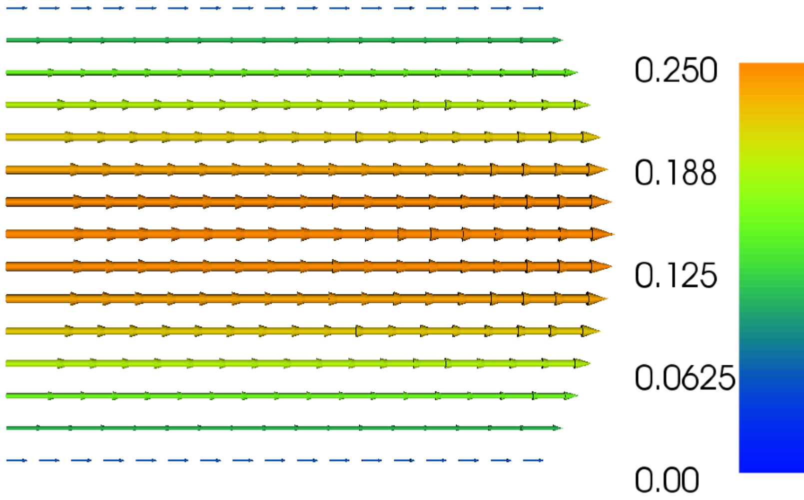


**Fig. 3.3** Plot of the velocity profile at the final time for the Navier–Stokes channel flow example.

rameters are taken from problem DFG 2D-2 in the FEATFLOW/1995-DFG benchmark suite[1] and is illustrated in Figure 3.4. The kinematic viscosity is given by $\nu = 0.001 = \mu/\varrho$ and the inflow velocity profile is specified as

$$u(x,y,t) = \left(1.5 \cdot \frac{4y(0.41-y)}{0.41^2}, 0\right),$$

which has a maximum magnitude of 1.5 at $y = 0.41/2$. We do not use any scaling for this problem since all exact parameters are known.

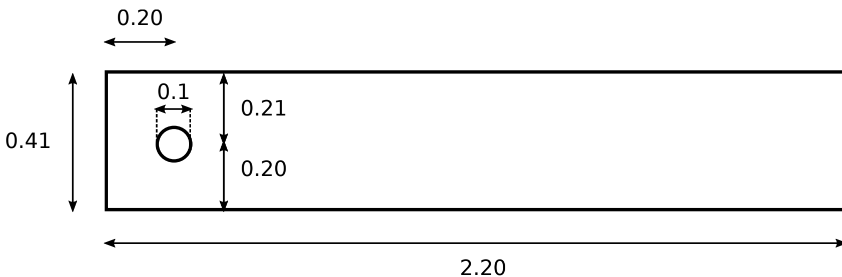


**Fig. 3.4** Geometry for the flow past a cylinder test problem. Notice the slightly perturbed and unsymmetric geometry.

[1] `http://www.featflow.de/en/benchmarks/cfdbenchmarking/flow/dfg_benchmark2_re100.html`

**FEniCS implementation.** So far all our domains have been simple shapes such as a unit square or a rectangular box. A number of such simple meshes may be created using the built-in mesh classes in FEniCS (`UnitIntervalMesh`, `UnitSquareMesh`, `UnitCubeMesh`, `IntervalMesh`, `RectangleMesh`, `BoxMesh`). FEniCS supports the creation of more complex meshes via a technique called *constructive solid geometry* (CSG), which lets us define geometries in terms of simple shapes (primitives) and set operations: union, intersection, and set difference. The set operations are encoded in FEniCS using the operators `+` (union), `*` (intersection), and `-` (set difference). To access the CSG functionality in FEniCS, one must import the FEniCS module `mshr` which provides the extended meshing functionality of FEniCS.

The geometry for the cylinder flow test problem can be defined easily by first defining the rectangular channel and then subtracting the circle:

```
channel = Rectangle(Point(0, 0), Point(2.2, 0.41))
cylinder = Circle(Point(0.2, 0.2), 0.05)
domain = channel - cylinder
```

We may then create the mesh by calling the function `generate_mesh`:

```
mesh = generate_mesh(domain, 64)
```

Here the argument `64` indicates that we want to resolve the geometry with 64 cells across its diameter (the channel length).

To solve the cylinder test problem, we only need to make a few minor changes to the code we wrote for the channel flow test case. Besides defining the new mesh, the only change we need to make is to modify the boundary conditions and the time step size. The boundaries are specified as follows:

```
inflow   = 'near(x[0], 0)'
outflow  = 'near(x[0], 2.2)'
walls    = 'near(x[1], 0) || near(x[1], 0.41)'
cylinder = 'on_boundary && x[0]>0.1 && x[0]<0.3 && x[1]>0.1 && x[1]<0.3'
```

The last line may seem cryptic before you catch the idea: we want to pick out all boundary points (`on_boundary`) that also lie within the 2D domain $[0.1, 0.3] \times [0.1, 0.3]$, see Figure 3.4. The only possible points are then the points on the circular boundary!

In addition to these essential changes, we will make a number of small changes to improve our solver. First, since we need to choose a relatively small time step to compute the solution (a time step that is too large will make the solution blow up) we add a progress bar so that we can follow the progress of our computation. This can be done as follows:

```
progress = Progress('Time-stepping')
set_log_level(PROGRESS)

# Time-stepping
t = 0.0
for n in range(num_steps):
```

```
    # Update current time
    t += dt

    # Place computation here

    # Update progress bar
    progress.update(t / T)
```

**Log levels and printing in FEniCS**

Notice the call to `set_log_level(PROGRESS)` which is essential to make FEniCS actually display the progress bar. FEniCS is actually quite informative about what is going on during a computation but the amount of information printed to screen depends on the current log level. Only messages with a priority higher than or equal to the current log level will be displayed. The predefined log levels in FEniCS are `DBG`, `TRACE`, `PROGRESS`, `INFO`, `WARNING`, `ERROR`, and `CRITICAL`. By default, the log level is set to `INFO` which means that messages at level `DBG`, `TRACE`, and `PROGRESS` will not be printed. Users may print messages using the FEniCS functions `info`, `warning`, and `error` which will print messages at the obvious log level (and in the case of `error` also throw an exception and exit). One may also use the call `log(level, message)` to print a message at a specific log level.

Since the system(s) of linear equations are significantly larger than for the simple channel flow test problem, we choose to use an iterative method instead of the default direct (sparse) solver used by FEniCS when calling `solve`. Efficient solution of linear systems arising from the discretization of PDEs requires the choice of both a good iterative (Krylov subspace) method and a good preconditioner. For this problem, we will simply use the biconjugate gradient stabilized method (BiCGSTAB) and the conjugate gradient method. This can be done by adding the keywords `bicgstab` or `cg` in the call to `solve`. We also specify suitable preconditioners to speed up the computations:

```
solve(A1, u1.vector(), b1, 'bicgstab', 'hypre_amg')
solve(A2, p1.vector(), b2, 'bicgstab', 'hypre_amg')
solve(A3, u1.vector(), b3, 'cg', 'sor')
```

Finally, to be able to postprocess the computed solution in ParaView, we store the solution to a file in each time step. We have previously created files with the suffix `.pvd` for this purpose. In the example program `ft04_heat_gaussian.py`, we first created a file named `heat_gaussian/solution.pvd` and then saved the solution in each time step using

```
vtkfile << (u, t)
```

For the present example, we will instead choose to save the solution to XDMF format. This file format works similarly to the .pvd files we have seen earlier but has several advantages. First, the storage is much more efficient, both in terms of speed and file sizes. Second, .xdmf files work in parallell, both for writing and reading (postprocessing). Much like .pvd files, the actual data will not be stored in the .xdmf file itself, but will instead be stored in a (single) separate data file with the suffix .hdf5 which is an advanced file format designed for high-performance computing. We create the XDMF files as follows:

```
xdmffile_u = XDMFFile('navier_stokes_cylinder/velocity.xdmf')
xdmffile_p = XDMFFile('navier_stokes_cylinder/pressure.xdmf')
```

In each time step, we may then store the velocity and pressure by

```
xdmffile_u.write(u, t)
xdmffile_p.write(p, t)
```

We also store the solution using a FEniCS TimeSeries. This allows us to store the solution not for visualization, but for later reuse in a computation as we will see in the next section. Using a TimeSeries it is easy and efficient to read in solutions from certain points in time during a simulation. The TimeSeries class also uses the HDF5 file format for efficient storage and access to data.

Figures 3.5 and 3.6 show the velocity and pressure at final time visualized in ParaView. For the visualization of the velocity, we have used the **Glyph** filter to visualize the vector velocity field. For the visualization of the pressure, we have used the **Warp By Scalar** filter.

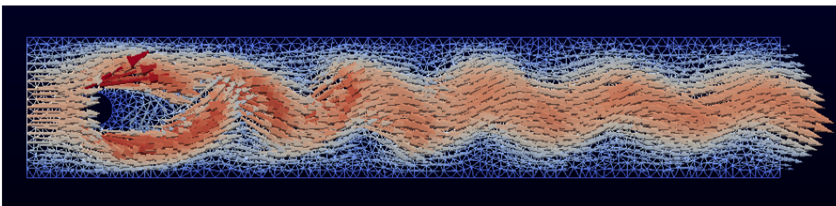

**Fig. 3.5** Plot of the velocity for the cylinder test problem at final time.

The complete code for the cylinder test problem looks as follows:

```
from fenics import *
from mshr import *
import numpy as np

T = 5.0            # final time
num_steps = 5000   # number of time steps
dt = T / num_steps # time step size
mu = 0.001         # dynamic viscosity
```

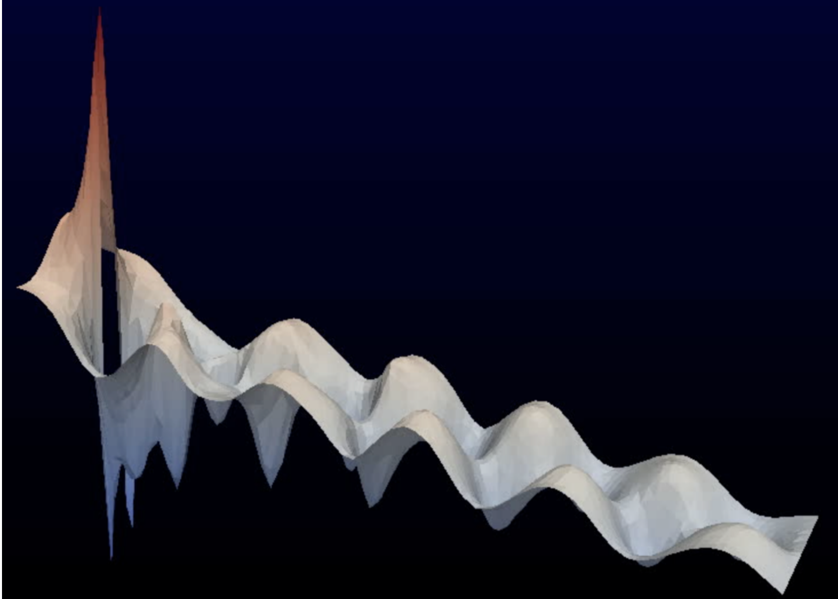

**Fig. 3.6** Plot of the pressure for the cylinder test problem at final time.

```
rho = 1            # density

# Create mesh
channel = Rectangle(Point(0, 0), Point(2.2, 0.41))
cylinder = Circle(Point(0.2, 0.2), 0.05)
domain = channel - cylinder
mesh = generate_mesh(domain, 64)

# Define function spaces
V = VectorFunctionSpace(mesh, 'P', 2)
Q = FunctionSpace(mesh, 'P', 1)

# Define boundaries
inflow   = 'near(x[0], 0)'
outflow  = 'near(x[0], 2.2)'
walls    = 'near(x[1], 0) || near(x[1], 0.41)'
cylinder = 'on_boundary && x[0]>0.1 && x[0]<0.3 && x[1]>0.1 && x[1]<0.3'

# Define inflow profile
inflow_profile = ('4.0*1.5*x[1]*(0.41 - x[1]) / pow(0.41, 2)', '0')

# Define boundary conditions
bcu_inflow = DirichletBC(V, Expression(inflow_profile, degree=2), inflow)
bcu_walls = DirichletBC(V, Constant((0, 0)), walls)
bcu_cylinder = DirichletBC(V, Constant((0, 0)), cylinder)
bcp_outflow = DirichletBC(Q, Constant(0), outflow)
bcu = [bcu_inflow, bcu_walls, bcu_cylinder]
```

```
bcp = [bcp_outflow]

# Define trial and test functions
u = TrialFunction(V)
v = TestFunction(V)
p = TrialFunction(Q)
q = TestFunction(Q)

# Define functions for solutions at previous and current time steps
u_n = Function(V)
u_  = Function(V)
p_n = Function(Q)
p_  = Function(Q)

# Define expressions used in variational forms
U   = 0.5*(u_n + u)
n   = FacetNormal(mesh)
f   = Constant((0, 0))
k   = Constant(dt)
mu  = Constant(mu)

# Define symmetric gradient
def epsilon(u):
    return sym(nabla_grad(u))

# Define stress tensor
def sigma(u, p):
    return 2*mu*epsilon(u) - p*Identity(len(u))

# Define variational problem for step 1
F1 = rho*dot((u - u_n) / k, v)*dx \
   + rho*dot(dot(u_n, nabla_grad(u_n)), v)*dx \
   + inner(sigma(U, p_n), epsilon(v))*dx \
   + dot(p_n*n, v)*ds - dot(mu*nabla_grad(U)*n, v)*ds \
   - dot(f, v)*dx
a1 = lhs(F1)
L1 = rhs(F1)

# Define variational problem for step 2
a2 = dot(nabla_grad(p), nabla_grad(q))*dx
L2 = dot(nabla_grad(p_n), nabla_grad(q))*dx - (1/k)*div(u_)*q*dx

# Define variational problem for step 3
a3 = dot(u, v)*dx
L3 = dot(u_, v)*dx - k*dot(nabla_grad(p_ - p_n), v)*dx

# Assemble matrices
A1 = assemble(a1)
A2 = assemble(a2)
A3 = assemble(a3)

# Apply boundary conditions to matrices
[bc.apply(A1) for bc in bcu]
[bc.apply(A2) for bc in bcp]
```

```
# Create XDMF files for visualization output
xdmffile_u = XDMFFile('navier_stokes_cylinder/velocity.xdmf')
xdmffile_p = XDMFFile('navier_stokes_cylinder/pressure.xdmf')

# Create time series (for use in reaction_system.py)
timeseries_u = TimeSeries('navier_stokes_cylinder/velocity_series')
timeseries_p = TimeSeries('navier_stokes_cylinder/pressure_series')

# Save mesh to file (for use in reaction_system.py)
File('navier_stokes_cylinder/cylinder.xml.gz') << mesh

# Create progress bar
progress = Progress('Time-stepping')
set_log_level(PROGRESS)

# Time-stepping
t = 0
for n in range(num_steps):

    # Update current time
    t += dt

    # Step 1: Tentative velocity step
    b1 = assemble(L1)
    [bc.apply(b1) for bc in bcu]
    solve(A1, u_.vector(), b1, 'bicgstab', 'hypre_amg')

    # Step 2: Pressure correction step
    b2 = assemble(L2)
    [bc.apply(b2) for bc in bcp]
    solve(A2, p_.vector(), b2, 'bicgstab', 'hypre_amg')

    # Step 3: Velocity correction step
    b3 = assemble(L3)
    solve(A3, u_.vector(), b3, 'cg', 'sor')

    # Plot solution
    plot(u_, title='Velocity')
    plot(p_, title='Pressure')

    # Save solution to file (XDMF/HDF5)
    xdmffile_u.write(u_, t)
    xdmffile_p.write(p_, t)

    # Save nodal values to file
    timeseries_u.store(u_.vector(), t)
    timeseries_p.store(p_.vector(), t)

    # Update previous solution
    u_n.assign(u_)
    p_n.assign(p_)

    # Update progress bar
```

```
    progress.update(t / T)
    print('u max:', u_.vector().array().max())

# Hold plot
interactive()
```

This program can be found in the file `ft08_navier_stokes_cylinder.py`. The reader should be advised that this example program is considerably more demanding than our previous examples in terms of CPU time and memory, but it should be possible to run the program on a reasonably modern laptop.

## 3.5 A system of advection–diffusion–reaction equations

The problems we have encountered so far—with the notable exception of the Navier–Stokes equations—all share a common feature: they all involve models expressed by a *single* scalar or vector PDE. In many situations the model is instead expressed as a system of PDEs, describing different quantities possibly governed by (very) different physics. As we saw for the Navier–Stokes equations, one way to solve a system of PDEs in FEniCS is to use a splitting method where we solve one equation at a time and feed the solution from one equation into the next. However, one of the strengths with FEniCS is the ease by which one can instead define variational problems that couple several PDEs into one compound system. In this section, we will look at how to use FEniCS to write solvers for such systems of coupled PDEs. The goal is to demonstrate how easy it is to implement fully implicit, also known as monolithic, solvers in FEniCS.

### 3.5.1 PDE problem

Our model problem is the following system of advection–diffusion–reaction equations:

$$\frac{\partial u_1}{\partial t} + w \cdot \nabla u_1 - \nabla \cdot (\epsilon \nabla u_1) = f_1 - K u_1 u_2, \tag{3.36}$$

$$\frac{\partial u_2}{\partial t} + w \cdot \nabla u_2 - \nabla \cdot (\epsilon \nabla u_2) = f_2 - K u_1 u_2, \tag{3.37}$$

$$\frac{\partial u_3}{\partial t} + w \cdot \nabla u_3 - \nabla \cdot (\epsilon \nabla u_3) = f_3 + K u_1 u_2 - K u_3. \tag{3.38}$$

This system models the chemical reaction between two species $A$ and $B$ in some domain $\Omega$:

$$A + B \to C.$$

We assume that the reaction is *first-order*, meaning that the reaction rate is proportional to the concentrations $[A]$ and $[B]$ of the two species $A$ and $B$:

$$\frac{\mathrm{d}}{\mathrm{d}t}[C] = K[A][B].$$

We also assume that the formed species $C$ spontaneously decays with a rate proportional to the concentration $[C]$. In the PDE system (3.36)–(3.38), we use the variables $u_1$, $u_2$, and $u_3$ to denote the concentrations of the three species:

$$u_1 = [A], \quad u_2 = [B], \quad u_3 = [C].$$

We see that the chemical reactions are accounted for in the right-hand sides of the PDE system (3.36)–(3.38).

The chemical reactions take part at each point in the domain $\Omega$. In addition, we assume that the species $A$, $B$, and $C$ diffuse throughout the domain with diffusivity $\epsilon$ (the terms $-\nabla \cdot (\epsilon \nabla u_i)$) and are advected with velocity $w$ (the terms $w \cdot \nabla u_i$).

To make things visually and physically interesting, we shall let the chemical reaction take place in the velocity field computed from the solution of the incompressible Navier–Stokes equations around a cylinder from the previous section. In summary, we will thus be solving the following coupled system of nonlinear PDEs:

$$\varrho\left(\frac{\partial w}{\partial t} + w \cdot \nabla w\right) = \nabla \cdot \sigma(w,p) + f, \tag{3.39}$$

$$\nabla \cdot w = 0, \tag{3.40}$$

$$\frac{\partial u_1}{\partial t} + w \cdot \nabla u_1 - \nabla \cdot (\epsilon \nabla u_1) = f_1 - K u_1 u_2, \tag{3.41}$$

$$\frac{\partial u_2}{\partial t} + w \cdot \nabla u_2 - \nabla \cdot (\epsilon \nabla u_2) = f_2 - K u_1 u_2, \tag{3.42}$$

$$\frac{\partial u_3}{\partial t} + w \cdot \nabla u_3 - \nabla \cdot (\epsilon \nabla u_3) = f_3 + K u_1 u_2 - K u_3. \tag{3.43}$$

We assume that $u_1 = u_2 = u_3 = 0$ at $t = 0$ and inject the species $A$ and $B$ into the system by specifying nonzero source terms $f_1$ and $f_2$ close to the corners at the inflow, and take $f_3 = 0$. The result will be that $A$ and $B$ are convected by advection and diffusion throughout the channel, and when they mix the species $C$ will be formed.

Since the system is one-way coupled from the Navier–Stokes subsystem to the advection–diffusion–reaction subsystem, we do not need to recompute the solution to the Navier–Stokes equations, but can just read back the previously

computed velocity field $w$ and feed it into our equations. But we *do* need to learn how to read and write solutions for time-dependent PDE problems.

### 3.5.2 Variational formulation

We obtain the variational formulation of our system by multiplying each equation by a test function, integrating the second-order terms $-\nabla\cdot(\epsilon\nabla u_i)$ by parts, and summing up the equations. When working with FEniCS it is convenient to think of the PDE system as a vector of equations. The test functions are collected in a vector too, and the variational formulation is the inner product of the vector PDE and the vector test function.

We also need introduce some discretization in time. We will use the backward Euler method as before when we solved the heat equation and approximate the time derivatives by $(u_i^{n+1}-u_i^n)/\Delta t$. Let $v_1$, $v_2$, and $v_3$ be the test functions, or the components of the test vector function. The inner product results in

$$
\begin{aligned}
&\int_\Omega (\Delta t^{-1}(u_1^{n+1}-u_1^n)v_1 + w\cdot\nabla u_1^{n+1}\,v_1 + \epsilon\nabla u_1^{n+1}\cdot\nabla v_1)\,\mathrm{d}x \\
+&\int_\Omega (\Delta t^{-1}(u_2^{n+1}-u_2^n)v_2 + w\cdot\nabla u_2^{n+1}\,v_2 + \epsilon\nabla u_2^{n+1}\cdot\nabla v_2)\,\mathrm{d}x \\
+&\int_\Omega (\Delta t^{-1}(u_3^{n+1}-u_3^n)v_3 + w\cdot\nabla u_3^{n+1}\,v_3 + \epsilon\nabla u_3^{n+1}\cdot\nabla v_3)\,\mathrm{d}x \\
-&\int_\Omega (f_1v_1 + f_2v_2 + f_3v_3)\,\mathrm{d}x \\
-&\int_\Omega (-Ku_1^{n+1}u_2^{n+1}v_1 - Ku_1^{n+1}u_2^{n+1}v_2 + Ku_1^{n+1}u_2^{n+1}v_3 - Ku_3^{n+1}v_3)\,\mathrm{d}x = 0.
\end{aligned}
\tag{3.44}
$$

For this problem it is natural to assume homogeneous Neumann boundary conditions on the entire boundary for $u_1$, $u_2$, and $u_3$; that is, $\partial u_i/\partial n = 0$ for $i = 1,2,3$. This means that the boundary terms vanish when we integrate by parts.

### 3.5.3 FEniCS implementation

The first step is to read the mesh from file. Luckily, we made sure to save the mesh to file in the Navier–Stokes example and can now easily read it back from file:

```
mesh = Mesh('navier_stokes_cylinder/cylinder.xml.gz')
```

The mesh is stored in the native FEniCS XML format (with additional gzipping to decrease the file size).

Next, we need to define the finite element function space. For this problem, we need to define several spaces. The first space we create is the space for the velocity field $w$ from the Navier–Stokes simulation. We call this space $W$ and define the space by

```
W = VectorFunctionSpace(mesh, 'P', 2)
```

It is important that this space is exactly the same as the space we used for the velocity field in the Navier–Stokes solver. To read the values for the velocity field, we use a `TimeSeries`:

```
timeseries_w = TimeSeries('navier_stokes_cylinder/velocity_series')
```

This will initialize the object `timeseries_w` which we will call later in the time-stepping loop to retrieve values from the file `velocity_series.h5` (in binary HDF5 format).

For the three concentrations $u_1$, $u_2$, and $u_3$, we want to create a *mixed space* with functions that represent the full system $(u_1, u_2, u_3)$ as a single entity. To do this, we need to define a `MixedElement` as the product space of three simple finite elements and then used the mixed element to define the function space:

```
P1 = FiniteElement('P', triangle, 1)
element = MixedElement([P1, P1, P1])
V = FunctionSpace(mesh, element)
```

**Mixed elements as products of elements**

FEniCS also allows finite elements to be defined as products of simple elements (or mixed elements). For example, the well-known Taylor–Hood element, with quadratic velocity components and linear pressure functions, may be defined as follows:

```
P2 = VectorElement('P', triangle, 2)
P1 = FiniteElement('P', triangle, 1)
TH = P2 * P1
```

This syntax works great for two elements, but for three or more elements we meet a subtle issue in how the Python interpreter handles the `*` operator. For the reaction system, we create the mixed element by `element = MixedElement([P1, P1, P1])` and one would be tempted to write

```
element = P1 * P1 * P1
```

However, this is equivalent to writing `element = (P1 * P1) * P1` so the result will be a mixed element consisting of two subsystems, the first of which in turn consists of two scalar subsystems.

Finally, we remark that for the simple case of a mixed system consisting of three scalar elements as for the reaction system, the definition is in fact equivalent to using a standard vector-valued element:

```
element = VectorElement('P', triangle, 1, dim=3)
V = FunctionSpace(mesh, element)
```

Once the space has been created, we need to define our test functions and finite element functions. Test functions for a mixed function space can be created by replacing `TestFunction` by `TestFunctions`:

```
v_1, v_2, v_3 = TestFunctions(V)
```

Since the problem is nonlinear, we need to work with functions rather than trial functions for the unknowns. This can be done by using the corresponding `Functions` construction in FEniCS. However, as we will need to access the `Function` for the entire system itself, we first need to create that function and then access its components:

```
u = Function(V)
u_1, u_2, u_3 = split(u)
```

These functions will be used to represent the unknowns $u_1$, $u_2$, and $u_3$ at the new time level $n+1$. The corresponding values at the previous time level $n$ are denoted by `u_n1`, `u_n2`, and `u_n3` in our program.

When now all functions and test functions have been defined, we can express the nonlinear variational problem (3.44):

```
F = ((u_1 - u_n1) / k)*v_1*dx + dot(w, grad(u_1))*v_1*dx \
  + eps*dot(grad(u_1), grad(v_1))*dx + K*u_1*u_2*v_1*dx  \
  + ((u_2 - u_n2) / k)*v_2*dx + dot(w, grad(u_2))*v_2*dx \
  + eps*dot(grad(u_2), grad(v_2))*dx + K*u_1*u_2*v_2*dx  \
  + ((u_3 - u_n3) / k)*v_3*dx + dot(w, grad(u_3))*v_3*dx \
  + eps*dot(grad(u_3), grad(v_3))*dx - K*u_1*u_2*v_3*dx + K*u_3*v_3*dx \
  - f_1*v_1*dx - f_2*v_2*dx - f_3*v_3*dx
```

The time-stepping simply consists of solving this variational problem in each time step by a call to the `solve` function:

```
t = 0
for n in range(num_steps):
    t += dt
    timeseries_w.retrieve(w.vector(), t)
    solve(F == 0, u)
    u_n.assign(u)
```

In each time step, we first read the current value for the velocity field from the time series we have previously stored. We then solve the nonlinear system,

and assign the computed values to the left-hand side values for the next time interval. When retrieving values from a `TimeSeries`, the values will by default be interpolated (linearly) to the given time `t` if the time does not exactly match a sample in the series.

The solution at the final time is shown in Figure 3.7. We clearly see the advection of the species $A$ and $B$ and the formation of $C$ along the center of the channel where $A$ and $B$ meet.

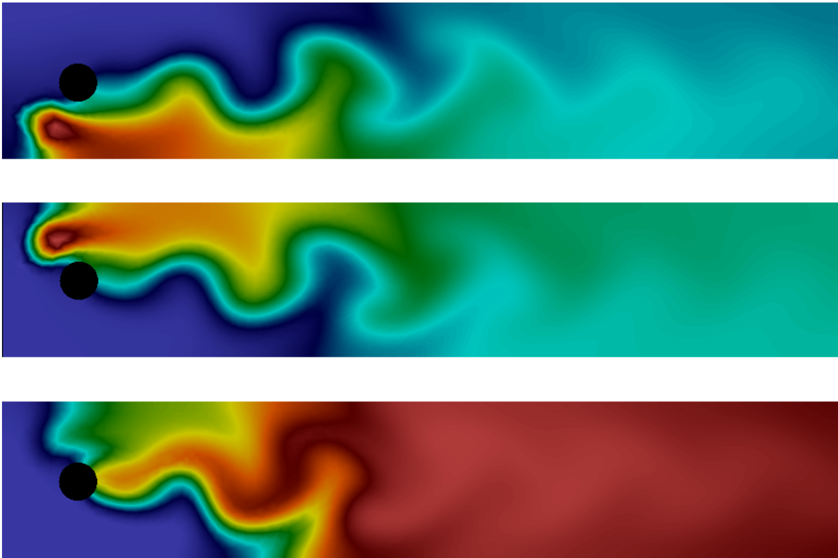

**Fig. 3.7** Plot of the concentrations of the three species $A$, $B$, and $C$ (from top to bottom) at final time.

The complete code is presented below.

```
from fenics import *

T = 5.0            # final time
num_steps = 500    # number of time steps
dt = T / num_steps # time step size
eps = 0.01         # diffusion coefficient
K = 10.0           # reaction rate

# Read mesh from file
mesh = Mesh('navier_stokes_cylinder/cylinder.xml.gz')

# Define function space for velocity
W = VectorFunctionSpace(mesh, 'P', 2)

# Define function space for system of concentrations
P1 = FiniteElement('P', triangle, 1)
```

```
element = MixedElement([P1, P1, P1])
V = FunctionSpace(mesh, element)

# Define test functions
v_1, v_2, v_3 = TestFunctions(V)

# Define functions for velocity and concentrations
w = Function(W)
u = Function(V)
u_n = Function(V)

# Split system functions to access components
u_1, u_2, u_3 = split(u)
u_n1, u_n2, u_n3 = split(u_n)

# Define source terms
f_1 = Expression('pow(x[0]-0.1,2)+pow(x[1]-0.1,2)<0.05*0.05 ? 0.1 : 0',
                 degree=1)
f_2 = Expression('pow(x[0]-0.1,2)+pow(x[1]-0.3,2)<0.05*0.05 ? 0.1 : 0',
                 degree=1)
f_3 = Constant(0)

# Define expressions used in variational forms
k = Constant(dt)
K = Constant(K)
eps = Constant(eps)

# Define variational problem
F = ((u_1 - u_n1) / k)*v_1*dx + dot(w, grad(u_1))*v_1*dx \
  + eps*dot(grad(u_1), grad(v_1))*dx + K*u_1*u_2*v_1*dx  \
  + ((u_2 - u_n2) / k)*v_2*dx + dot(w, grad(u_2))*v_2*dx \
  + eps*dot(grad(u_2), grad(v_2))*dx + K*u_1*u_2*v_2*dx  \
  + ((u_3 - u_n3) / k)*v_3*dx + dot(w, grad(u_3))*v_3*dx \
  + eps*dot(grad(u_3), grad(v_3))*dx - K*u_1*u_2*v_3*dx + K*u_3*v_3*dx \
  - f_1*v_1*dx - f_2*v_2*dx - f_3*v_3*dx

# Create time series for reading velocity data
timeseries_w = TimeSeries('navier_stokes_cylinder/velocity_series')

# Create VTK files for visualization output
vtkfile_u_1 = File('reaction_system/u_1.pvd')
vtkfile_u_2 = File('reaction_system/u_2.pvd')
vtkfile_u_3 = File('reaction_system/u_3.pvd')

# Create progress bar
progress = Progress('Time-stepping')
set_log_level(PROGRESS)

# Time-stepping
t = 0
for n in range(num_steps):

    # Update current time
    t += dt
```

```
    # Read velocity from file
    timeseries_w.retrieve(w.vector(), t)

    # Solve variational problem for time step
    solve(F == 0, u)

    # Save solution to file (VTK)
    _u_1, _u_2, _u_3 = u.split()
    vtkfile_u_1 << (_u_1, t)
    vtkfile_u_2 << (_u_2, t)
    vtkfile_u_3 << (_u_3, t)

    # Update previous solution
    u_n.assign(u)

    # Update progress bar
    progress.update(t / T)

# Hold plot
interactive()
```

This example program can be found in the file `ft09_reaction_system.py`.

Finally, we comment on three important techniques that are very useful when working with systems of PDEs: setting initial conditions, setting boundary conditions, and extracting components of the system for plotting or postprocessing.

**Setting initial conditions for mixed systems.** In our example, we did not need to worry about setting an initial condition, since we start with $u_1 = u_2 = u_3 = 0$. This happens automatically in the code when we set `u_n = Function(V)`. This creates a `Function` for the whole system and all degrees of freedom are set to zero.

If we want to set initial conditions for the components of the system separately, the easiest solution is to define the initial conditions as a vector-valued `Expression` and then project (or interpolate) this to the `Function` representing the whole system. For example,

```
u_0 = Expression(('sin(x[0])', 'cos(x[0]*x[1])', 'exp(x[1])'), degree=1)
u_n = project(u_0, V)
```

This defines $u_1$, $u_2$, and $u_2$ to be the projections of $\sin x$, $\cos(xy)$, and $\exp(y)$, respectively.

**Setting boundary conditions for mixed systems.** In our example, we also did not need to worry about setting boundary conditions since we used a natural Neumann condition. If we want to set Dirichlet conditions for individual components of the system, this can be done as usual by the class `DirichletBC`, but we must specify for which subsystem we set the boundary condition. For example, to specify that $u_2$ should be equal to $xy$ on the boundary defined by `boundary`, we do

```
u_D = Expression('x[0]*x[1]', degree=1)
bc = DirichletBC(V.sub(1), u_D, boundary)
```

The object `bc` or a list of such objects containing different boundary conditions, can then be passed to the `solve` function as usual. Note that numbering starts at 0 in FEniCS so the subspace corresponding to $u_2$ is `V.sub(1)`.

**Accessing components of mixed systems.** If `u` is a `Function` defined on a mixed function space in FEniCS, there are several ways in which `u` can be *split* into components. Above we already saw an example of the first of these:

```
u_1, u_2, u_3 = split(u)
```

This extracts the components of `u` as *symbols* that can be used in a variational problem. The above statement is in fact equivalent to

```
u_1 = u[0]
u_2 = u[1]
u_3 = u[2]
```

Note that `u[0]` is not really a `Function` object, but merely a symbolic expression, just like `grad(u)` in FEniCS is a symbolic expression and not a `Function` representing the gradient. This means that `u_1`, `u_2`, `u_3` can be used in a variational problem, but cannot be used for plotting or postprocessing.

To access the components of `u` for plotting and saving the solution to file, we need to use a different variant of the `split` function:

```
u_1_, u_2_, u_3_ = u.split()
```

This returns three subfunctions as actual objects with access to the common underlying data stored in `u`, which makes plotting and saving to file possible. Alternatively, we can do

```
u_1_, u_2_, u_3_ = u.split(deepcopy=True)
```

which will create `u_1_`, `u_2_`, and `u_3_` as stand-alone `Function` objects, each holding a copy of the subfunction data extracted from `u`. This is useful in many situations but is not necessary for plotting and saving solutions to file.

**Open Access** This chapter is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, duplication, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the work's Creative Commons license, unless indicated otherwise in the credit line; if such material is not included in the work's Creative Commons license and the respective action is not permitted by statutory regulation, users will need to obtain permission from the license holder to duplicate, adapt or reproduce the material.